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

James L. McClelland

Printer-Friendly PDF Version

Second Edition, DRAFT

NOTE: Software currently works only on MATLAB versions R2013b and earlier.

See installation guide, **Appendix A**, for details.

Send comments and corrections to:
mcclelland@stanford.edu

Send software issues and bug reports to:
pdplab-support@stanford.edu

December 16, 2015

# 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. As of Fall, 2011, a new version of the software is under development. The present version of the handbook is consistent with the current version of the software, version 2.06, released in April, 2011. 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. Reports of software bugs or installation difficulties should be sent to pdplab-support@stanford.edu.

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
September, 2011

# 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

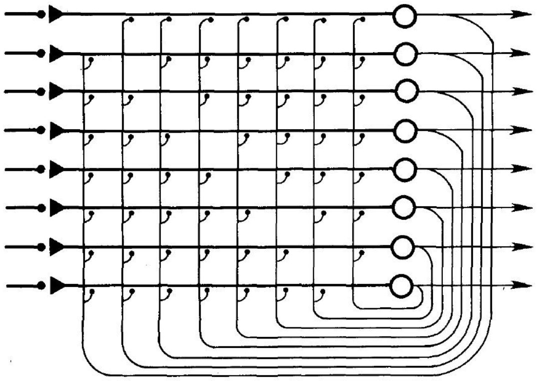


Figure 1.1: A simple PDP network consisting of one pool of units, and one projection, such that each unit recieves connections from all other units in the same pool. Each unit also can receive external input (shown coming in from the left). If this were a pool in a larger network, the units could receive 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 $i$th 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 $j$th column of the $i$th 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 $\Delta a_i$. For specificity, let's just suppose the net input has some fixed, positive value. Then we can see that $\Delta a_i$ will get smaller and smaller as the activation of the unit gets greater and greater. For some values of the unit's activation, $\Delta a_i$ will actually be negative. In particular, suppose that the unit's activation is equal to the resting level. Then $\Delta a_i$ is simply $(max - rest)net_i$. Now suppose that the unit's activation is equal to $max$, its maximum activation level. Then $\Delta a_i$ is simply $(-decay)(max - rest)$. Between these extremes there is an equilibrium value of $a_i$ at which $\Delta a_i$ is 0. We can find what the equilibrium value is by setting $\Delta a_i$ 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 $\gamma$ 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 PDPtool 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. In iac models, the connections are benerally bidirectionally symmetric, so that whenever there is an excitatory connection from unit $i$ to unit $j$, there is also an equal excitatory connection from unit $j$ back to unit $i$. IAC networks can, however, be created in which connections violate these characteristics of the model.

### 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) containing mutually inhibitory connections and also a projection from the other pool (projection 2 in each case) containing between-pool excitatory connections. These connections are bi-directionally symmatric.

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

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

and the symmetric return connection is

$$net.pool(3).projection(2).weight(j, i)$$

### 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

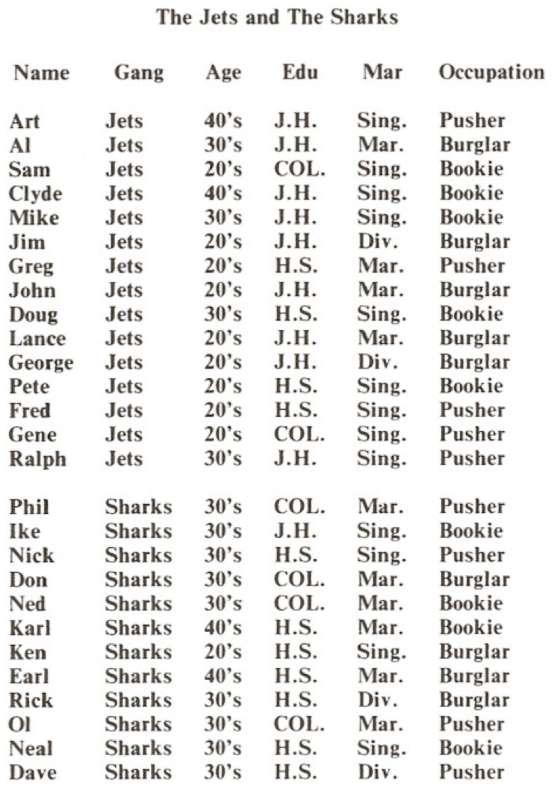
Use the **iac** program to examine how the mechanisms of interactive activation and competition can be used to illustrate the following properties of human memory:

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



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.)

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".

*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 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

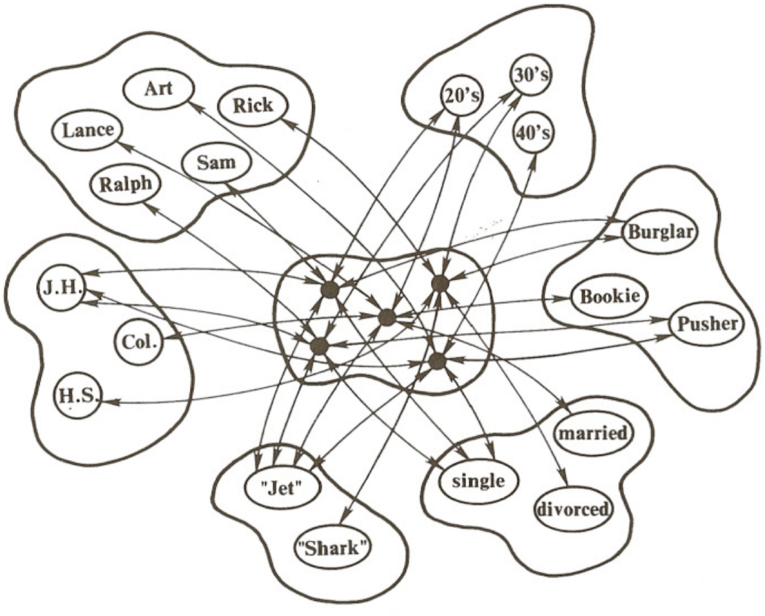


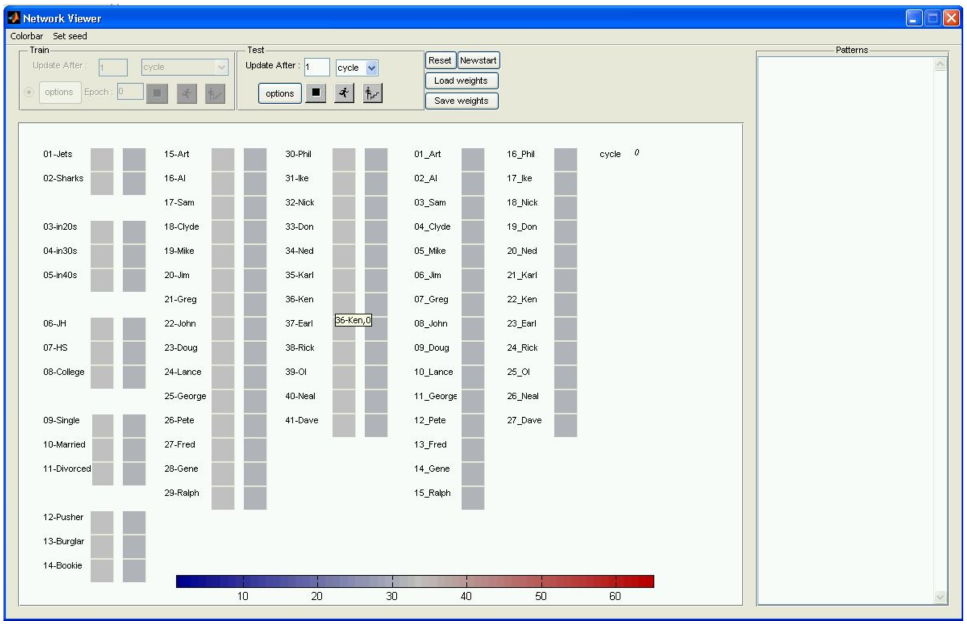
Figure 2.2: The units and connections for some of the individuals in Figure 2.1. (Two slight errors in the connections depicted in the original of this figure have been corrected in this version.) (From "Retrieving General and Specific Knowledge From Stored Knowledge of Specifics" by J. L. McClelland,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

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



Test: Update After : 1 cycle

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

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 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 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.

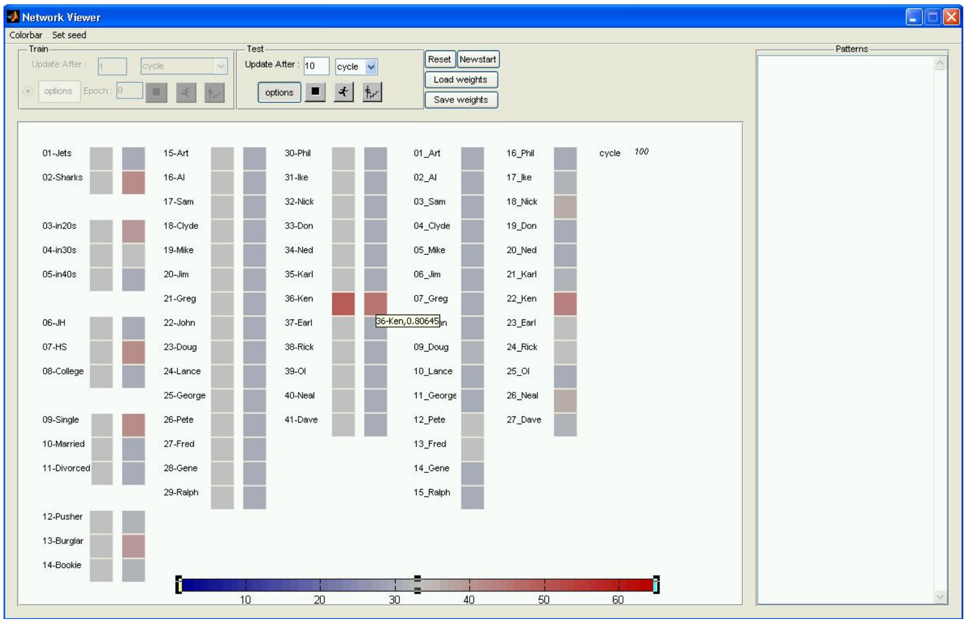


Figure 2.4: The display screen after 100 cycles with external input to the name unit for Ken.

*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. Appendix B and *The PDPTool User Guide* provide information on how to do this. You may wish to refer to the jets.m, jets.net, and jets.tem files for examples.

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.

# Chapter 3

# Constraint Satisfaction in PDP Systems

In the previous chapter we showed how PDP networks could be used for content-addressable memory retrieval, for prototype generation, for plausibly making default assignments for missing variables, and for spontaneously generalizing to novel inputs. In fact, these characteristics are reflections of a far more general process that many PDP models are capable of -namely, finding near-optimal solutions to problems with a large set of simultaneous constraints. This chapter introduces this constraint satisfaction process more generally and discusses two models for solving such problems. The specific models are the schema model, described in *PDP:14* and the Boltzmann machine, described in *PDP:7*. These models are embodied in the **cs** (constraint satisfaction) program. We begin with a general discussion of constraint satisfaction and some general results. We then turn to the schema model. We describe the general characteristics of the schema model, show how it can be accessed from **cs**, and offer a number of examples of it in operation. This is followed in turn by a discussion of the Boltzmann machine model.

## 3.1 BACKGROUND

Consider a problem whose solution involves the simultaneous satisfaction of a very large number of constraints. To make the problem more difficult, suppose that there may be no perfect solution in which all of the constraints are completely satisfied. In such a case, the solution would involve the satisfaction of as many constraints as possible. Finally, imagine that some constraints may be more important than others. In particular, suppose that each constraint has an importance value associated with it and that the solution to the problem involves the simultaneous satisfaction of as many of the most important of these constraints as possible. In general, this is a very difficult problem. It is what Minsky and Papert (1969) have called the best match problem. It is a problem that is central to much of cognitive science. It also happens to be one of the kinds of problems that PDP systems solve in a very natural way. Many of the chapters in the two PDP volumes pointed to the importance of this problem and to the kinds of solutions offered by PDP systems.

To our knowledge, Hinton was the first to sketch the basic idea for using parallel networks to solve constraint satisfaction problems (Hinton, 1977). Basically, such problems are translated into the language of PDP by assuming that each unit represents a hypothesis and each connection a constraint among hypotheses. Thus, for example, if whenever hypothesis A is true, hypothesis B is usually true, we would have a positive connection from unit A to unit B. If, on the other hand, hypothesis A provides evidence against hypothesis B, we would have a negative connection from unit A to unit B. PDP constraint networks are designed to deal with weak constraints (Blake, 1983), that is, with situations in which constraints constitute a set of desiderata that ought to be satisfied rather than a set of hard constraints that must be satisfied. The goal is to find a solution in which as many of the most important constraints are satisfied as possible. The importance of the constraint is reflected by the strength of the connection representing that constraint. If the constraint is very important, the weights are large. Less important constraints involve smaller weights. In addition, units may receive external input. We can think of the external input as providing direct evidence for certain hypotheses. Sometimes we say the input "clamps" a unit. This means that, in the solution, this particular unit must be on if the input is positive or must be off if the input is negative. Other times the input is not clamped but is graded. In this case, the input behaves as simply another weak constraint. Finally, different hypotheses may have different a priori probabilities. An appropriate solution to a constraint satisfaction problem must be able to reflect such prior information as well. This is done in PDP systems by assuming that each unit has a bias, which influences how likely the unit is to be on in the absence of other evidence. If a particular unit has a positive bias, then it is better to have the unit on; if it has a negative bias, there is a preference for it to be turned off.

We can now cast the constraint satisfaction problem described above in the following way. Let goodness of fit be the degree to which the desired constraints are satisfied. Thus, goodness of fit (or more simply goodness) depends on three things. First, it depends on the extent to which each unit satisfies the constraints imposed upon it by other units. Thus, if a connection between two units is positive, we say that the constraint is satisfied to the degree that both units are turned on. If the connection is negative, we can say that the constraint is violated to the degree that the units are turned on. A simple way of expressing this is to let the product of the activation of two units times the weight connecting them be the degree to which the constraint is satisfied. That is, for units $i$ and $j$ we let the product $w_{ij}a_ia_j$ represent the degree to which the pairwise constraint between those two hypotheses is satisfied. Note that for positive weights the more the two units are on, the better the constraint is satisfied; whereas for negative weights the more the two units are on, the less the constraint is satisfied. Second, the satistfaction of the constraint associated with the *a priori* strength or probability of the hypothesis is captured by including the activation of each unit times its bias, $a_i bias_i$, in the goodness measure. Finally, the goodness of fit for a hypothesis when direct evidence is available includes also the product of the input value times the activation value of the unit, $a_i input_i$. The bigger this product, the better the system is satisfying this external constraint.

Having identified the three types of constraint, and having defined mathematically the degree to which each is satisfied by the state of a network, we can now provide an expression for the total *goodness*, or degree of constraint satisfaction, associated with the state. This overall goodness is the function we want the network to maximize as processing takes place. This overall goodness is just the sum over all of the weights of the constraint satisfaction value for each weight, plus the sum over all external inputs and biases of the constraint satisfaction associated with each one of them:

$$G = \sum_i \sum_{j>i} w_{ij} a_i a_j + \sum_i a_i input_i + \sum_i a_i bias_i \tag{3.1}$$

Note: In constraint satisfaction systems, we imagine there is only a single (bidirectional) weight between each pair of units; $w_{ij}$ is really the same weight as $w_{ji}$. Thus, the double summation over weights in Equation 3.1 is deliberately constructed so that each unique weight is counted only once.

We have solved a particular constraint satisfaction problem when we have found a set of activation values that maximizes the function shown in the above equation. It should be noted that since we want to have the activation values of the units represent the degree to which a particular hypothesis is satisfied, we want our activation values to range between a minimum and maximum value, in which the maximum value is understood to mean that the hypothesis should be accepted and the minimum value means that it should be rejected. Intermediate values correspond to intermediate states of certainty. We have now reduced the constraint satisfaction problem to the problem of maximizing the goodness function given above. There are many methods of finding the maxima of functions. Importantly, John Hopfield (1982) noted that there is a method that is naturally and simply implemented in a class of PDP networks with symmetric wegiths. Under these conditions it is easy to see how a PDP network naturally sets activation values so as to maximize the goodness function stated above. To see this, first notice that the set of terms in the goodness function that include the activation of a given unit i correspond to the product of its current net input times its activation value. We will call this set of terms $G_i$, and write

$$G_i = net_i a_i \tag{3.2}$$

where, as usual for PDP networks, $net_i$ is defined as

$$net_i = \sum w_{ij} a_j + input_i + bias_i \tag{3.3}$$

Thus, the net input to a unit provides the unit with information as to its contribution to the goodness of the entire network state. Consider any particular unit in the network. That unit can always behave so as to increase its contribution to the overall goodness of fit if, whenever its net input is positive, the unit moves its activation toward its maximum activation value, and whenever its net input is negative, it moves its activation toward its minimum value. Moreover, since the global goodness of fit is simply the sum of the individual goodnesses, a whole network of units behaving in such a way will always increase the global goodness measure. This can be demonstated more formally by examining the partial derivative of the overall goodness with respect to the state of unit $i$. If we take this derivative, all terms in which $a_i$ is not a factor drop out, and we are simply left with the net input:

$$\partial G/\partial a_i = net_i = \sum w_{ij}a_j + input_i + bias_i \tag{3.4}$$

By definition, the partial derivative expresses how a change in $a_i$ will affect $G$. Thus, again we see that when the net input is positive, increasing $a_i$ will increase goodness, and when the net input is negative, decreasing $a_i$ will increase goodness.

It might be noted that there is a slight problem here. Consider the case in which two units are simultaneously evaluating their net inputs. Suppose that both units are off and that there is a large negative weight between them; suppose further that each unit has a small positive net input. In this case, both units may turn on, but since they are connected by a negative connection, as soon as they are both on the overall goodness may decline. In this case, the next time these units get a chance to update they will both go off and this cycle can continue. There are basically two solutions to this. The standard solution is not to allow more than one unit to update at a time. In this case, one or the other of the units will come on and prevent the other from coming on. This is the case of so-called asynchronous update. The other solution is to use a synchronous update rule but to have units increase their activation values very slowly so they can" feel" each other coming on and achieve an appropriate balance.

In practice, goodness values generally do not increase indefinitely. Since units can reach maximal or minimal values of activation, they cannot continue to increase their activation values after some point so they cannot continue to increase the overall goodness of the state. Rather, they increase it until they reach their own maximum or minimum activation values. Thereafter, each unit behaves so as to never decrease the overall goodness. In this way, the global goodness measure continues to increase until all units achieve their maximally extreme value or until their net input becomes exactly O. When this is achieved, the system will stop changing and will have found a maximum in the goodness function and therefore a solution to our constraint satisfaction problem.

When it reaches this peak in the goodness function, the goodness can no longer change and the network is said to have reached a stable state; we say it has settled or relaxed to a solution. Importantly, this solution state can be guaranteed only to be a local rather than a global maximum in the goodness function. That is, this is a hill-climbing procedure that simply ensures that the system will find a peak in the goodness function, not that it will find the highest peak. The problem of local maxima is difficult for many systems. We address it at length in a later section. Suffice it to say, that different PDP systems differ in the difficulty they have with this problem.

The development thus far applies to both of the models under discussion in this chapter. It can also be noted that if the weight matrix in an lAC network is symmetric, it too is an example of a constraint satisfaction system. Clearly, there is a close relation between constraint satisfaction systems and content-addressable memories. We turn, at this point, to a discussion of the specific models and some examples with each. We begin with the schema model of *PDP:14*.

## 3.2 THE SCHEMA MODEL

The schema model is one of the simplest of the constraint satisfaction models, but, nevertheless, it offers useful insights into the operation of all of the constraint satisfaction models. Update in the schema model is asynchronous. That is, units are chosen to be updated sequentially in random order. When chosen, the net input to the unit is computed and the activation of the unit is modified. Once a unit has been chosen for updating, the activation process in the schema model is continuous and deterministic. The connection matrix is symmetric and the units may not connect to themselves ($w_{ii} = 0$).

The logic of the hill-climbing method implies that whenever the net input ($net_i$) is positive we must increase the activation value of the unit, and when it is negative we must decrease the activation value. To keep activations bounded between 1 and 0, we use the following simple update rule:

if $net_i > 0$

$$\Delta a_i = net_i(1 - a_i)$$

otherwise,

$$\Delta a_i = net_i a_i$$

Note that in this second case, since $net_i$ is negative and $a_i$ is positive, we are decreasing the activation of the unit. This rule has two virtues: it conforms to the requirements of our goodness function and it naturally constrains the activations between 0 and 1. As usual in these models, the net input comes from three sources: a unit's neighbors, its bias, and its external inputs. These sources are added. Thus, we have

$$net_i = istr(\sum_j w_{ij} a_j + bias_i) + estr(input_i). \tag{3.5}$$

Here the constants *istr* and *estr* are parameters that allow the relative contributions of the input from external sources and that from internal sources to be readily manipulated.

## 3.3 IMPLEMENTATION

The **cs** program implementing the schema model is much like **iac** in structure. It differs in that it does asynchronous updates using a slightly different activation rule, as specified above. **cs** consists of essentially two routines: (a) an update routine called *rupdate* (for random update), which selects units at random and computes their net inputs and then their new activation values, and (b) a control routine, *cycle*, which calls *rupdate* in a loop for the specified number of cycles. Thus, in its simplest form, *cycle* is as follows:

```
function cycle
   for i = 1:ncycles
     cycleno = cycleno+1;
     rupdate();
   end
```

Thus, each time *cycle* is called, the system calls *rupdate* ncycles times, and updates the current cycle number (a second call to cycle will continue cycling where the first one left off). Note that the actual code includes checks to see if the display should be updated and/or if the process should be interrupted. We have suppressed those aspects here to focus on the key ideas.

The *rupdate* routine itself does all of the work. It randomly selects a unit, computes its net input, and assigns the new activation value to the unit. It does this *nupdates* times. Typically, nupdates is set equal to nunits, so a single call to rupdate, on average, updates each unit once:

```
function rupdate
  for updateno = 1:nupdates
    i = randint(1, nunits);
    netinput(i) = activation*weight(i,:);
    netinput = istr*(netinput+bias(i)) + estr*input(i);
    if  netinput > 0
        activation(i) = activation(i) + netinput*(l-activation(i));
    else
        activation(i) = activation(i) + netinput*activation(i);
    end
  end
```

The code shown here not only suppresses the checks for interupts and display updates; it also suppresses the fact that units are organized into pools and projections. Instead it represents in simple form the processing that would occur in a network with a single pool of units and a single matrix of connections. It is a constraint of the model, not enforced in the code, that the weight matrix must be symmetric and its diagonal elements should all be 0.

## 3.4 RUNNING THE PROGRAM

The basic structure of **cs** and the mechanics of interacting with it are identical to those of **iac**. The cs program requires a .net file specifying the particular network under consideration, and may use a .wts file to specify particular values for the weights, and a template (.tem) file that specifies what is displayed on the screen. It also allows for a .pat file for specifying a set of patterns that can be presented to the network. Once you are in MATLAB the **cs** the program can be accessed by entering 'pdp' at the matlab command prompt, or by entering the name of the pre-defined example network that has been created.

The normal sequence for running the model may involve applying external inputs to some of the units, then clicking the run button to cause the network to cycle. The system will cycle ncycles times and then stop. The value of the goodness as well as the states of activations of units can be displayed every 1 or more update or every one or more cycle, as specified in the test control panel. The step command can be used to run the network just until the next mandated display update. Once cycling stops, one can step again, or continue cycling for another ncycles if desired. While cycling, the system can be interrupted with the stop button.

### 3.4.1 Reset, Newstart, and the Random Seed

There are two ways to reinitialize the state of the network. One of these commands, *newstart*, causes the program to generate a new random seed, then reseed its random number generator, so that it will follow a new random sequence of updates. The other command, *reset*, seeds the random number generator with the same random seed that was just used, so that it will go through the very same random sequence of updates that it followed after the previous newstart or reset. The user can also specify a particular value for the random seed to use after the next reset. This can be entered by clicking *set seed* in the upper left corner of the network viewer window. In this case, when *reset* is next called, this value of the seed will be used, producing results identical to those produced on other runs begun with this same seed.

### 3.4.2 Options and parameters

The following options and parameters of the model may be specified via the options button under the Test window on the network viewer or the Set Testing options item under the Network menu in the main pdp window. They can also be set using the command *settestopts('param',value)* where *param* is the name of the parameter.

**actfunction.** Two models we will consider are available within the cs program: 'Schema', 'Boltzmann'.The user can select whether the network follows the schema model (already described) or the Boltzmann model (to be described below) via the actfunction dropdown menu.

**nupdates.** Determines the number of updates per cycle. Generally, it is set to be equal to nunits, so that each unit will be updated once per cycle, on the average.

**ncycles.** Determines the number of cycles to run when the run button is clicked or the *runprocess('test')* command is entered.

**istr.** This parameter scales the effect of the internal inputs (the bias input to each unit and the input coming from other units via the connection weights).

**estr.** Determines via a dropdown menu whether the external input is clamped or scaled. If *clamp* is selected, *estr* is ignored, and external inputs to units specify the activation value to which the unit will be set. If *scale* is selected, external inputs are treated as one contributing factor entering into a unit's net input, and are scaled by the value of *estr*, which can be entered in the numeric box to the right.

**annealsched.** This command will be described later when the concept of annealing has been introduced in the Boltzmann machine section.

**testset.** The user may choose to load one or more pattern file specifying patterns of external inputs to units. When such a file has been loaded, a checkbox called 'pat' is added to the test window. When checked, one of the patterns in the current pattern file can be selected. values will be applied as clamps or external inputs as specified by the Ext Input selector.

**Create/Edit logs** This allows the user to create logs and graphs of network variables as described in the *PDPTool User's Guide.*

## 3.5 OVERVIEW OF EXERCISES

We offer two exercises using the **cs** program. We begin with an exercise on the schema model. In Ex. 3.1, we give you the chance to explore the basic properties of this constraint satisfaction system, using the Necker cube example in *PDP:14* (originally from Feldman (1981)). The second exercise is introduced after a discussion of the problem of local maxima and of the relationship between a network's probability of being in a state and that state's goodness. In the second exercise, Ex. 3.2, we will explore these issues in a type of constraint satisfaction model called the Boltzmann machine.

### Ex3.1. The Necker Cube

Feldman (1981) has provided a clear example of a constraint satisfaction problem well-suited to a PDP implementation. That is, he has shown how a simple constraint satisfaction model can capture the fact that there are exactly two good interpretations of a Necker cube. In *PDP:14* (pp. 8-17), we describe a variant of the Feldman example relevant to this exercise. In this example we assume that we have a 16-unit network (as illustrated in Figure 3.1). Each unit in the network represents a hypothesis about the correct interpretation of a vertex of a Necker cube. For example, the unit in the lower left-hand part of the network represents the hypothesis that the lower left-hand vertex of the drawing is a front-lower-left (FLL) vertex. The upper right-hand unit of the network represents the hypothesis that the upper right-hand vertex of the Necker cube represents a front-upper-right (FUR) vertex. Note that these two interpretations are inconsistent in that we do not normally see both of those vertices as being in the frontal plane. The Necker cube has eight vertices, each of which has two possible interpretations-one corresponding to each of the two interpretations of the cube. Thus, we have a total of 16 units.

Three kinds of constraints are represented in the network. First, units that represent consistent interpretations of neighboring vertices should be mutually exciting. These constraints are all represented by positive connections with a weight of 1. Second, since each vertex can have only one interpretation, we have a negative connection between units representing alternative interpretations of the same input vertex. Also, since two different vertexes in the input can't both be the same corner in the percept (e.g. there cannot be two front-lower-left corners when a single cube is perceived) the units representing the same corner of the cube in each of the interpretations are mutually inhibitory. These inhibitory connections all have weights of -1.5. Finally, we assume that the system is, essentially, viewing the ambiguous figure, so that each vertex gets some bottom up excitation. This is actually implemented through a positive bias equal to .5, coming to each unit in the network. The above values are all scaled by the *istr* parameter, which is set initially at .4 for this network.

After setting the cs directory and the current directory, you can start up the **cs** program on the cube example by simply typing *cube* at the command prompt. At this point the screen should look like the one shown in Figure 3.2. The display depicts the two interpretations of the cube and shows the activation values of the units, the current cycle number, the current update number, the name of the most recently updated unit (there is none yet so this is blank), the current value of goodness, and the current temperature. (The temperature is irrelevant for this exercise, but will become important later.) The activation values of all 16 units are shown, initialized to 0, at the corners of the two cubes drawn on the screen. The units in the cube on the left, cube A, are the ones consistent with the interpretation that the cube is facing down and to the left. Those in the cube on the right, cube B, are the ones consistent with the interpretation of the cube as facing up and to the right. The dashed lines do not correspond to the connections among the units, but simply indicate the interpretations of the units. The connections are those shown in the Necker cube network in Figure 1. The vertices are labeled, and the labels on the screen correspond to those in Figure 3.1. All units have names. Their names are given by a capital letter indicating which interpretation is involved (A or B), followed by the label appropriate to the associated vertex. Thus, the unit displayed at the lower left vertex of cube A is named Afll, the one directly above it is named

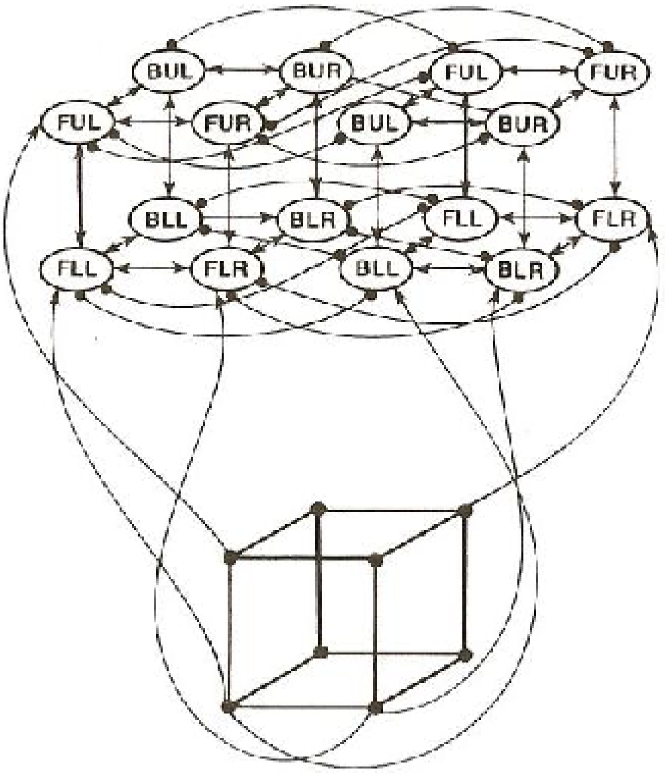
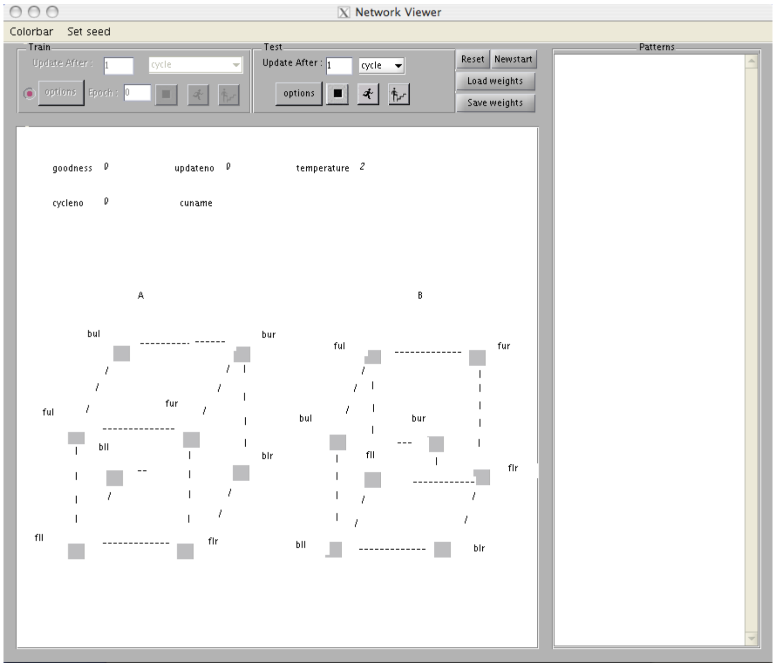


Figure 3.1: A simple network representing some of the constraints involved in perceiving the Necker cube (From *PDP:14*). A valid interprtation of the cube would involve all of the units in on side of the network turned on and all of the units in the other side turned off.

Aful (for the front-upper-left vertex of cube A), and so on.

We are now ready to begin exploring the cube example. The biases and connections among the units have already been read into the program. In this example, as stated above, all units have positive biases, therefore there is no need to specify inputs. Simply click run. After the command is typed, the display will be updated once per cycle (that is, after every 16 unit updates). After the display stops flashing you should see the desplay shown in Figure 3.3. The variable *cycle* should be 20, indicating that the program has completed 20 cycles. The variable *update* should be at 16, indicating that we have completed the 16th update of the cycle. The *uname* will indicate the last unit updated. The goodness should have a value of 6.4. This value corresponds to a global maximum; and indeed, if you inspect the activations, you will see that the units on the right have reached the maximal value of one, and the units on the left are



Update After : 1 cycle

Update After : 1 cycle

goodness 0 updateno 0 temperature 2

cycleno 0 cuname

Figure 3.2: Initial screen appearance for Cube example.

all 0, corresponding to one of the two "standard" interpretations of the cube.

Q.3.1.1.

> Using 3.1, explain quantitatively how the exact value of goodness comes to be 6.4 when the network has reached the state shown in the display. Remember that all weights and biases are scaled by the *istr* parameter, which is set to .4. Thus each excitatory weight can be treated as having value .4, each inhibitory weight -.6, and the positive bias as having value .2.

You can run the cube example again by issuing the *newstart* command and then hitting the run button. Do this until you find a case where, after 20 cycles, there are four units on in cube A and four on in cube B. The goodness will be 4.8.

Q.3.1.2.

> Using 3.1, explain why the state you have found in this case corresponds to a goodness value of 4.8.

Continue again until you find a case where, after 20 cycles, there are two units on in one of the two cubes and six units on in the other.

Q.3.1.3.

> Using Equation 3.1, explain why the state you have found in this case also corresponds to a goodness value of 4.8.

Now run about 20 more cases of newstart followed by run, and record for each the number of units on in each subnetwork after 20 cycles, making a simple tally of cases in which the result was [8 0] (all eight units in the left cube activated, none in the right), [6 2], [4 4], [2 6], and [0 8]. Examine the states where there are units on in both subnetworks.

To facilitate this process, we have provided a little function called *onecube(n)* that you can execute from the command line. This function issues one *newstart* and then runs $n$ cycles, showing the final state only. To enter the command again, you can use *ctrl-p*, followed by enter. You can change the value of $n$ by editing the command before you hit enter. For present purposes, you should simply leave $n$ set at 20. Standalone users must follow the directions in this footnote.[1]

Q.3.1.4.

> How many times was each of the two valid interpretations found? How many times did the system settle into a local maximum? What were the local maxima the system found? To what extent do they correspond to reasonable interpretations of the cube?

Now that you have a feeling for the range of final states that the system can reach, try to see if you can understand the course of processing leading up to the final state.

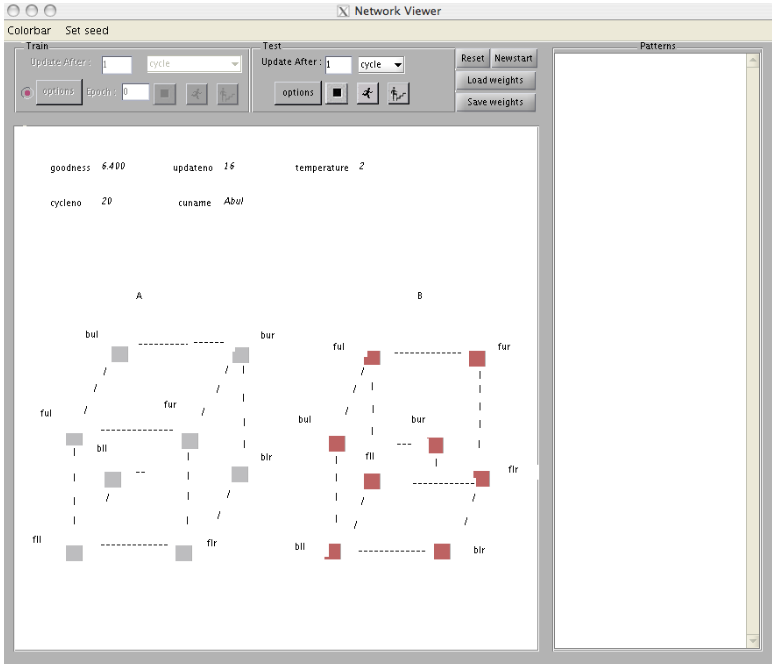
Q.3.1.5.

> What causes the system to reach one interpretation or the other? How early in the processing cycle does the eventual interpretation become clear? What happens when the system reaches a local maximum? Is there a characteristic of the early stages of processing that leads the system to move toward a local maximum?

*Hint.*

> Note that if you wish to study how the network evolved to a particular solution you obtained at the end of 20 cycles following a newstart, you can use *reset* to prepare the network to run through the very same sequence of unit updates again. If at that point you set *Update after* to 1 update, you can then then follow the steps to the solution

[1] Standalone users should use the command *'runscript onecbscript.m'* to achieve the same effect as calling the *onecube* function. The command can be re-executed using the up-arrow key followed by return. To change the number of cycles, you will need to edit the onecbscript.m file using your preferred text editor such as wordpad or emacs.



goodness 6.400 updateno 16 temperature 2

cycleno 20 cuname Abul

Figure 3.3: The state of the system 20 cycles after the network was initialized at startup.

update-by-update by clicking repeatedly on the step button. So, you can repeatedly issue the *onecube(20)* command until you find a case you'd like to study, click *reset*, set *Update after* as described, and step your way along to watch the state evolve.

There is a parameter in the schema model, *istr*, that multiplies the weights and biases and that, in effect, determines the rate of activation flow within the model. The probability of finding a local maximum depends on the value of this parameter. You can set this variable through the value of *istr* under the *options* popup under the *test* panel. Another way to set this is by using the command *settestopts('istr',value)* where *value* a positive number. Try several values from 0.1 to 2.0, running *onecube* 20 times for each.

Q.3.1.6.

How does the distribution of final states vary as the value of *istr* is varied? Report your results in table form for each value of *istr*, showing the number of times the network settles to [8 0], [6 2], [4 4], [2 6], and [0 8] in each case. Consider the distribution of different types of local maxima for different values of *istr* carefully. Do your best explain the results you obtain.

*Hint.*

> At low values of *istr* you will want to increase the value of the ncycles argument to *onecube*; 80 works well for values like 0.1. Do not be disturbed by the fact that the values of goodness are different here than in the previous runs. Since *istr* effectively scales the weights and biases, it also multiplies the goodness so that goodness is proportional to *istr*.

*Biasing the necker cube toward one of the two possible solutions.* Although we do not provide an excise associated with this, we note that it is possible to use external inputs to bias the network in favor of one of the two interpretations. Study the effects of adding an input of 1.0 to some of the units in one of the subnetworks, using a command like the following at the command prompt:

```
net.pool(2).extinput = [1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0];
```

The first eight units are for interpretation A, so this should provide external input (scaled by *estr*, which is set to the same .4 value as *istr*) to all eight of the units corresponding to that interpreation. If you wish, you can explore how the distribution of interpretations changes as a result of varying the number of units receiving external input of 1. You can also vary the strength of the external inputs, either by supplying values differet from 1 in the external input vector, or by adjusting the value of the *estr* parameter.

### Schemata for Rooms

One other pre-defined network is available using the schema model. This is a network described in *PDP:14* that is intended to illustrate various features of a PDP implementation of the idea of a 'schema'. Consult *PDP:14* for a the background and for the details of this model.

The network is called the *room* network, and can be started (after closing other networks) by typing *room* to the Matlab command prompt while in the cs directory. The network can be used in either of two ways. The first involves setting external input values for one or a few of the units in the network, and then allowing the network to settle to a stable state (weights are symmetric so a fixed point will eventually be reached). To use the program in this way, simply provide external inputs by right clicking of the box to the *left* of the corresponding unit name, typing in a value, and hitting enter (the box to the *right* of the unit name represents the unit's activation). These values are initialized to 0 at startup. You can actually run the program with no external inputs, and it will settle to a state that seems to be something like a bedroom, with a television and some bookshelves in it, and a fireplace. If you provide input to 'stove', the network will settle to a very different state. You can play with different combinations, hitting reset then run after setting the desired extenal inputs. Remember to re-enter 0 values for external inputs you do not want to keep active.

The second way of using the *room* network involves clamping entire patterns corresponding to particular states onto the units. In this mode there is no real settling. Actiations are fixed to specific values (1 or 0) by the values specified in the patterns, which are contained in a file named *ROOM.PAT*. In this mode, while multiple cyles can occur, nothing happens to the activations because they are clamped, but the weights and activations are used to calculate goodness values for the specified states. These patterns are shown in the network view window after the program is launched. To use the program in this mode, click the *Pat* checkbox in the Test control panel. Click *Reset* to set activations back to 0 (although the display does not update after reset is clicked in this case). Then click on a pattern that you wish to present, then click the *step* icon in the Test control panel. You can explore other patterns most easily by adding your own patterns to the *ROOM.PAT* file or to a file of another name that you then load yourself – this file can be loaded through the *Load new* button within the testing options subpanel, then selecting your new file, which should be in the cs directory and have the *.pat* extension. Note that in the pattern file, a value of 1 is used to clamp a unit on and -1 is used to clamp the unit off.

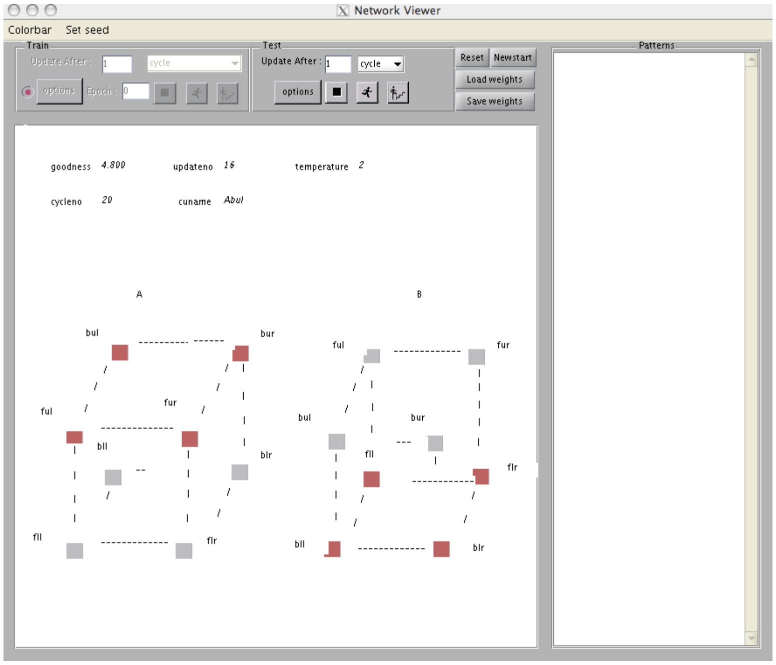
## 3.6 GOODNESS AND PROBABILITY

In this section we introduce Hinton and Sejnowski's Boltzmann machine, described in *PDP:7*. This model was developed from an analogy with statistical physics and it is useful to put it in this context. We thus begin with a description of the physical analogy and then show how this analogy solves some of the problems of the schema model described above. Then we turn to a description of the Boltzmann machine, show how it is implemented, and allow you to explore how the cs program can be used in boltzmann mode to address constraint satisfaction problems.

The initial focus of this material when first written in the 1980's was on the use of the Boltzmann machine to solve the problem of local maxima. However, an alternative, and perhaps more important, perspective on the Boltzmann machine is that is also provides an important step toward a theory of the relationship between the content of neural networks (i.e. the knowledge in the weights) and the probability that they settle to particular outcomes. We have already taken a first step in the direction of such a theory, by seeing that networks tend to seek maxima in Goodness, which is in turn related to the knowledge in the weights. In this section we introduce an elegant analysis of the quantitative form of the relationship between the probability that a network will find itself in a particular state, and that state's Goodness. Because of the link between goodness and constraints encoded in the weights this analysis allows a deeper understanding of the relationship between the constraints and the outcome of processing.

### 3.6.1 Local Maxima

As stated above, one advantage of the Boltzmann machine over the deterministic constraint satisfaction system used in the schema model is its ability to overcome



Update After : 1 cycle

goodness 4.800
updateno 16
temperature 2
cycleno 20
cuname Abul

Figure 3.4: A local maximum found in the schema model.

the problem of local maxima in the goodness function. To understand how this is done, it will be useful to begin with an example of a local maximum and try to understand in some detail why it occurs and what can be done about it. Figure 3.6.1 illustrates a typical example of a local maximum with the Necker cube. Here we see that the system has settled to a state in which the upper four vertices were organized according to interpretation A and the lower four vertices were organized according to interpretation B. Local maxima are always blends of parts of the two global maxima. We never see a final state in which the points are scattered randomly across the two interpretations.

All of the local maxima are cases in which one small cluster of adjacent vertices are organized in one way and the rest are organized in another. This is because the constraints are local. That is, a given vertex supports and receives support from its neighbors. The units in the cluster mutually support one another. Moreover, the two clusters are always arranged so that none of the inhibitory connections are active. Note in this case, Afur is on and the two units it inhibits, Bfur and Bbur, are both off. Similarly, Abur is on and Bbur and Bfur are both off. Clearly the system has found little coalitions of units that hang together and conflict minimally with the other coalitions. In Ex. 3.1, we had the opportunity to explore the process of settling into one of these local maxima. What happens is this. First a unit in one subnetwork comes on. Then a unit in the other subnetwork, which does not interact directly with the first, is updated, and, since it has a positive bias and at that time no conflicting inputs, it also comes on. Now the next unit to come on may be a unit that supports either of the two units already on or possibly another unit that doesn't interact directly with either of the other two units. As more units come on, they will fit into one or another of these two emerging coalitions. Units that are directly inconsistent with active units will not come on or will come on weakly and then probably be turned off again. In short, local maxima occur when units that don't interact directly set up coalitions in both of the subnetworks; by the time interaction does occur, it is too late, and the coalitions are set.

Interestingly, the coalitions that get set up in the Necker cube are analogous to the bonding of atoms in a crystalline structure. In a crystal the atoms interact in much the same way as the vertices of our cube. If a particular atom is oriented in a particular way, it will tend to influence the orientation of nearby atoms so that they fit together optimally. This happens over the entire crystal so that some atoms, in one part of the crystal, can form a structure in one orientation while atoms in another part of the crystal can form a structure in another orientation. The points where these opposing orientations meet constitute flaws in the crystal. It turns out that there is a strong mathematical similarity between our network models and these kinds of processes in physics. Indeed, the work of Hopfield (1982, 1984) on so-called Hopfield nets, of Hinton and Sejnowski (1983), *PDP:7*, on the Boltzmann machine, and of Smolensky (1983), *PDP:6*, on harmony theory were strongly inspired by just these kinds of processes. In physics, the analogs of the goodness maxima of the above discussion are energy minima. There is a tendency for all physical systems to evolve from highly energetic states to states of minimal energy. In 1982, Hopfield, (who is a physicist), observed that symmetric networks using deterministic update rules behave in such a way as to minimize an overall measure he called energy defined over the whole network. Hopfield's energy measure was essentially the negative of our goodness measure. We use the term goodness because we think of our system as a system for maximizing the goodness of fit of the system to a set of constraints. Hopfield, however, thought in terms of energy, because his networks behaved very much as thermodynamical systems, which seek minimum energy states. In physics the stable minimum energy states are called attractor states. This analogy of networks falling into energy minima just as physical systems do has provided an important conceptual tool for analyzing parallel distributed processing mechanisms.

Hopfield's original networks had a problem with local "energy minima" that was much worse than in the schema model described earlier. His units were binary. (Hopfield (1984) subsequently proposed a version in which units take on a continuum of values to help deal with the problem of local minima in his original model. The schema model is similar to Hopfield's 1984 model, and with small values of *istr* we have seen that it is less likely to settle to a local minimum). For binary units, if the net input to a unit is positive, the unit takes on its maximum value; if it is negative, the unit takes on its minimum value (otherwise, it doesn't change value). Binary units are more prone to local minima because the units do not get an opportunity to communicate with one another before committing to one value or the other. In Ex. 3.1, we gave you the opportunity to run a version close to the binary Hopfield model by setting *istr* to 2.0 in the Necker cube example. In this case the units are always at either their maximum or minimum values. Under these conditions, the system reaches local goodness maxima (energy minima in Hopfield's terminology) much more frequently.

Once the problem has been cast as an energy minimization problem and the analogy with crystals has been noted, the solution to the problem of local goodness maxima can be solved in essentially the same way that flaws are dealt with in crystal formation. One standard method involves *annealing*. Annealing is a process whereby a material is heated and then cooled very slowly. The idea is that as the material is heated, the bonds among the atoms weaken and the atoms are free to reorient relatively freely. They are in a state of high energy. As the material is cooled, the bonds begin to strengthen, and as the cooling continues, the bonds eventually become sufficiently strong that the material freezes. If we want to minimize the occurrence of flaws in the material, we must cool slowly enough so that the effects of one particular coalition of atoms has time to propagate from neighbor to neighbor throughout the whole material before the material freezes. The cooling must be especially slow as the freezing temperature is approached. During this period the bonds are quite strong so that the clusters will hold together, but they are not so strong that atoms in one cluster might not change state so as to line up with those in an adjacent cluster even if it means moving into a momentarily more energetic state. In this way annealing can move a material toward a global energy minimum.

The solution then is to add an annealing-like process to our network models and have them employ a kind of simulated annealing. The basic idea is to add a global parameter analogous to temperature in physical systems and therefore called temperature. This parameter should act in such a way as to decrease the strength of connections at the start and then change so as to strengthen them as the network is settling. Moreover, the system should exhibit some random behavior so that instead of always moving uphill in goodness space, when the temperature is high it will sometimes move downhill. This will allow the system to "step down from" goodness peaks that are not very high and explore other parts of the goodness space to find the global peak. This is just what Hinton and Sejnowski have proposed in the Boltzmann machine, what Geman and Geman (1984) have proposed in the Gibbs sampler, and what Smolensky has proposed in harmony theory. The essential update rule employed in all of these models is probabilistic and is given by what we call the logistic function:

$$p(a_i = 1) = \frac{e^{net_i/T}}{e^{net_i/T} + 1} \tag{3.6}$$

where $T$ is the temperature. Dividing the numerator and denominator by $e^{net_i/T}$ gives the following version of this function, which is the one must typically used:

$$p(a_i = 1) = \frac{1}{1 + e^{-net_i/T}} \tag{3.7}$$

This differs from the basic schema model in several important ways. First, like Hopfield's original model, the units are binary. They can only take on values of 0 and 1. Second, they are *stochastic* – that is, their value is subject to uncertainty. The update rule specifies only a probability that the units will take on one or the other of their values. This means that the system need not necessarily go uphill in goodness-it can move downhill as well. Third, the behavior of the systems depends on a global parameter, $T$, which determines the relative likelihood of different states in the network.

In fact, in networks of this type, a very important relationship holds betwen the *equilibrium probability* of a state and the state's goodness:

$$p(S_i) = \frac{e^{G_i/T}}{\sum_{i'} e^{G_{i'}/T}} \tag{3.8}$$

The denominator of this expression is a sum over all possible states, and is often difficult to compute, but we now can see the likehood ratio of being in either of two states $S_1$ or $S_2$ is given by

$$\frac{p(S_1)}{p(S_2)} = \frac{e^{G_1/T}}{e^{G_2/T}}, \tag{3.9}$$

or alternatively,

$$\frac{p(S_1)}{p(S_2)} = e^{(G_1-G_2)/T}. \tag{3.10}$$

A final way of looking at this relationship that is sometimes useful comes if we take the log of both sides of this expression:

$$log(\frac{p(S_1)}{p(S_2)}) = (G_1 - G_2)/T. \tag{3.11}$$

At equilibrium, the log odds of the two states is equal to the difference in goodness, divided by the temperature.

These simple expressions above can serve two important purposes for neural network theory. First, they allow us to predict what a network will do from knowledge of the constraints encoded in its weights, biases, and inputs, when the network is run at a fixed temperature. This allows a mathematical derivation of aspects a network's behavior and it allows us to relate the network's behavior to theories of optimal inference.

Second, these expressions allow us to prove that we can, in fact, find a way to have networks settle to one of their global maxima. In essence, the reason for this is that, as $T$ grows small, the probility ratios of particular pairs of states become more and more extreme. Consider two states with Goodness 5 and goodness 4. When T is 1, the ratio of the probabilities is $e$, or 2.73:1. But when the temperature is .1, the ratio $e^{10}$, or 22,026:1. In general, as temperature goes down we can make the ratio of the probabilities of two states as large as we like, even if the goodness difference between their probabilities is small.

However, there is one caveat. This is that the above is true, only *at equilibrium*, and provided that the system is *ergodic*.

The equilibrium probability of a state is a slightly tricky concept. It is best understood by thinking of a very large number of copies of the very same network, each evolving randomly over time (with a different random seed, so each one is different). Then we can ask: What fraction of these networks are in any given state at any given time? They might all start out in the same state, and they may all tend to evolve toward better states, but at some point the tendency to move into a good state is balanced by the tendency to move out of it again, and at this point we say that the probability distribution has reached equilibrium. At moderate temperatures, the flow between states occurs readily, and the networks tend to follow the equilibrium distribution at they jump around from state to state. At low temperatures, however, jumping between states becomes very unlikely, and so the networks may be more likely to be found in local maxima than in states that are actually better but are also neighbors of even better states. When the flow is possible, we say that the system is *ergodic*. When the system is ergotic, the equilibrium is independent of the starting state. When the flow is not completely open, it is not possible to get from some states to some other states.

In practice "ergodicity" is a matter of degree. If nearby relatively bad states have very low probability of being entered from nearly higher states, it can simply take more time that it seems practical to wait for much in the way of flow to occur. This is where simulated annealing comes in. We can start with the temperature high, allowing a lot of flow out of relatively good states and gradually lower the temperature to some desired level. At this temperature, the distribution of states can in some cases approximate the equilibrium distribution, even though there is not really much ongoing movement between different states.

## Ex3.2. Exploring Equilibria in the Cube Network

To get a sense of all of this, we will run a few more exercises. Each exercise will involve running an ensemble of networks (100 of them) and then looking at the distribution across states at the end of some number of cycles. Luckily, we are using a simple network, so running an ensemble of them goes quickly.

To start this exercise, exit from the cube example through the pdp window (so that all your windows close). To make sure everything is cleared up, you can type

```
close all; clear all;
```

at the command prompt. Then start up again using the command *boltzcube* to the command prompt. What is different about this compared to *cube* is that the model is initiated in Boltzmann mode, with an annealing schedule specified, and both *istr* and *estr* are set to 1 so that the goodness values are more transparent. The two global optima are associated with goodness of 16. This consists of 12 times 1 for the weights along the edges of the interpretation in which all of the units are active, plus 8 times .5 for the bias inputs to the eight units representing the corners of that cube. The local maxima we have considered all have goodness of 12.

The annealing schedule can be set through the options popup (annealsched pushbutton) but is in fact easier to specify at the command prompt via the settestopts command. Throughout this exercise, we will work with a final temperature of 0.5. The initial annealing schedule is set in the script by the command:

```
settestopts('annealsched',[0 2;20 .5]);
```

This tells the network to initialize the temperature at 2, then linearly reduce it over 20 cycles to .5. In general the schedule consists of a set of *time value* pairs, separated by a semicolon. Times must increase, and values must be greater than 0 (use .001 if you want effectively 0 temperature).

Each time you run an ensemble of networks, you will do so using the *manycubes* command. This command takes two arguments. The first is the number of instances of the network settling process to run and the second is the number of cycles to use in running each instance. Enter the command shown below now at the command prompt with arguments 100 and 20, to run 100 instances of the network each for 20 cycles. Standalone users must follow the directions in this footnote.[2]

```
histvals = manycubes(100,20)
```

If you have the Network Viewer window up on your screen, you'll see the initial and final states for each instance of the settling process flash by one by one. At the end of 100 instances, a bargraph will pop up, showing the distribution of goodness values of the states reached after 20 cycles. The actual numbers corresponding to the bars on the graph are stored in the *histvals* variable; there are 33 values corresponding to goodnesses from 0 to 16 in steps of 0.5. Enter *histvals(17:33)* to display the entries corresponding to goodness values from 8 to 16. In one run of the manycubes(100,20), the results came out like this:

- 62 states with goodness 16
- 14 states with goodness 13.5
- 2 states with goodness 12.5
- 10 states with goodness 12

We want to know whether we are getting about the right equilibrium distribution of states, given that our final temperature is .5. We can calculate the

[2] Standalone users should use the command *runscript manycbscript.m*. The variable histvals will be created and its contents can be dumped to the command window by simply typing the variable name 'histvals', followed by enter, at the command prompt. The parameters of the command, called ncubes and mccycles, are set at the top of the manycbscript.m file, and can be changed by editing their values using your preferred text editor.

ratios of probabilities of being in particular states, but we need to take into account that in fact there are several states with each of the four goodness values mentioned above. The probability of having a particular goodness is equal to the probability of being in a particular state with that goodness times the number of such states.

Q.3.2.1.

> How many different states of the network have goodness values of 16? Here, it is immediately obvious that there are two such states. How many local maxima are there, that have goodness values of 12? Consider, also, the state in which all but one unit is on in one cube and no units are on in the other. What is the goodness value of that state? How many such states are there? Finally, consider the states in which all units are on in one cube, and one is on in the other. What is the goodness value of such a state? How many such states are there? Although the answers are given below you should try to work this out yourself first, then, in your written answer, explain how you get these numbers.

Since we need to have the right numbers to proceed, we give the answers to the question above:

- 2 states with goodness 16.
- 16 states with goodness 13.5, which are near misses with an extra unit on in the other cube.
- 16 states with goodness 12.5, which are near misses with a unit off in the active cube.
- 12 states with goodness 12 (4 [6 2] states, 4 [4 4] states, and 4 [2 6] states).

If your answer differs with regard to the number of goodness 12 states, here is an explanation. Competition occurs between pairs of units in the two cubes corresponding to the two alternative interpretations of each of the four front-to-back edges. Either pair of units can be on without direct mutual inhibitory conflict. Other edges of the cubes do not have this property. There are thus four ways to have an edge missing from one cube, and four ways to have an edge missing from the other cube. Thus there are four 6-2 and four 2-6 maxima. Similarly, the top surface of cube A can coexist with the bottom surface of cube B (or vise versa) without mutual inhibition, and the left surface of cube A can coexist with the right surface of cube B (or vice versa) without mutual inhibition, giving four possible 4-4 local maxima. Note that the four units corresponding to the front or back surface of cube A cannot coexist with the four units corresponding to either the front or the back surface of cube B due to the mutual inhibition.

So now we can finally ask, what is the relative probability of being in a state with one of these four goodnesses, given the final temperature achieved in the network?

We can calculate this as follows. For each Goodness Value ($GV$), we have:

$$p(GV) = N_{GV}\frac{e^{(GV/T)}}{Z} \tag{3.12}$$

Here $N_{GV}$ represents the number of different states having the goodness value in question, and $e^{(GV/T)}$ is proportional to the probability of being in any one of these states. We use $Z$ to represent the denominator, which we will not need to calculate. Consider first the value of this expression for the highest goodness value, $GV = 16$, corresponding to the global maxima. There are two such maxima, so $N_{GV} = 2$. So to calculate the numerator of this expression (disregarding Z) for our temperature $T = .5$, we enter the following at or MATLAB command prompt:

```
2*exp(16/.5)
```

We see this is a very large number. To see it in somewhat more compact format enter *format short g* at the MATLAB prompt then enter the above expression again. The number is 1.58 times 10 to the 14th power. Things come back into perspective when we look at the ratio of the probability of being in a state with goodness 16 to the probability of being in a state with goodness 13.5. There are 16 such states, so the ratio is given by:

```
(2*exp(16/.5))/(16*exp(13.5/.5))
```

The ratio is manageable: it is 18.6 or so. Thus we should have a little more than 18 times as many states of goodness 16 as we have states of goodness 13.5. In fact we are quite far off from this in my run; we have 62 cases of states of goodness 16, and 14 cases of states of goodness 13.5, so the ratio is 4.4 and is too low.

Let's now do the same thing for the ratio of states of goodness 12 to states of goodness 16. There are 12 states of goodness 12, so the ratio is entered as

```
(2*exp(16/.5))/(12*exp(12/.5))
```

Calculating, we find this ratio is 496.8. Since I observed 10 states of goodness 12, and 62 of goodness 16, the observed ratio is *hugely* off: 62/10 is only 6.2.

Looking at the probability ratio for states of goodness 12.5 vs. 12, we have:

```
(16*exp(12.5/.5))/(12*exp(12/.5))
```

The ratio is 3.6. Thus, at equilibrium the network should be in a 12.5 goodness state more often than a 12 goodness state. However, we have the opposite pattern, with 10 instances of goodness 12 and 2 of goodness 12.5. Clearly, we have not achieved an approximation of the equilibrium distribution; it appears that many instances of the network are stuck in one of the local maxima, i.e. the states with goodness of 12.

The approximate expected counts of 100 samples at equilibrium should be:

| Goodness | Expected Count |
|---|---|
| 16 | 93.1 |
| 13.5 | 5.0 |
| 12.5 | 0.7 |
| 12 | 0.2 |
| All others | 1.0 |

Q.3.2.2.

> Try to find an annealing schedule that runs over 100 cycles with a final temperature of 0.5 that ends up as close as possible to the right final distribution over the states with goodness values of 16, 13.5, 12.5, and 12. (Some lower goodness values may also appear.) In particular, try to find a schedule that produces 1 or fewer states with a goodness value of 12. You'll need to do several runs of 100 cycles, 100 instances each using the manycubes command. Report the results of your best annealing schedule and three other schedules in a table, showing the annealing schedules used as well as the number of states per goodness value (between 8 and 16) at the end of each run. For your best schedule, report the results of two runs, since there is variability from run to run. Explain the adjustments you make to the annealing schedule and the thoughts that led you to make them.

To make your life easy you can use a command like

```
histvals = manycubes(100,100); histvals(17:33)
```

with no semi-colon at the end to show the entries in histvals corresponding to goodnesses ranging from 8 (histvals(17)) to 16 (histvals(33)). The bargraph will make it easy to get a sense is what is going on while the screen dump of the numbers gives you the actual values.

If the line before the call to manycubes sets the annealing schedule (see example below), then the main MATLAB window will contain a record of all of your commands and all of your data, and you can copy and paste individual lines into your homework paper; of course, you will want to edit and reformat for readability. Remember you can use the arrow keys to access previous commands and that you can edit them before hitting enter.

In adjusting the annealing schedule, you will want to consider whether the intial temperature is high enough. Thus, if you want to use a higher start value and an intermediate milestone, you might have the following commands (note that the total number of cycles is specified in both the annealing schedule and manycubes commands):

```
settestopts('annealsched',[0 5; 50 2; 100 .5]);
histvals = manycubes(100,100); histvals(17:33)
```

*Hint.*

> A higher initial temperature with two intermediate milestones produces results that come close to matching the correct equilibrium distribution. It is hard to get a perfect match – see what you come up with.

The next question harkens back to the discussion of the physical process of annealing in the discussion of the physics analogy:

Q.3.2.3.

> Discuss the likelihood of a network escaping from a [4 4] local maximum to another local maximum at $T = 1$: Provide an example of 2 sequences of evens by which this might occur. Picking one such sequence, calculate the probability that such a series of states would occur. Once such a maximum is escaped to a global maximum, what sequence of events would have to happen to get back to this [4 4] local maximum? Also consider excape and return to [6 2] and [2 6] type local maxima. We don't expect exact answers here, but the question will hopefully elicit reasonable intuitions. With these thoughts in hand, discuss your understanding of why your best schedule reported above works as well or poorly as it does, and how you might improve on it further.

### 3.6.2 Escaping from Local Maxima

The title of this section is a bit of a play on words. We have been focusing on local maxima, and how to escape from them, in a way that may distract from the deeper contribution of the relationship between goodness and probability. For example, it may be that local maxima are not as much of a problem in PDP systems as the cube example makes it seem like they might be. The constraints used in this example were deliberately chosen to create a network that would have such maxima, so that it would help illustrate several useful concepts. But are local maxima inevitable?

Q.3.2.4.

> Consider how you might change the network used in the cube example to avoid the problem of local maxima. Assume you still have the same sixteen vertex units, and the same bias inputs making each of the two interpretations equally likely. Briefly explain (with an optional drawing) one example in which adding connections would help and one example in which adding hidden units would help.

Your answer may help to illustrate both that local maxima are not necessarily inevitable and that hidden units (units representing important clusters of inputs that tend to occur together in experience) may play a role in solving the local maximum problem. More generally, the point here is to suggest that the relationship between constraints, goodness, and probability may be a useful one even beyond avoiding the problem of getting stuck in local maxima.

# Chapter 4

# Learning in PDP Models: The Pattern Associator

In previous chapters we have seen how PDP models can be used as content-addressable memories and constraint-satisfaction mechanisms. PDP models are also of interest because of their learning capabilities. They learn, naturally and incrementally, in the course of processing. In this chapter, we will begin to explore learning in PDP models. We will consider two "classical" procedures for learning: the so-called Hebbian, or correlational learning rule, described by Hebb (1949) and before him by William James (1950), and the error-correcting or "delta" learning rule, as studied in slightly different forms by Widrow and Hoff (1960) and by Rosenblatt (1959).

We will also explore the characteristics of one of the most basic network architectures that has been widely used in distributed memory modeling with the Hebb rule and the delta rule. This is the pattern associator. The pattern associator has a set of input units connected to a set of output units by a single layer of modifiable connections that are suitable for training with the Hebb rule and the delta rule. Models of this type have been extensively studied by James Anderson (see Anderson, 1983), Kohonen (1977), and many others; a number of the papers in the Hinton and Anderson (1981) volume describe models of this type. The models of past-tense learning and of case-role assignment in *PDP:18* and *PDP:19* are pattern associators trained with the delta rule. An analysis of the delta rule in pattern associator models is described in *PDP:11.*

As these works point out, one-layer pattern associators have several suggestive properties that have made them attractive as models of learning and memory. They can learn to act as content-addressable memories; they generalize the responses they make to novel inputs that are similar to the inputs that they have been trained on; they learn to extract the prototype of a set of repeated experiences in ways that are very similar to the concept learning characteristics seen in human cognitive processes; and they degrade gracefully with damage and noise. In this chapter our aim is to help you develop a basic understanding of the characteristics of these simple parallel networks. However, it must be noted that these kinds of networks have limitations. In the next chapter we will examine these limitations and consider learning procedures that allow the same positive characteristics of pattern associators to manifest themselves in networks and overcome one important class of limitations.

We begin this chapter by presenting a basic description of the learning rules and how they work in training connections coming into a single unit. We will then apply them to learning in the pattern associator.

## 4.1 BACKGROUND

### 4.1.1 The Hebb Rule

In Hebb's own formulation, this learning rule was described eloquently but only in words. He proposed that when one neuron participates in firing another, the strength of the connection from the first to the second should be increased. This has often been simplified to 'cells that fire together wire together', and this in turn has often been represented mathematically as:

$$\Delta w_{ij} = \epsilon a_i a_j \tag{4.1}$$

Here we use $\epsilon$ to refer to the value of the learning rate parameter. This version has been used extensively in the early work of James Anderson (e.g., Anderson, 1977). If we start from all-zero weights, then expose the network to a sequence of learning events indexed by $l$, the value of any weight at the end of a series of learning events will be

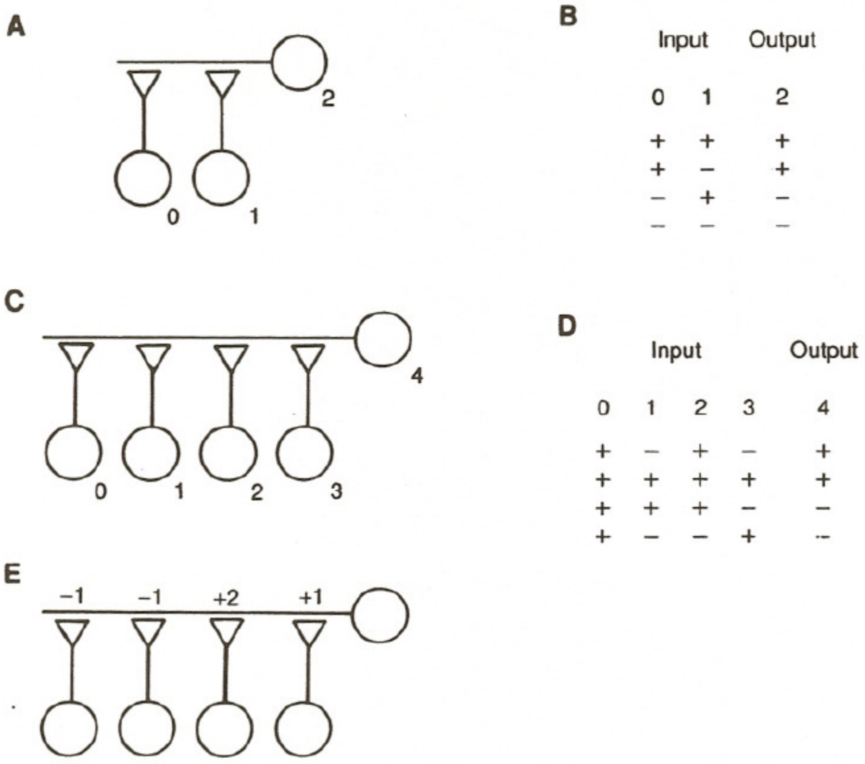
$$w_{ij} = \epsilon \sum_l a_{il} a_{jl} \tag{4.2}$$

In studying this rule, we will assume that activations are distributed around 0 and that the units in the network have activations that can be set in either of two ways: They may be clamped to particular values by external inputs or they may be determined by inputs via their connections to other units in the network. In the latter case, we will initially focus on the case where the units are completely linear; that is, on the case in which the activation and the output of the unit are simply set equal to the net input:

$$a_i = \sum_j a_j w_{ij} \tag{4.3}$$

In this formulation, with the activations distributed around 0, the $w_{ij}$ assigned by Equation 4.2 will be proportional to the correlation between the activations of units $i$ and $j$; normalizations can be used to preserve this correlational property when units have mean activations that vary from 0.

The correlational character of the Hebbian learning rule is at once the strength of the procedure and its weakness. It is a strength because these



| Input | | Output |
|---|---|---|
| 0 | 1 | 2 |
| + | + | + |
| + | − | + |
| − | + | − |
| − | − | − |

| Input | | | | Output |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| + | − | + | − | + |
| + | + | + | + | + |
| + | + | + | − | − |
| + | − | − | + | − |

Figure 4.1: Two simple associative networks and the patterns used in training them.

correlations can sometimes produce useful associative learning; that is, particular units, when active, will tend to excite other units whose activations have been correlated with them in the past. It can be a weakness, though, since correlations between unit activations often are not sufficient to allow a network to learn even very simple associations between patterns of activation.

First let's examine a positive case: a simple network consisting of two input units and one output unit (Figure 4.1A). Suppose that we arrange things so that by means of inputs external to this network we are able to impose patterns of activation on these units, and suppose that we use the Hebb rule (Equation 4.1 above) to train the connections from the two input units to the output unit. Suppose further that we use the four patterns shown in Figure 4.1B; that is, we present each pattern, forcing the units to the correct activation, then we adjust the strengths of the connections between the units. According to Equation 4.1, $w_{20}$ (the weight on the connection to unit 2 from unit 0) will be increased in strength for each pattern by amount $\epsilon$, which in this case we will set to 1.0. On the other hand, $w_{21}$ will be increased by amount $\epsilon$ in two of the cases (first and last pattern) and reduced by $\epsilon$ in the other cases, for a net change of 0.

As a result of this training, then, this simple network would have acquired a positive connection weight to unit 2 from unit 0. This connection will now allow unit 0 to make unit 2 take on an activation value correlated with that of unit 0. At the same time, the network would have acquired a null connection from unit 1 to unit 2, capturing the fact that the activation of unit 1 has no predictive relation to the activation of unit 2. In this way, it is possible to use Hebbian learning to learn associations that depend on the correlation between activations of units in a network.

Unfortunately, the correlational learning that is possible with a Hebbian learning rule is a "unitwise" correlation, and sometimes, these unitwise correlations are not sufficient to learn correct associations between whole input patterns and appropriate responses. To see that this is so, suppose we change our network so that there are now four input units and one output unit, as shown in Figure 4.1C. And suppose we want to train the connections in the network so that the output unit takes on the values given in Figure 4.1D for each of the four input patterns shown there. In this case, the Hebbian learning procedure will not produce correct results. To see why, we need to examine the values of the weights (equivalently, the pairwise correlations of the activations of each sending unit with the receiving unit). What we see is that three of the connections end up with 0 weights because the activation of the corresponding input unit is uncorrelated with the activation of the output unit. Only one of the input units, unit 2, has a positive correlation with unit 4 over this set of patterns. This means that the output unit will make the same response to the first three patterns since in all three of these cases the third unit is on, and this is the only unit with a nonzero connection to the output unit.

Before leaving this example, we should note that there are values of the connection strengths that will do the job. One such set is shown in Figure 4.1E. The reader can check that this set produces the correct results for each of the four input patterns by using Equation 4.3.

Apparently, then, successful learning may require finding connection strengths that are not proportional to the correlations of activations of the units. How can this be done?

### 4.1.2 The Delta Rule

One answer that has occurred to many people over the years is the idea of using the difference between the desired, or *target*, activation and the obtained activation to drive learning. The idea is to adjust the strengths of the connections so that they will tend to reduce this *difference or error* measure. Because the rule is driven by differences, we have tended to call it the delta rule. Others have called it the Widrow-Hoff learning rule or the least mean square (LMS) rule (Widrow and Hoff, 1960); it is related to the perceptron convergence procedure of Rosenblatt (1959).

This learning rule, in its simplest form, can be written

$$\Delta w_{ij} = \epsilon e_i a_j \tag{4.4}$$

where $e_i$, the error for unit $i$, is given by

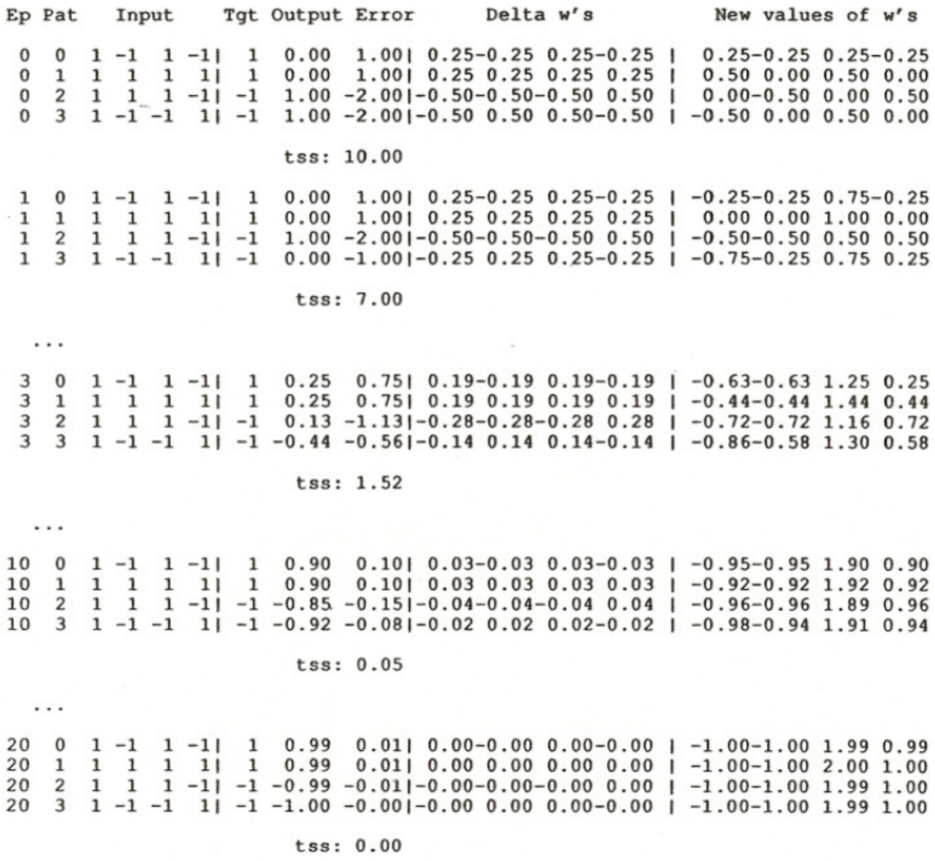
$$e_i = t_i - a_i \tag{4.5}$$

the difference between the teaching input to unit $i$ and its obtained activation.

To see how this rule works, let's use it to train the five-unit network in Figure 4.1C on the patterns in Figure 4.1D. The training regime is a little different here: For each pattern, we turn the input units on, then we see what effect they have on the output unit; its activation reflects the effects of the current connections in the network. (As before we assume the units are linear.) We compute the difference between the obtained output and the teaching input (Equation 4.5). Then, we adjust the strengths of the connections according to Equation 4.4. We will follow this procedure as we cycle through the four patterns several times, and look at the resulting strengths of the connections as we go. The network is started with initial weights of 0. The results of this process for the first cycle through all four patterns are shown in the first four rows of Figure 4.2.

The first time pattern 0 is presented, the response (that is, the obtained activation of the output unit) is 0, so the error is +1. This means that the changes in the weights are proportional to the activations of the input units. A value of 0.25 was used for the learning rate parameter, so each $\Delta w$ is ±0.25. These are added to the existing weights (which are 0), so the resulting weights are equal to these initial increments. When pattern 1 is presented, it happens to be uncorrelated with pattern 0, and so again the obtained output is 0. (The output is obtained by summing up the pairwise products of the inputs on the current trial with the weights obtained at the end of the preceding trial.) Again the error is +1, and since all the input units are on in this case, the change in the weight is +0.25 for each input. When these increments are added to the original weights, the result is a value of +0.5 for $w_{04}$ and $w_{24}$, and 0 for the other weights. When the next pattern is presented, these weights produce an output of +1. The error is therefore −2, and so relatively larger $\Delta w$ terms result. Even so, when the final pattern is presented, it produces an output of +1 as well. When the weights are adjusted to take this into account, the weight from input unit 0 is negative and the weight from unit 2 is positive; the other weights are 0. This completes the first sweep through the set of patterns. At this point, the values of the weights are far from perfect; if we froze them at these values, the network would produce 0 output to the first three patterns. It would produce the correct answer (an output of −1) only for the last pattern.

The correct set of weights is approached asymptotically if the training procedure is continued for several more sweeps through the set of patterns. Each of these sweeps, or *training epochs*, as we will call them henceforth, results in a set of weights that is closer to a perfect solution. To get a measure of the closeness of the approximation to a perfect solution, we can calculate an error measure for each pattern as that pattern is being processed. For each pattern, the error measure is the value of the error $(t - a)$ squared. This measure is then summed over all patterns to get a *total sum of squares* or *tss* measure. The resulting error measure, shown for each of the illustrated epochs in Figure 4.2,



```
Ep Pat    Input     Tgt Output Error       Delta w's              New values of w's

 0  0  1 -1  1 -1|  1   0.00   1.00| 0.25-0.25 0.25-0.25 |  0.25-0.25 0.25-0.25
 0  1  1  1  1  1|  1   0.00   1.00| 0.25 0.25 0.25 0.25 |  0.50 0.00 0.50 0.00
 0  2  1  1  1 -1| -1   1.00 -2.00|-0.50-0.50-0.50 0.50 |  0.00-0.50 0.00 0.50
 0  3  1 -1 -1  1| -1   1.00 -2.00|-0.50 0.50 0.50-0.50 | -0.50 0.00 0.50 0.00

                        tss: 10.00

 1  0  1 -1  1 -1|  1   0.00   1.00| 0.25-0.25 0.25-0.25 | -0.25-0.25 0.75-0.25
 1  1  1  1  1  1|  1   0.00   1.00| 0.25 0.25 0.25 0.25 |  0.00 0.00 1.00 0.00
 1  2  1  1  1 -1| -1   1.00 -2.00|-0.50-0.50-0.50 0.50 | -0.50-0.50 0.50 0.50
 1  3  1 -1 -1  1| -1   0.00 -1.00|-0.25 0.25 0.25-0.25 | -0.75-0.25 0.75 0.25

                         tss: 7.00

  ...

 3  0  1 -1  1 -1|  1   0.25   0.75| 0.19-0.19 0.19-0.19 | -0.63-0.63 1.25 0.25
 3  1  1  1  1  1|  1   0.25   0.75| 0.19 0.19 0.19 0.19 | -0.44-0.44 1.44 0.44
 3  2  1  1  1 -1| -1   0.13 -1.13|-0.28-0.28-0.28 0.28 | -0.72-0.72 1.16 0.72
 3  3  1 -1 -1  1| -1 -0.44 -0.56|-0.14 0.14 0.14-0.14 | -0.86-0.58 1.30 0.58

                         tss: 1.52

  ...

10  0  1 -1  1 -1|  1   0.90   0.10| 0.03-0.03 0.03-0.03 | -0.95-0.95 1.90 0.90
10  1  1  1  1  1|  1   0.90   0.10| 0.03 0.03 0.03 0.03 | -0.92-0.92 1.92 0.92
10  2  1  1  1 -1| -1 -0.85 -0.15|-0.04-0.04-0.04 0.04 | -0.96-0.96 1.89 0.96
10  3  1 -1 -1  1| -1 -0.92 -0.08|-0.02 0.02 0.02-0.02 | -0.98-0.94 1.91 0.94

                         tss: 0.05

  ...

20  0  1 -1  1 -1|  1   0.99   0.01| 0.00-0.00 0.00-0.00 | -1.00-1.00 1.99 0.99
20  1  1  1  1  1|  1   0.99   0.01| 0.00 0.00 0.00 0.00 | -1.00-1.00 2.00 1.00
20  2  1  1  1 -1| -1 -0.99 -0.01|-0.00-0.00-0.00 0.00 | -1.00-1.00 1.99 1.00
20  3  1 -1 -1  1| -1 -1.00 -0.00|-0.00 0.00 0.00-0.00 | -1.00-1.00 1.99 1.00

                         tss: 0.00
```

Figure 4.2: Learning with the delta rule. See text for explanation.

gets smaller over epochs, as do the changes in the strengths of the connections. The weights that result at the end of 20 epochs of training are very close to the perfect solution values. With more training, the weights converge to these values.

The error-correcting learning rule, then, is much more powerful than the Hebb rule. In fact, it can be proven rather easily that the error-correcting rule will find a set of weights that drives the error as close to 0 as we want for each and every pattern in the training set, provided such a set of weights exists. Many proofs of this theorem have been given; a particularly clear one may be found in Minsky and Papert (1969) (one such proof may be found in *PDP:11*).

### 4.1.3 Division of Labor in Error Correcting Learning

It is worth noting an interesting characteristic of error correcting learning rules. This is that, when possible, they divide up the work of driving output units to their correct target values. A simple case of this could arise in the simple network we have already been considering.

Suppose we create a very simple training set consisting of two input patterns and corresponding single desired outputs:

```
 input   output
+ + + +    +
- - - +    -
```

In this case, we see that three of the input units are perfectly correlated with the output and one is uncorrelated with it. If we present these two input patterns repeatedly with a small learning rate, the connection weights will converge to

```
1/3 1/3 1/3 0
```

You can verify that with these connection weights, the network will produce the correct output for both inputs. Now, consider what would happen if the input patterns were:

```
 input   output
+ + + +    +
- + + +    -
```

Here, only one input unit is correlated with the output unit. What will the connection weights converge to in this case? The second, third, and fourth unit cannot help predict the output, so these weights will all be 0. The first unit will have to do all the work, and so the first weight will be 1. While this set of weights would work for the first set of input patterns, the learning rule tends to spread the responsibility or divide the labor among the units that best predict the output. This tendency to ’divide the labor’ among the input units is a characteristic of error correcting learning, and does not occur with the simple Hebbian learning rule because that rule is only sensitive to pairwise input-output correlations.

### 4.1.4 The Linear Predictability Constraint

We have just noted that the delta rule will find a set of weights that solves a network learning problem, provided such a set of weights exists. What are the conditions under which such a set actually does exist?

Such a set of weights exists only if for each input-pattern-target-pair the target can be predicted from a weighted sum, or *linear combination*, of the activations of the input units. That is, the set of weights must satisfy

$$t_{ip} = \sum_j w_{ij} a_{jp} \tag{4.6}$$

for output unit $i$ in all patterns $p$.

This constraint (which we called the *linear predictability constraint* in *PDP:17*) can be overcome by the use of hidden units, but hidden units cannot be trained using the delta rule as we have described it here because (by definition) there is no teacher for them. Procedures for training such units are discussed in Chapter 5.

Up to this point, we have considered the use of the Hebb rule and the delta rule for training connections coming into a single unit. We now consider how these learning rules produce the characteristics of *pattern associator* networks.

## 4.2 THE PATTERN ASSOCIATOR

In a pattern associator, there are two sets of units: input units and output units. There is also a matrix representing the connections from the input units to the output units. A pattern associator is really just an extension of the simple networks we have been considering up to now, in which the number of output units is greater than one and each input unit has a connection to each output unit. An example of an eight-unit by eight-unit pattern associator is shown in Figure 4.3.

The pattern associator is a device that learns associations between input patterns and output patterns. It is interesting because what it learns about one pattern tends to generalize to other similar patterns. In what follows we will see how this property arises, first in the simplest possible pattern associator: a pattern associator consisting of linear units, trained by the Hebb rule.[1]

### 4.2.1 The Hebb Rule in Pattern Associator Models

To begin, let us consider the effects of training a network with a single learning trial $l$, involving an input pattern $\boldsymbol{i_l}$, and an output pattern $\boldsymbol{o_l}$. We will use the notational convention that vector names are bolded.

[1] Readers who wish to gain a better grasp on the mathematical basis of this class of models may find it worthwhile to read *PDP:9*. An in-depth analysis of the delta rule in pattern associators is in *PDP:11*.

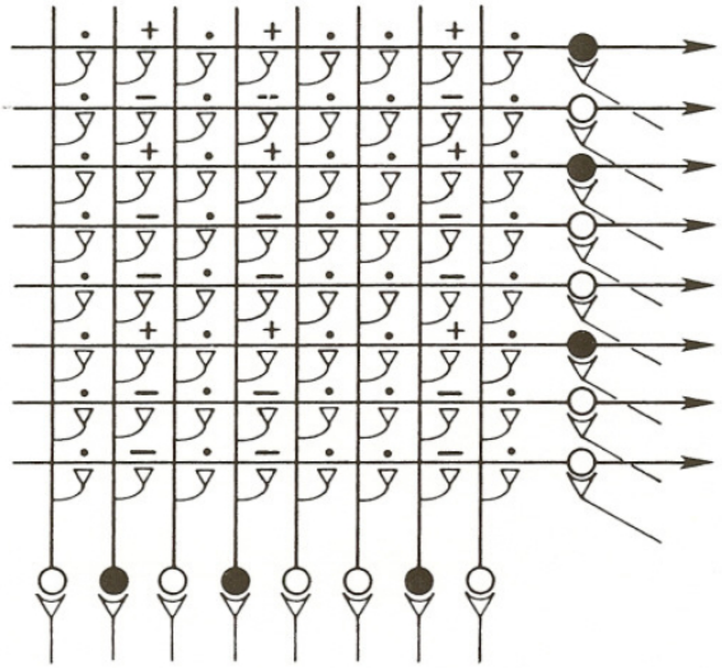


Figure 4.3: A schematic diagram of an eight-unit pattern associator. An input pattern, an output pattern, and values for the weights that will allow the input to produce the output are shown. (From *PDP:18*, p. 227.)

Assuming all the weights in the network are initially 0, we can express the value of each weight as

$$w_{ij} = \epsilon i_{jl} o_{il} \tag{4.7}$$

Note that we are using the variable $i_{jl}$ to stand for the activation of input unit $j$ in input pattern $\boldsymbol{i_l}$, and we are using $o_{il}$ to stand for the activation of output unit $i$ in output pattern $\boldsymbol{o_l}$. Thus, each weight is just the product of the activation of the input unit times the activation of the output unit in the learning trial $l$.

In this chapter, many of the formulas are also presented as MATLAB routines to further familiarize the reader with the MATLAB operations. In these routines, the subscript on the vector names will be dropped when clear. Thus, $\boldsymbol{i_l}$ will just be denoted $i$ in the code. Vectors are assumed to be row vectors.

In MATLAB, the above formula (Eq. 4.7) is an outer product:

```
W = epsilon * (o’ * i);
```

where the prime is the transpose operator. Dimensions of the outer product are the outer dimensions of the contributing vectors: $o'$ dims are [8 1], $i$ dims are [1 8], and so $W$ dims are [8 8]. We also adopt the convention that weight matrices are of size [noutputs ninputs].

Now let us present a test input pattern, $\boldsymbol{i_t}$, and examine the resulting output pattern it produces. Since the units are linear, the activation of output unit $i$ when tested with input pattern $\boldsymbol{i_t}$ is

$$o_{it} = \sum_j w_{ij} i_{jt} \tag{4.8}$$

which is equivalent to

```
o = W * i’;
```

in MATLAB, where $o$ is a column vector. Substituting for $w_{ij}$ from Equation 4.7 yields

$$o_{it} = \sum_j \epsilon i_{jl} o_{il} i_{jt} \tag{4.9}$$

Since we are summing with respect to $j$ in this last equation, we can pull out $\epsilon$ and $o_{il}$:

$$o_{it} = \epsilon o_{il} \sum_j i_{jl} i_{jt} \tag{4.10}$$

Equation 4.10 says that the output at the time of test will be proportional to the output at the time of learning times the sum of the elements of the input pattern at the time of learning, each multiplied by the corresponding element of the input pattern at the time of test.

This sum of products of corresponding elements is called the *dot product*. It is very important to our analysis because it expresses the *similarity* of the two patterns $i_l$ and $i_t$. It is worth noting that we have already encountered an expression similar to this one in Equation 4.2. In that case, though, the quantity was proportional to the correlation of the activations of two *units* across an ensemble of *patterns*. Here, it is proportional to the correlation of two *patterns* across an ensemble of *units*. It is often convenient to normalize the dot product by taking out the effects of the number of elements in the vectors in question by dividing the dot product by the number of elements. We will call this quantity the *normalized dot product*. For patterns consisting of all +1s and −1s, it corresponds to the correlation between the two patterns. The normalized dot product has a value of 1 if the patterns are identical, a value of −1 if they are exactly opposite to each other, and a value of 0 if the elements of one vector are completely uncorrelated with the elements of the other. To compute the normalized dot product with MATLAB:

```
ndp = sum(a.*b)/length(a);
```

or for two row vectors

```
ndp = (a*b')/length(a);
```

We can rewrite Equation 4.10, then, replacing the summed quantity by the normalized dot product of input pattern $\boldsymbol{i_l}$ and input pattern $\boldsymbol{i_t}$, which we denote by $(\boldsymbol{i_l} \cdot \boldsymbol{i_t})_n$:

$$o_{it} = ko_{il}(\boldsymbol{i_l} \cdot \boldsymbol{i_t})_n \tag{4.11}$$

where $k = n\epsilon$ ($n$ is the number of units). Since Equation 4.11 applies to all of the elements of the output pattern $o_t$, we can write

$$\boldsymbol{o_t} = k\boldsymbol{o_l}(\boldsymbol{i_l} \cdot \boldsymbol{i_t})_n \tag{4.12}$$

In MATLAB, this is

```
ot = k * ol * sum(it .* il) / length(it);
```

This result is very basic to thinking in terms of patterns since it demonstrates that what is crucial for the performance of the network is the similarity relations among the input patterns–their correlations–rather than their specific properties considered as individuals.[2] Thus Equation 4.12 says that the output pattern produced by our network at test is a scaled version of the pattern stored on the learning trial. The magnitude of the pattern is proportional to the similarity of the learning and test patterns. In particular, if $k = 1$ and if the test pattern is identical to the training pattern, then the output at test will be identical to the output at learning.

An interesting special case occurs when the normalized dot product between the learned pattern and the test pattern is 0. In this case, the output is 0: There is no response whatever. Patterns that have this property are called *orthogonal* or *uncorrelated*; note that this is not the same as being opposite or *anticorrelated*.

[2] Technically, performance depends on the similarity relations among the patterns and on their overall strength or magnitude. However, among vectors of equal strength (e.g., the vectors consisting of all +1s and −1s), only the similarity relations are important.

To develop intuitions about orthogonality, you should compute the normalized dot products of each of the patterns $b$, $c$, $d$, and $e$ below with pattern $a$:

```
a = [ 1  1 -1 -1]
b = [ 1 -1  1 -1]
c = [ 1 -1 -1  1]
d = [ 1  1  1  1]
e = [-1 -1  1  1]

ndp_ab = sum(a.*b)/length(a);
```

You will see that patterns $b$, $c$, and $d$ are all orthogonal to pattern $a$; in fact, they are all orthogonal to each other. Pattern $e$, on the other hand, is not orthogonal to pattern $a$, but is anticorrelated with it. Interestingly, it forms an orthogonal set with patterns $b$, $c$, and $d$. When all the members of a set of patterns are orthogonal to each other, we call them an *orthogonal set*.

Now let us consider what happens when an entire ensemble of patterns is presented during learning. In the Hebbian learning situation, the set of weights resulting from an ensemble of patterns is just the sum of the sets of weights resulting from each individual pattern. Note that, in the model we are considering, the output pattern, when provided, is always thought of as clamping the state the output units to the indicated values, so that the existing values of the weights actually play no role in setting the activations of the output units. Given this, after learning trials on a set of input patterns $\boldsymbol{i_l}$ each paired with an output pattern $\boldsymbol{o_l}$, the value of each weight will be

$$w_{ij} = \epsilon \sum_l i_{jl} o_{il} \tag{4.13}$$

Thus, the output produced by each test pattern is

$$\boldsymbol{o_t} = k \sum_l \boldsymbol{o_l} (\boldsymbol{i_l} \cdot \boldsymbol{i_t})_n \tag{4.14}$$

In words, the output of the network in response to input pattern $t$ is the sum of the output patterns that occurred during learning, with each pattern's contribution weighted by the similarity of the corresponding input pattern to the test pattern. Three important facts follow from this:

1. If a test input pattern is orthogonal to all training input patterns, the output of the network will be 0; there will be no response to an input pattern that is completely orthogonal to all of the input patterns that occurred during learning.

2. If a test input pattern is similar to one of the learned input patterns and is uncorrelated with all the others, then the test output will be a scaled version of the output pattern that was paired with the similar input pattern during learning. The magnitude of the output will be proportional to the similarity of the test input pattern to the learned input pattern.

3. For other test input patterns, the output will always be a blend of the training outputs, with the contribution of each output pattern weighted by the similarity of the corresponding input pattern to the test input pattern.

In the exercises, we will see how these properties lead to several desirable features of pattern associator networks, particularly their ability to generalize based on similarity between test patterns and patterns presented during training.

These properties also reflect the limitations of the Hebbian learning rule; when the input patterns used in training the network do not form an orthogonal set, it is not in general possible to avoid contamination, or “cross-talk,” between the response that is appropriate to one pattern and the response that occurs to the others. This accounts for the failure of Hebbian learning with the second set of training patterns considered in Figure 4.1. The reader can check that the input patterns we used in our first training example in Figure 4.1 (which was successful) were orthogonal but that the patterns used in the second example were not orthogonal.

### 4.2.2 The Delta Rule in Pattern Associator Models

Once again, the delta rule allows us to overcome the orthogonality limitation imposed by the Hebb rule. For the pattern associator case, the delta rule for a particular input-target pair $\boldsymbol{i_l}$, $\boldsymbol{t_l}$ is

$$\Delta w_{ij} = \epsilon(t_{il} - o_{il})i_{jl}. \tag{4.15}$$

which in MATLAB is (again, assuming row vectors)

```
delta_w = epsilon * (t-o)’ * i;
```

Therefore the weights that result from an ensemble of learning pairs indexed by $l$ can be written:

$$w_{ij} = \epsilon \sum_l (t_{il} - o_{il})i_{jl} \tag{4.16}$$

It is interesting to compare this to the Hebb rule. Consider first the case where each of the learned patterns is orthogonal to every other one and is presented exactly once during learning. Then $\boldsymbol{o_l}$ will be $\mathbf{0}$ (a vector of all zeros) for all learned patterns $l$, and the above formula reduces to

$$w_{ij} = \epsilon \sum_l t_{il} i_{jl} \tag{4.17}$$

In this case, the delta rule produces the same results as the Hebb rule; the teaching input simply replaces the output pattern from Equation 4.13. As long as the patterns remain orthogonal to each other, there will be no cross-talk between patterns. Learning will proceed independently for each pattern. There is one difference, however. If we continue learning beyond a single epoch, the delta rule will stop learning when the weights are such that they allow the network to produce the target patterns exactly. In the Hebb rule, the weights will grow linearly with each presentation of the set of patterns, getting stronger without bound.

In the case where the input patterns $\boldsymbol{i_l}$, are not orthogonal, the results of the two learning procedures are more distinct. In this case, though, we can observe the following interesting fact: We can read Equation 4.15 as indicating that the change in the weights that occurs on a learning trial is storing an association of the input pattern with the *error* pattern; that is, we are adding to each weight an increment that can be thought of as an association between the *error* for the output unit and the activation of the input unit. To see the implications of this, let's examine the effects of a learning trial with input pattern $\boldsymbol{i_l}$ paired with output pattern $\boldsymbol{t_l}$ on the output produced by test pattern $\boldsymbol{i_t}$. The effect of the change in the weights due to this learning trial (as given by Equation 4.15) will be to change the output of some output unit $i$ by an amount proportional to the error that occurred for that unit on the learning trial, $e_i$, times the dot product of the learned pattern with the test pattern:

$$\Delta o_{it} = ke_{il}(\boldsymbol{i_l} \cdot \boldsymbol{i_t})_n$$

Here $k$ is again equal to $\epsilon$ times the number of input units $n$. In vector notation, the change in the output pattern $\boldsymbol{o_t}$ can be expressed as

$$\Delta \boldsymbol{o_t} = k\boldsymbol{e_l}(\boldsymbol{i_l} \cdot \boldsymbol{i_t})_n$$

Thus, the change in the output pattern at test is proportional to the error vector times the normalized dot product of the input pattern that occurred during learning and the input pattern that occurred during test. Two facts follow from this:

1. If the input on the learning trial is identical to the input on the test trial so that the normalized dot product is 1.0 and if k = 1.0, then the change in the output pattern will be exactly equal to the error pattern. Since the error pattern is equal to the difference between the target and the obtained output on the learning trial, this amounts to one trial learning of the desired association between the input pattern on the training trial and the target on this trial.

2. However, if $\boldsymbol{i_t}$ is different from $\boldsymbol{i_l}$ but not completely different so that $(\boldsymbol{i_l} \cdot \boldsymbol{i_t})_n$ n is not equal to either 1 or 0, then the output produced by $\boldsymbol{i_t}$ will be affected by the learning trial. The magnitude of the effect will be proportional to the magnitude of $(\boldsymbol{i_l} \cdot \boldsymbol{i_t})_n$.

The second effect–the transfer from learning one pattern to performance on another–may be either beneficial or interfering. Importantly, for patterns of all +1s and −1s, the transfer is always less than the effect on the pattern used on the learning trial itself, since the normalized dot product of two different patterns must be less than the normalized dot product of a pattern with itself. This fact plays a role in several proofs concerning the convergence of the delta rule learning procedure (see Kohonen, 1977, and *PDP:11* for further discussion).

### 4.2.3 Linear Predictability and the Linear Independence Requirement

Earlier we considered the linear predictability constraint for training a single output unit. Since the pattern associator can be viewed as a collection of several different output units, the constraint applies to each unit in the pattern associator. Thus, to master a set of patterns there must exist a set of weights $w_{ij}$ such that

$$t_{ip} = \sum_j w_{ij} i_{jp} \tag{4.18}$$

for all output units $i$ for all target-input pattern pairs $p$. A consequence of this constraint for the sets of input-output patterns that can be learned by a pattern associator is something we will call the *linear independence requirement*:

> An *arbitrary* output pattern $\boldsymbol{o_p}$ can be correctly associated with a particular input pattern $\boldsymbol{i_p}$ without ruining associations between other input-output pairs, only if $\boldsymbol{i_p}$ is linearly independent of all of the other patterns in the training set, that is, as long as $\boldsymbol{i_p}$ *cannot* be written as a linear combination of the other input patterns.

If pattern $\boldsymbol{i_p}$ *can* be written as a linear combination of the other input patterns, then the output for $\boldsymbol{i_p}$ will be a linear combination of the outputs produced by the other patterns (each other pattern's contribution to the linear combination will be weighted by it's similarity to $\boldsymbol{i_p}$). If this linear combination just happens to be the correct output for $\boldsymbol{i_p}$, then all is well, but if it is not, the weights will be changed by the delta rule, and this will distort the output pattern produced by one or more of the other patterns in the set.

A pattern that cannot be written as a linear combination of a set of other patterns is said to be *linearly independent* from these other patterns. When all the members of a set of patterns are linearly independent, we say they form a *linearly independent* set. To ensure that arbitrary associations to each of a set of input patterns can be learned, the input patterns must form a linearly independent set.

Although this is a serious limitation on what a network can learn, it is worth noting that there are cases in which the response that we need to make to one input pattern can be predictable from the responses that we make to other patterns with which they overlap. In these cases, the fact that the pattern associator produces a response that is a combination of the responses to other patterns can allow it to produce very efficient, often rule-like solutions to the problem of mapping each of a set of input patterns to the appropriate response. We will examine this property of pattern associators in the exercises.

### 4.2.4 Nonlinear Pattern Associators

Not all pattern associator models that have been studied in the literature make use of the linear activation assumptions we have been using in this analysis.

Several different kinds of nonlinear pattern associators (i.e., associators in which the output units have nonlinear activation functions) fall within the general class of pattern associator models. These nonlinearities have effects on performance, but the basic principles that we have observed here are preserved even when these nonlinearities are in place. In particular:

1. Orthogonal inputs are mutually transparent.
2. The learning process converges with the delta rule as long as there is a set of weights that will solve the learning problem.
3. A set of weights that will solve the problem does not always exist.
4. What is learned about one pattern tends to transfer to others.

## 4.3 THE FAMILY OF PATTERN ASSOCIATOR MODELS

With the above as background, we turn to a brief specification of several members of the class of pattern associator models that are available through the **pa** program. These are all variants on the pattern associator theme. Each model consists of a set of input units and a set of output units. The activations of the input units are clamped by externally supplied input patterns. The activations of the output units are determined in a single two-phase processing cycle. First, the net input to each output unit is computed. This is the sum of the activations of the input units times the corresponding weights, plus an optional bias term associated with the output unit:

$$net_i = \sum_j w_{ij} a_j + bias_i \tag{4.19}$$

### 4.3.1 Activation Functions

After computing the net input to each output unit, the activation of the output unit is then determined according to an activation function. Several variants are available:

- *Linear.* Here the activation of output unit $i$ is simply equal to the net input.
- *Linear threshold.* In this variant, each of the output units is a *linear threshold unit*; that is, its activation is set to 1 if its net input exceeds 0 and is set to 0 otherwise. Units of this kind were used by Rosenblatt in his work on the perceptron 1959.
- *Stochastic.* This is the activation function used in *PDP:18* and *PDP:19.* Here, the output is set to 1, with a probability $p$ given by the logistic

function:

$$p(o_i = 1) = \frac{1}{1 + e^{-net_i/T}} \tag{4.20}$$

This is the same activation function used in Boltzmann machines.

- *Continuous sigmoid.* In this variant, each of the output units takes on an activation that is nonlinearly related to its input according to the logistic function:

  $$o_i = \frac{1}{1 + e^{-net_i/T}} \tag{4.21}$$

  Note that this is a continuous function that transforms net inputs between $+\infty$ and $-\infty$ into real numbers between 0 and 1. This is the activation function used in the back propagation networks we will study in Chapter 5.

### 4.3.2 Learning Assumptions

Two different learning rules are available in the **pa** program:

- *The Hebb rule.* Hebbian learning in the pattern associator model works as follows. Activations of input units are clamped based on an externally supplied input pattern, and activations of the output units are clamped to the values given by some externally supplied target pattern. Learning then occurs by adjusting the strengths of the connections according to the Hebbian rule:

  $$\Delta w_{ij} = \epsilon o_i i_j \tag{4.22}$$

- *The delta rule.* Error-correcting learning in the pattern associator model works as follows. Activations of input units are clamped to values determined by an externally supplied input pattern, and activations of the output units are calculated as described earlier. The difference between the obtained activation of the output units and the target activation, as specified in an externally supplied target pattern, is then used in changing the weights according to the following formula:

  $$\Delta w_{ij} = \epsilon(t_i - o_i)i_j \tag{4.23}$$

### 4.3.3 The Environment and the Training Epoch

In the pattern associator models, there is a notion of an *environment* of pattern pairs. Each pair consists of an input pattern and a corresponding output pattern. A training *epoch* consists of one learning trial on each pattern pair in the environment. On each trial, the input is presented, the corresponding output is computed, and the weights are updated. Patterns may be presented in fixed sequential order or in permuted order within each epoch.

### 4.3.4 Performance Measures

After processing each pattern, several measures of the output that is produced and its relation to the target are computed. One of these is the normalized dot product of the output pattern with the target. This measure is called the *ndp*. We have already described this measure quantitatively; here we note that it gives a kind of combined indication of the similarity of two patterns and their magnitudes. In the cases where this measure is most useful-where the target is a pattern of +1s and −1s–the magnitude of the target is fixed and the normalized dot product varies with the similarity of the output to the target and the magnitude of the output itself. To unconfound these factors, we provide two further measures: the normalized vector length, or *nvl*, of the output vector and the vector correlation, or *vcor*, of the output vector with the target vector. The *nvl* measures the magnitude of the output vector, normalizing for the number of elements in the vector. It has a value of 1.0 for vectors consisting of all +1s and −1s. The *vcor* measures the similarity of the vectors independent of their length; it has a value of 1.0 for vectors that are perfectly correlated, 0.0 for orthogonal vectors, and −1.0 for anticorrelated vectors.

Quantitative definitions of vector length and vector correlation are given in *PDP:9* (pp. 376-379). The vector length of vector $\boldsymbol{v}$, $||\boldsymbol{v}||$, is the square root of the dot product of a vector with itself:

$$||\boldsymbol{v}|| = \sqrt{\boldsymbol{v} \cdot \boldsymbol{v}}$$

and the vector correlation (also called the cosine of the angle between two vectors) is the dot product of the two vectors divided by the product of their lengths:

$$vcor(\boldsymbol{u}, \boldsymbol{v}) = \frac{(\boldsymbol{u} \cdot \boldsymbol{v})}{||\boldsymbol{u}||\ ||\boldsymbol{v}||}$$

The normalized vector length is obtained by dividing the length by the square root of number of elements. Given these definitions, we can now consider the relationships between the various measures. When the target pattern consists of +1s and −1s, the normalized dot product of the output pattern and the target pattern is equal to the normalized vector length of the output pattern times the vector correlation of the output pattern and the target:

$$ndp = nvl \cdot vcor. \tag{4.24}$$

In addition to these measures, we also compute the *pattern sum of squares* or *pss* and the *total sum of squares* or *tss*. The *pss* is the sum over all output units of the squared error, where the error for each output unit is the difference between the target and the obtained activation of the unit. This quantity is computed for each pattern processed. The *tss* is just the sum over the *pss* values computed for each pattern in the training set. These measures are not very meaningful when learning occurs by the Hebb rule, but they are meaningful when learning occurs by the delta rule.

## 4.4 IMPLEMENTATION

The **pa** program implements the pattern associator models in a very straightforward way. The program is initialized by defining a network, as in previous chapters. A PA network consists of a pool of input units (*pool(2)*) and a pool of output units (*pool(3)*). *pool(1)* contains the bias unit which is always on but is not used in these exercises. Connections are allowed from input units to output units only. The network specification file (*pa.net*) defines the number of input units and output units, as well as the total number of units, and indicates which connections exist. It is also generally necessary to read in a file specifying the set of pattern pairs that make up the environment of the model.

Once the program is initialized, learning occurs through calls to a routine called *train*. This routine carries out nepochs of training, where the training mode can be selected in the Train options window. *strain* trains the network with patterns in sequential order, while *ptrain* permutes the order. The number of epochs can also be set in that window. The routine exits if the total sum of squares measure, *tss*, is less than some criterion value, *ecrit* which can also be set in Train options. Here is the *train* routine:

```
function train()
for iter = 1:nepochs
     patn = getpatternrange(data,options);
     for p = 1:npatterns
          pno = patn(p);
          setinput(data,pno,options);
          compute_output(data,pno,options);
          compute_error;
          sumstats;
          if (options.lflag)
               change_weights(options);
          end
          if (net.tss < options.ecrit)
               return;
          end
     end
end
```

This calls four other routines: one that sets the input pattern (*setinput*), one that computes the activations of the output units from the activations of the input units (*compute_output*), one that computes the error measure (*compute_error*), and one that computes the various summary statistics (*sumstats*).

Below we show the *compute_output* and the *compute_error* routines. First, *compute_output*:

```
function compute_output(pattern,patnum,opts)

p=net.pool(3); %reference to the output pool
```

```
p.netinput = net.pool(2).output * p.proj.weight';
switch opts.actfunction
     case 'st'  %stochastic
          logout = logistic(p.netinput, opts.temperature);
          r = rand(1,n);
          p.output(r < logout) = 1.0;
          p.output(r >= logout) = 0.0;
     case 'li'   %linear
           p.output = p.netinput;
     case 'cs'   %continuous sigmoid
          p.output = logistic(p.netinput,opts.temperature);
     case 'lt'   %linear threshold
          p.output(p.netinput > 0) = 1.0;
          p.output(p.netinput <= 0) = 0.0;
end
```

The activation function can be selected from the menu in Train options. There are represented in the code as Linear (*li*), Linear threshold (*lt*), Stochastic (*st*), and Continuous Sigmoid (*cs*). With the linear activation function, the output activation is just the net input. For linear threshold, activation is 1 if the net input is greater than 0, and 0 otherwise. The continuous sigmoid function calls the logistic function shown in Chapter 3. This function returns a number between 0 and 1. For stochastic activation, the logistic activation is first calculated and the result is then used to set the activation of the unit to 0 or 1 using the logistic activation as the probability. The activation function can be specified separately for training and testing via the Train and Test options.

The *compute_error* function is exceptionally simple for the pa program:

```
function compute_error()
net.pool(3).error = net.pool(3).target - net.pool(3).output;
```

Note that when the targets and the activations of the output units are both specified in terms of 0s and 1s, the error will be 0, 1, or -1.

If learning is enabled (as it is by default in the program, as indicated by the value of the *lflag* variable which corresponds to the learn checkbox under Train options), the *train* routine calls the *change_weights* routine, which actually carries out the learning:

```
function change_weights(opts)
p = net.pool(3); %output pool
if (hebb)
     p.output = p.target;
     scalewith = p.output;
else %delta rule instead
     scalewith = p.error;
end
p.proj.weight = p.proj.weight + (scalewith' * net.pool(2).output * lr);
net.pool(3) = p;
```

*Hebb* and *Delta* are the two possible values of the *lrule* field under Train options. The *lr* variable in the code corresponds to the learning rate, which is set by the *lrate* field in Train options.

Note that for Hebbian learning, we use the target pattern directly in the learning rule, since this is mathematically equivalent to clamping the activations of the output units to equal the target pattern and then using these activations.

## 4.5 RUNNING THE PROGRAM

The **pa** program is used much like the other programs we have described in earlier chapters. The main things that are new for this program are the *strain* and *ptrain* options for training pattern associator networks.

Training or Testing are selected with the radio button just next to the "options" button. The "Test all" radio button in the upper right corner of the test panel allows you to test the network's response to all of the patterns in the list of pattern pairs with learning turned off so as not to change the weights while testing.

As in the **cs** program, the *newstart* and *reset* buttons are both available as alternative methods for reinitializing the programs. Recall that *reset* reinitializes the random number generator with the same seed used the last time the program was initialized, whereas *newstart* seeds the random number generator with a new random seed. Although there can be some randomness in *pa*, the problem of local minima does not arise and different random sequences will generally produce qualitatively similar results, so there is little reason to use *reset* as opposed to *newstart*.

As mentioned in "Implementation" (Section 4.4), there are several activation functions, and linear is default. Also, *Hebb* and *Delta* are alternative rules under *lrule* in Train options. *nepochs* in Train options is the number of training epochs run when the "Run" button is pushed in the train panel on the main window. *ecrit* is the stop criterion value for the error measure. The step-size for the screen updates during training can be set to *pattern*, *cycle* or *epoch* (default) in the train panel on the main window. When *pattern* is selected and the network is run, the window is updated for every pattern trial of every epoch. If the value is *cycle*, the screen is updated after processing each pattern and then updated again after the weights are changed for each pattern. Likewise, *epoch* updates the window just once per epoch after all pattern presentations, which is the fastest but shows the fewest updates.

There are other important options under the Train options. *lrate* sets the learning rate, which is equivalent to the parameter $\epsilon$ from the Background section (4.1). *noise* determines the amount of random variability added to elements of input and target patterns, and *temp* is used as the denominator of the logistic function to scale net inputs with the continuous sigmoid and with the stochastic activation function.

There are also several new performance measures displayed on the main window: the normalized dot product, *ndp*; the normalized vector length measure, *nvl*; the vector correlation measure, *vcor*; the pattern sum of squares, *pss*; and the total sum of squares, *tss*.

### 4.5.1 Commands and Parameters

Here follows a more detailed description of the new commands and parameters in **pa**:

**newstart** Button on the Network Viewer, in the train panel. It seeds the random number with a new random seed, and then returns the program to its initial state before any learning occurred. That is, sets all weights to 0, and sets nepochs to 0. Also clears activations and updates the display.

**ptrain** Option under *trainmode* in the Train options. This option, when the network is trained, presents each pattern pair in the pattern list once in each epoch. Order of patterns is rerandomized for each epoch.

**reset** Button on the main network window. Same as newstart, but reseeds the random number generator with the same seed that was used last time the network was initialized.

**strain** Option under *trainmode* in the Train options. This option, when the network is trained, pairs are presented in the same, fixed order in each epoch. The order is simply the order in which the pattern pairs are encountered in the list.

**Test all** Radio button on the test panel on the Network Viewer. If this option is checked and testing is run, the network will test each testing pattern in sequence. Pressing the step button will present each one by one for better viewing. If it is not checked, the network will test just the selected test pattern. To select a pattern, click on it in the Testing Patterns frame.

**ecrit** Parameter in Train options. Error criterion for stopping training. If the *tss* at the end of an epoch of training is less than this, training stops.

**lflag** Check box in Train options. Normally checked, it enables weight updates during learning.

**nepochs** Number of training epochs conducted each time the run button is pressed.

**Update After** Field in the train and test windows of Network Viewer. Values in the menu are *cycle*, *pattern*, and *epoch*. If the value is *cycle*, the screen is updated after processing each pattern and then updated again after the weights are changed. This only applies for training. If the value is *pattern*, the screen is only updated after the weights are changed. If the value is *epoch*, the screen is updated at the end of each epoch. The number field to the left of this option controls how many cycles, patterns, or epochs occur before an update is made.

**actfunction** Field in Train options or Test options. Select from linear, linear threshold, stochastic, or continuous sigmoid.

**lrule** Field in Train options. Select between the Hebb and Delta update rules.

**lrate** Parameter in Train options. Scales the size of the changes made to the weights. Generally, if there are $n$ input units, the learning rate should be less than or equal to $1/n$.

**noise** Parameter in Train and Test options. Range of the random distortion added to each input and target pattern specification value during training and testing. The value added is uniformly distributed in the interval $[-noise, +noise]$.

**temp** Denominator used in the logistic function to scale net inputs in both the continuous sigmoid and stochastic modes. Generally, temp can be set to 1. Note that there is only one cycle of processing in **pa**, so there is no annealing.

### 4.5.2 State Variables

Sate variables are all associated with the net structure, and some are available for viewing on the Network Viewer. Type "net" at the MATLAB command prompt after starting an exercise to access these variables.

**cpname** Name of the current pattern, as given in the pattern file.

**epochno** Number of the current epoch; updated at the beginning of each epoch.

**error** Vectors of errors, or differences between the current target pattern and the current pattern of activation over the output units.

**input** Vector of activations of the input units in the network, based on the current input pattern (subject to the effects of noise). Type net.pool(2).input in the MATLAB command prompt to view this.

**ndp** Normalized dot product of the obtained activation vector over the output units and the target vector.

**netinput** Vector of net inputs to each output unit. Type net.pool(3).netinput in the MATLAB command prompt to view this.

**nvl** Normalized length of the obtained activation vector over the output units.

**output** Vector of activations of the output units in the network. Type net.pool(3).output to view.

**patno** The number of the current pattern, updated at the beginning of processing the pattern. Note that this is the index of the pattern on the program's pattern list; when *ptrain* is used, it is not the same as the pattern's position within the random training sequence in force for a particular epoch.

**pss** Pattern sum of squares, equal to the sum over all output units of the squared difference between the target for each unit and the obtained activation of the unit.

**target** Vector of target values for output units, based on the current target pattern, subject to effects of noise.

**tss** Total sum of squares, equal to the sum of all patterns so far presented during the current epoch of the pattern sum of squares.

**vcor** Vector correlation of the obtained activation vector over the output units and the target vector.

## 4.6 OVERVIEW OF EXERCISES

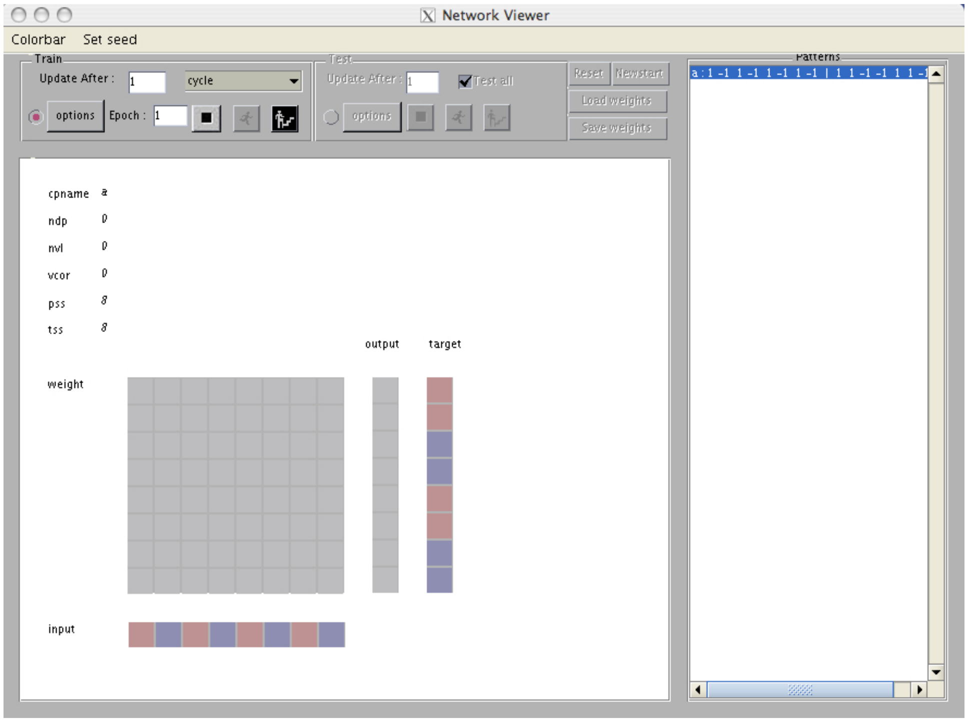
In these exercises, we will study several basic properties of pattern associator networks, starting with their tendency to generalize what they have learned to do with one input pattern to other similar patterns; we will explore the role of similarity and the learning of responses to unseen prototypes. These first studies will be done using a completely linear Hebbian pattern associator. Then, we will shift to the linear delta rule associator of the kind studied by Kohonen (1977) and analyzed in *PDP:11.* We will study what these models can and cannot learn and how they can be used to learn to get the best estimate of the correct output pattern, given noisy input and outputs. Finally, we will examine the acquisition of a rule and an exception to the rule in a nonlinear (stochastic) pattern associator.

### Ex4.1. Generalization and Similarity With Hebbian Learning

In this exercise, you will train a linear Hebbian pattern associator on a single input-output pattern pair, and study how its output, after training, is affected by the similarity of the input pattern used at test to the input pattern used during training.

Open MATLAB, and make sure your path is set to include pdptool and all its children, and then move into the pdptool/pa directory. Type “lin” at the MATLAB command prompt. This sets up the network to be a linear Hebbian pattern associator with eight input units and eight output units, starting with initial weights that are all 0. The lin.m file sets the value of the learning rate parameter to 0.125, which is equal to 1 divided by the number of units. With this value, the Hebb rule will learn an association between a single input pattern consisting of all +1s and −1s and any desired output pattern perfectly in one trial.

The file one.pat is loaded and contains a single pattern (or, more exactly, a single input-output pattern pair) to use for training the associator. Both the input pattern and the output pattern are eight-element vectors of +1s and −1s.



Update After : 1 cycle

Epoch : 1

Update After : 1 ☑ Test all

cpname a

ndp 0

nvl 0

vcor 0

pss 8

tss 8

a : 1 -1 1 -1 1 -1 1 -1 | 1 1 -1 -1 1 1 -1

Figure 4.4: Display layout for the first **pa** exercise while processing pattern *a*, before any learning has occurred.

Now you can train the network on this first pattern pair for one epoch. Select the train panel, and then select *cycle* in the train panel. With this option, the program will present the first (and, in this case, only) input pattern, compute the output based on the current weights, and then display the input, output, and target patterns, as well as some summary statistics. If you click "step" in the train panel, the network will pause after the pattern presentation.

In the upper left corner of the display area, you will see some summary information, including the current *ndp*, or normalized dot product, of the output obtained by the network with the target pattern; the *nvl*, or normalized vector length, of the obtained output pattern; and the *vcor*, or vector correlation, of the output with the target. All of these numbers are 0 because the weights are 0, so the input produces no output at all. Below these numbers are the *pss*, or pattern sum of squares, and the *tss*, or total sum of squares. They are the sum of squared differences between the target and the actual output patterns. The first is summed over all output units for the current pattern, and the second is summed over all patterns so far encountered within this epoch (they are, therefore, identical at this point).

Below these entries you will see the weight matrix on the left, with the input vector that was presented for processing below it and the output and target vectors to the right. The display uses shade of red for positive values and shades of blue for negative values as in previous models. A value of +1 or −1 is not very saturated, so that a value can be distinguished over a larger range.

The window of the right of the screen shows the patterns in use for training or test, whichever is selected. Input and target patterns are separated by a vertical separator. You will see that the input pattern shown below the weights matches the single input pattern shown on the right panel and that the target pattern shown to the right of the weights matches the single target pattern to the right of the vertical separator.

If you click step a second time, the target will first be clamped onto the output units, then the weights will be updated according to the Hebbian learning rule:

$$\Delta w_{ij} = (lrate) o_i i_j \tag{4.25}$$

Q.4.1.1.

> Explain the values of the weights in rows 2 and 3 (counting from 1, which is the convention in MATLAB). Explain the values of the weights in column 8, the last column of the matrix. You can examine the weight values by rolling over them.

Now, with just this one trial of learning, the network will have “mastered” this particular association, so that if you test it at this point, you will find that, given the learned input, it perfectly reproduces the target. You can test the network using the test command. Simply select the test panel, then click step. In this particular case the display will not change much because in the previous display the output had been clamped to reflect the very target pattern that the network has now computed. The only thing that actually changes in the display are the *ndp*, *vcor*, and *nvl* fields; these will now reflect the normalized dot product and correlation of the computed output with the target and the normalized length of the output. They should all be equal to 1.0 at this point.

You are now ready to test the generalization performance of the network. You can enter patterns into a file. Start by opening the “one.pat” file, copy the existing pattern and paste several times in a new .pat file. Save this file as “gen.pat”. Edit the input pattern entries for the patterns and give each pattern its own name. See Q.4.1.2 for information on the patterns to enter. Leave the target part of the patterns the same. Then, click Test options, click Load new, and load the new patterns for testing.

Q.4.1.2.

> Try at least 4 different input patterns, testing each against the original target. Include in your set of patterns one that is orthogonal to the training pattern and one that is perfectly anticorrelated with it, as well as one or two others with positive normalized dot products

> with the input pattern. Report the input patterns, the output pattern produced, and the *ndp*, *vcor*, and *nvl* in each case. Relate the obtained output to the specifics of the weights and the input patterns used and to the discussion in the "Background" section (4.1) about the test output we should get from a linear Hebbian associator, as a function of the normalized dot product of the input vector used at test and the input vector used during training.

If you understand the results you have obtained in this exercise, you understand the basis of similarity-based generalization in one-layer associative networks. In the process, you should come to develop your intuitions about vector similarity and to clearly be able to distinguish uncorrelated patterns from anticorrelated ones.

## Ex4.2. Orthogonality, Linear Independence, and Learning

This exercise will expose you to the limitation of a Hebbian learning scheme and show how this limitation can be overcome using the delta rule. For this exercise, you are to set up two different sets of training patterns: one in which all the input patterns form an orthogonal set and the other in which they form a linearly independent, but not orthogonal, set. For both cases, choose the output patterns so that they form an orthogonal set, then arbitrarily assign one of these output patterns to go with each input pattern. In both cases, use only three pattern pairs and make sure that both patterns in each pair are eight elements long. The pattern files you construct in each case should contain three lines formatted like the single line in the *one.pat* file:

```
first     1.0 -1.0 1.0 -1.0 1.0 -1.0 1.0 -1.0     1.0 1.0 -1.0 -1.0 1.0 1.0 -1.0 -1.0
```

We provide sets of patterns that meet these conditions in the two files *ortho.pat* and *li.pat*. However, we want you to make up your own patterns. Save both sets for your use in the exercises in files called *myortho.pat* and *myli.pat*. For each set of patterns, display the patterns in a table, then answer each of the next two questions.

Q.4.2.1.

> Read in the patterns using the "Load New" option in both the Train and Test options, separately. Reset the network (this clears the weights to 0s). Then run one epoch of training using the Hebbian learning rule by pressing the "Run" button. What happens with each pattern? Run three additional epochs of training (one at a time), testing all the patterns after each epoch. What happens? In what ways do things change? In what ways do they stay the same? Why?

Q.4.2.2.

> Turn off Hebb mode in the program by enabling the delta rule under Train options, and try the above experiment again. Make sure to reset the weights before training. Describe the similarities and differences between the results obtained with the various measures (concentrate on *ndp* and *tss*) and explain in terms of the differential characteristics of the Hebbian and delta rule learning schemes.

For the next question, reset your network, and load the pattern set in the file *li.pat* for both training and testing. Run one epoch of training using the Hebb rule, and save the weights, using a command like:

```
liHebbwts = net.pool(3).proj(1).weight
```

Then press *reset* again, and switch to the delta rule. Run one epoch of training at a time, and examine performance at the end of each epoch by testing all patterns.

Q.4.2.3.

> In *li.pat*, one of the input patterns is orthogonal to both of the others, which are partially correlated with each other. When you test the network at the end of one epoch of training, the network exhibits perfect performance on two of the three patterns. Which pattern is not perfectly correct? Explain why the network is not perfectly correct on this pattern and why it is perfectly correct on the other two patterns.

Keep running training epochs using the delta rule until the *tss* measure drops below 0.01. Store the weights in a variable, such as liDeltawts, so that you can display them numerically.

Q.4.2.4.

> Examine and explain the resulting weight matrix, contrasting it with the weight matrix obtained after one cycle of Hebbian learning with the same patterns (these are the weights you saved before). What are the similarities between the two matrices? What are the differences? For one thing, take note of the weight to output unit 1 from input unit 1, and the weight to output unit 8 to input unit 8. These are the same under the Hebb rule, but different under the Delta rule. Why? Make sure you find other differences, and explain them as well. For all of the differences you notice, try to explain rather than just describe the differences.

*Hint.*

> To answer this question fully, you will need to refer to the patterns. Remember that in the Hebb rule, each weight is just the sum of the co-products of corresponding input and output activations, scaled by the learning rate parameter. But this is far from the case with the Delta rule, where weights can compensate for one another, and where such things as a division of labor can occur. You can fully explain the weights learned by the Delta rule, if you take note of the fact that all eight input units contribute to the activation of each of the output units. You can consider each output unit independently, however, since the error measure treats each output unit independently.

As the final exercise in this set, construct a set of input-output pattern pairs that cannot be learned by a delta rule network, referring to the *linear independence requirement* and the text in Section **4.2.3** to help you construct an unlearnable set of patterns. Full credit will be given for sets containing more than 2 patterns, such that with all but one of the patterns, the set can be learned, but with all three, the set cannot be learned.

Q.4.2.5.

> Present your set of patterns, explain why they cannot be learned, and describe what happens when the network tries to learn them, both in terms of the time course of learning and in terms of the weights that result.

*Hint.*

> We provide a set of impossible pattern pairs in the file *imposs.pat*, but, once again, you should construct your own. When you examine what happens during learning, you will probably want to use a small value of the learning rate; this affects the size of the oscillations that you will probably observe in the weights. A learning rate of 0.0125 or less is probably good. Keep running more training epochs until the *tss* at the end of each epoch stabilizes.

## Ex4.3. Learning Central Tendencies

One of the positive features of associator models is their ability to filter out noise in their environments. In this exercise we invite you to explore this aspect of pattern associator networks. For this exercise, you will still be using linear units but with the delta rule and with a relatively small learning rate. You will also be introducing noise into your training patterns.

For this exercise, exit the PDP program and then restart it by typing **ct** at the command prompt (**ct** is for "central tendency"). This file sets the learning rate to 0.0125 and uses the Delta rule. It also sets the noise variable to 0.5.

This means that each element in each input pattern and in each target pattern will have its activation distorted by a random amount uniformly distributed between +0.5 and −0.5.

Then load in a set of patterns (your orthogonal set from Ex. 4.2 or the patterns in *ortho.pat*). Then you can see how well the model can do at pulling out the "signals" from the "noise." The clearest way to see this is by studying the weights themselves and comparing them to the weights acquired with the same patterns without noise added. You can also test with noise turned off; in fact as loaded, noise is turned off for testing, so running a test allows you to see how well the network can do with patterns without noise added.

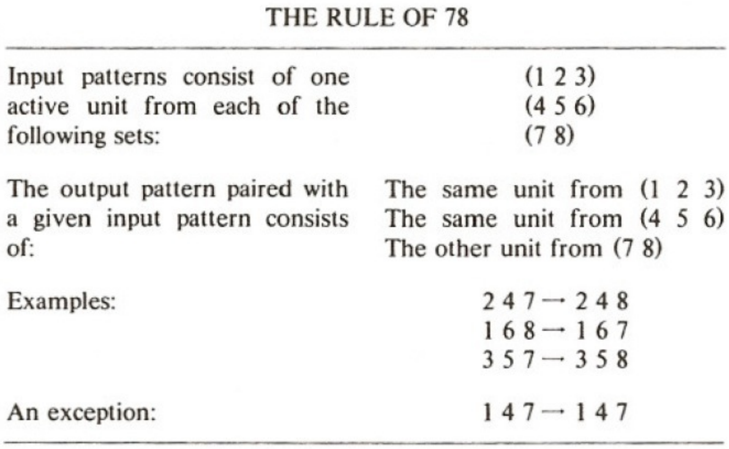
Q.4.3.1.

> Compare learning of the three orthogonal patterns you used in Ex. 4.2 without noise, to the learning that occurs in this exercise, with noise added. Compare the weight matrix acquired after "noiseless" learning with the matrix that evolves given the noisy input-target pairs that occur in the current situation. Run about 60 epochs of training to get an impression of the evolution of the weights through the course of training and compare the results to what happens with errorless training patterns (and a higher learning rate). What effect does changing the learning rate have when there is noise? Try higher and lowers values. You should interleave training and testing, and use up to 1000 epochs when using very low learning rates.
>
> We have provided a pop-up graph that will show how the *tss* changes over time. A new graph is created each time you start training after resetting the network.

*Hint.*

> You may find it useful to rerun the relevant part of Ex. 4.2 (Q. 4.2.2). You can save the weights you obtain in the different runs as before, e.g.
>
> ```
> nonoisewts = pool(3).proj(1).weight;
> ```
>
> For longer runs, remember that you can set *Epochs* in Train options to a number larger than the default value to run more epochs for each press of the "Run" button.

The results of this simulation are relevant to the theoretical analyses described in *PDP:11* and are very similar to those described under "central tendency learning" in *PDP:25*, where the effects of amnesia (taken as a reduction in connection strength) are considered.



THE RULE OF 78

| | |
|---|---|
| Input patterns consist of one active unit from each of the following sets: | (1 2 3)<br>(4 5 6)<br>(7 8) |
| The output pattern paired with a given input pattern consists of: | The same unit from (1 2 3)<br>The same unit from (4 5 6)<br>The other unit from (7 8) |
| Examples: | 2 4 7 → 2 4 8<br>1 6 8 → 1 6 7<br>3 5 7 → 3 5 8 |
| An exception: | 1 4 7 → 1 4 7 |

Figure 4.5: Specification of the Rule of 78. From *PDP:18*, p. 229.

## Ex4.4. Lawful Behavior

We now turn to one of the principle characteristics of pattern associator models that has made us take interest in them: their ability to pick up regularities in a set of input-output pattern pairs. The ability of pattern associator models to do this is illustrated in the past-tense learning model, discussed in *PDP:18*. Here we provide the opportunity to explore this aspect of pattern associator models, using the example discussed in that chapter, namely, the *rule of 78* (see *PDP:18*, pp. 226-234). We briefly review this example here.

The rule of 78 is a simple rule we invented for the sake of illustration. The rule first defines a set of eight-element input patterns. In each input pattern, one of units 1, 2, and 3 must be on; one of units 4, 5, and 6 must be on; and one of units 7 and 8 must be on. For the sake of consistency with *PDP:18*, we adopt the convention for this example only of numbering units starting from 1. The rule of 78 also defines a mapping from input to output patterns. For each input pattern, the output pattern that goes with it is the same as the input pattern, except that if unit 7 is on in the input pattern, unit 8 is on in the output and vice versa. Figure 4.5 shows this rule.

The rule of 78 defines 18 input-output pattern pairs. Eighteen *arbitrary* input-output pattern pairs would exceed the capacity of an eight-by-eight pattern associator, but as we shall see, the patterns that exemplify the rule of 78 can easily be learned by the network.

The version of the pattern associator used for this example follows the assumptions we adopted in *PDP:18* for the past-tense learning model. Input units are binary and are set to 1 or 0 according to the input pattern. The output units are binary, stochastic units and take on activation values of 0 or 1 with probability given by the logistic function:

$$p(act_i = 1) = \frac{1}{1 + e^{-net_i/T}} \tag{4.26}$$

where $T$ is equivalent to the *Temp* parameter in Train and Test options. Note that, although this function is the same as for the Boltzmann machine, the calculation of the output is only done once, as in other versions of the pattern associator; there is no annealing, so *Temp* is just a scaling factor.

Learning occurs according to the delta rule, which in this case is equivalent to the perceptron convergence procedure because the units are binary. Thus, when an output unit should be on (target is 1) but is not (activation is 0), an increment of size *lrate* is added to the weight coming into that unit from each input unit that is on. When an output unit should be off (target is 0) but is not (activation is 1), an increment of size *lrate* is subtracted from the weight coming into that unit from each input unit that is on.

For this example, we follow *PDP:18* and use *Temp* of 1 and a learning rate of .05. (The simulations that you will do here will not conform to the example in *PDP:18* in all details, since in that example an approximation to the logistic function was used. The basic features of the results are the same, however.)

To run this example, exit the PDP system if running, and then enter

```
seventy_eight
```

at the command prompt. This will read in the appropriate network specification file (in *8X8.net*) and the 18 patterns that exemplify the rule of 78, then display these on the screen to the right of the weight matrix. Since the units are binary, there is only a single digit of precision for both the input, output, and target units.

You should now be ready to run the exercise. The variable *Epochs* is initialized to 10, so if you press the Run button, 10 epochs of training will be run. We recommend using *ptrain* because it does not result in a consistent bias in the weights favoring the patterns later in the pattern list. If you want to see the screen updated once per pattern, set the Update After field in the train panel to be “pattern” instead of “epoch.” If “pattern” is selected, the screen is updated once per pattern after the weights have been adjusted, so you should see the weights and the input, output, and target bits changing. The *pss* and *tss* (which in this case indicate the number of incorrect output bits) will also be displayed once per pattern.

Q.4.4.1.

> At the end of the 10th epoch, the *tss* should be in the vicinity of 30, or about 1.5 errors per pattern. Given the values of the weights and the fact that Temp is set to 1, calculate the net input to the last output unit for the first two input patterns, and calculate the approximate probability that this last output unit will receive the correct activation in each of these two patterns. MATLAB will

calculate this probability if you enter it into the logistic function yourself:

```
p = 1/(1+exp(-net.pool(3).netinput(8)))
```

At this point you should be able to see the solution to the rule of 78 patterns emerging. Generally, there are large positive weights between input units and corresponding output units, with unit 7 exciting unit 8 and unit 8 exciting unit 7. You'll also see rather large inhibitory weights from each input unit to each other unit within the same subgroup (i.e., 1, 2, and 3; 4, 5, and 6; and 7 and 8). Run another 40 or so epochs, and a subtler pattern will begin to emerge.

Q.4.4.2.

> Generally there will be slightly negative weights from input units to output units in other subgroups. See if you can understand why this happens. Note that this does not happen reliably for weights coming into output units 7 and 8. Your explanation should explain this too.

At this point, you have watched a simple PDP network learn to behave in accordance with a simple rule, using a simple, local learning scheme; that is, it adjusts the strength of each connection in response to its errors on each particular learning experience, and the result is a system that exhibits lawful behavior in the sense that it conforms to the rule.

For the next part of the exercise, you can explore the way in which this kind of pattern associator model captures the three-stage learning phenomenon exhibited by young children learning the past tense in the course of learning English as their first language. To briefly summarize this phenomenon: Early on, children know only a few words in the past tense. Many of these words happen to be exceptions, but at this point children tend to get these words correct. Later in development, children begin to use a much larger number of words in the past tense, and these are predominantly regular. At this stage, they tend to overregularize exceptions. Gradually, over the course of many years, these exceptions become less frequent, but adults have been known to say things like ringed or taked, and lower-frequency exceptions tend to lose their exceptionality (i.e., to become regularized) over time.

The 78 model can capture this pattern of results; it is interesting to see it do this and understand how and why this happens. For this part of the exercise, you will want to reset the weights, and read in the file *hf.pat*, which contains a exception pattern (147 ⟶ 147) and one regular pattern (258 ⟶ 257). If we imagine that the early experience of the child consists mostly of exposure to high-frequency words, a large fraction of which are irregular (8 of the 10 most frequent verbs are irregular), this approximates the early experience the child might have with regular and irregular past-tense forms. If you run 30 epochs of training using *ptrain* with these two patterns, you will see a set of weights that allows the model to often set each output bit correctly, but not reliably. At this point, you can read in the file *all.pat*, which contains these two pattern pairs, plus all of the other pairs that are consistent with the rule of 78. This file differs from the *78.pat* file only in that the input pattern *147* is associated with the "exceptional" output pattern *147* instead of what would be the "regular" corresponding pattern *148*. Save the weights that resulted from learning *hf.pat*. Then read in *all.pat* and run 10 more epochs.

Q.4.4.3.

> Given the weights that you see at this point, what is the network's most probable response to *147*? Can you explain why the network has lost the ability to produce *147* as its response to this input pattern? What has happened to the weights that were previously involved in producing *147* from *147*?

One way to think about what has happened in learning the *all.pat* stimuli is that the 17 regular patterns are driving the weights in one direction and the single exception pattern is fighting a lonely battle to try to drive the weights in a different direction, at least with respect to the activation of units 7 and 8. Since eight of the input patterns have unit 7 on and "want" output unit 8 to be on and unit 7 to be off and only one input pattern has input unit 7 on and wants output unit 7 on and output unit 8 off, it is hardly a fair fight.

If you run more epochs (upwards of 300), though, you will find that the network eventually finds a compromise solution that satisfies all of the patterns.

Q.4.4.4.

> Although it takes a fair number of epochs, run the model until it finds a set of weights that gets each output unit correct about 90% of the time for each input pattern (90% correct corresponds to a net input of about 2 or so for units that should be on and −2 for units that should be off). Explain why it takes so long to get to this point.

## Ex4.5. Learning quasi-regular exceptions

Pinker and Ullman (2002) argue for a two-system model, in which a connectionist like system deals with exceptions, but there is a separate "procedural" system for rules. Consider the response to this position contained in the short reply to Pinker and Ullman (2002) by McClelland and Patterson (2002).

Q.4.5.1.

> Express the position taken by McClelland and Patterson. Now, consider whether the seventy-eight model is sensitive to (and benefits from) quasi-regularity in exceptions. Compare learning of quasi-regular vs. truly arbitrary exceptions to the rule of 78 by creating two new training sets from the fully regular *seventy-eight.pat* training set. Create a quasi-regular training set by modifying the output patterns of 2-3 of the items so that they are quasi-regular, as defined by McClelland and Patterson. Create a second training set with 2-3 arbitrary exceptions by assigning completely arbitrary output patterns to the same 2-3 input patterns. Carry out training experiments with both training sets. Report differences in learnability, training time, and pattern of performance for these two different sets of items, and discuss whether (and how) your results support the idea that the PDP model explains why exceptions tend to be quasi-regular rather than completely arbitrary.

### 4.6.1 Further Suggestions for Exercises

There are other exercises for further exploration. In the 78 exercise just described, there was only one exception pattern, and when vocabulary size increased, the ratio of regular to exception patterns increased from 1:1 to 17:1. Pinker and Prince (1988) have shown that, in fact, as vocabulary size increases, the ratio of regular to exception verbs stays roughly constant at 1:1. One interesting exercise is to set up an analog of this situation. Start training the network with one regular and one exception pattern, then increase the "vocabulary" by introducing new regular patterns and new exceptions. Note that each exception should be idiosyncratic; if all the exceptions were consistent with each other, they would simply exemplify a different rule. You might try an exercise of this form, setting up your own correspondence rules, your own exceptions, and your own regime for training.

You can also explore other variants of the pattern associator with other kinds of learning problems. One thing you can do easily is see whether the model can learn to associate each of the individuals from the Jets and Sharks example in Chapter 2 with the appropriate gang (relying only on their properties, not their names; the files *jets.tem*, *jets.net*, and *jets.pat* are available for this purpose). Also, you can play with the continuous sigmoid (or logistic) activation function.

# Chapter 5

# Training Hidden Units with Back Propagation

In this chapter, we introduce the back propagation learning procedure for learning internal representations. We begin by describing the history of the ideas and problems that make clear the need for back propagation. We then describe the procedure, focusing on the goal of helping the student gain a clear understanding of gradient descent learning and how it is used in training PDP networks. The exercises are constructed to allow the reader to explore the basic features of the back propagation paradigm. At the end of the chapter, there is a separate section on extensions of the basic paradigm, including three variants we call *cascaded* back propagation networks, *recurrent* networks, and *sequential* networks. Exercises are provided for each type of extension.

## 5.1 BACKGROUND

The pattern associator described in the previous chapter has been known since the late 1950s, when variants of what we have called the delta rule were first proposed. In one version, in which output units were linear threshold units, it was known as the perceptron (cf. Rosenblatt, 1959, 1962). In another version, in which the output units were purely linear, it was known as the LMS or least mean square associator (cf. Widrow and Hoff, 1960). Important theorems were proved about both of these versions. In the case of the perceptron, there was the so-called perceptron convergence theorem. In this theorem, the major paradigm is pattern classification. There is a set of binary input vectors, each of which can be said to belong to one of two classes. The system is to learn a set of connection strengths and a threshold value so that it can correctly classify each of the input vectors. The basic structure of the perceptron is illustrated in Figure 5.1. The perceptron learning procedure is the following: An input vector is presented to the system (i.e., the input units are given an activation of 1 if the corresponding value of the input vector is 1 and are given 0 otherwise).

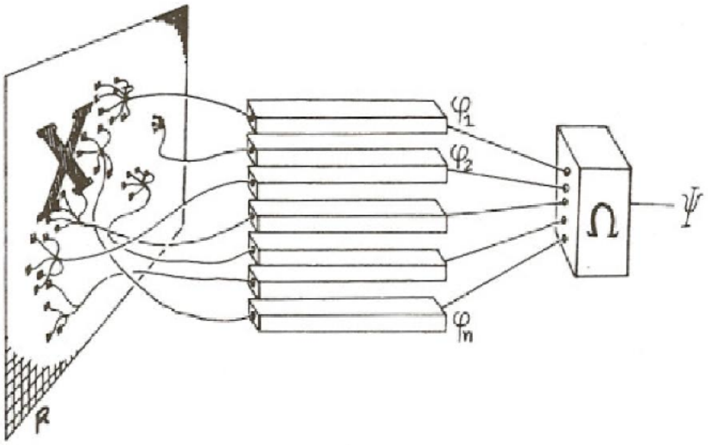


Figure 5.1: The one-layer perceptron analyzed by Minsky and Papert. (From *Perceptrons* by M. L Minsky and S. Papert, 1969, Cambridge, MA: MIT Press. Copyright 1969 by MIT Press. Reprinted by permission.)

The net input to the output unit is computed: $net = \sum_i w_i i_i$. If $net$ is greater than the threshold $\theta$, the unit is turned on, otherwise it is turned off. Then the response is compared with the actual category of the input vector. If the vector was correctly categorized, then no change is made to the weights. If, however, the output turns on when the input vector is in category 0, then the weights and thresholds are modified as follows: The threshold is incremented by 1 (to make it less likely that the output unit will come on if the same vector were presented again). If input $i_i$ is 0, no change is made in the weight $W_i$ (that weight could not have contributed to its having turned on). However, if $i_i = 1$, then $W_i$ is decremented by 1. In this way, the output will not be as likely to turn on the next time this input vector is presented. On the other hand, if the output unit does not come on when it is supposed to, the opposite changes are made. That is, the threshold is decremented, and those weights connecting the output units to input units that are on are incremented.

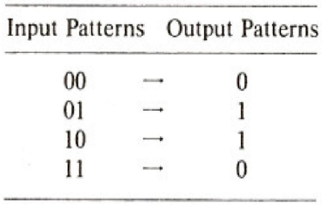
Mathematically, this amounts to the following: The output, $o$, is given by

$$\begin{aligned} o &= 1 \text{ if } net < \theta \\ o &= 0 \;\; otherwise \end{aligned}$$

The change in the threshold, $\Delta\theta$, is given by

$$\Delta\theta = -(t_p - o_p) = -\delta_p$$

where $p$ indexes the particular pattern being tested, $tp$ is the target value indicating the correct classification of that input pattern, and $\delta_p$ is the difference



| Input Patterns | | Output Patterns |
|---|---|---|
| 00 | → | 0 |
| 01 | → | 1 |
| 10 | → | 1 |
| 11 | → | 0 |

Figure 5.2: (The XOR Problem. From *PDP:8,* p. 319).

between the target and the actual output of the network. Finally, the changes in the weights, $\Delta w_i$. are given by

$$\Delta w_i = (t_p - o_p) i_{ip} = \delta_p i_{ip}$$

The remarkable thing about this procedure is that, in spite of its simplicity, such a system is guaranteed to find a set of weights that correctly classifies the input vectors *if such a set of weights exists.* Moreover, since the learning procedure can be applied independently to each of a set of output units, the perceptron learning procedure will find the appropriate mapping from a set of input vectors onto a set of output vectors *if such a mapping exists.* Unfortunately, as indicated in Chapter 4, such a mapping does not always exist, and this is the major problem for the perceptron learning procedure.

In their famous book *Perceptrons*, Minsky and Papert (1969) document the limitations of the perceptron. The simplest example of a function that cannot be computed by the perceptron is the exclusive-or (XOR), illustrated in Figure 5.1. It should be clear enough why this problem is impossible. In order for a perceptron to solve this problem, the following four inequalities must be satisfied:

$$\begin{aligned}
0 \times w_1 + 0 \times w_2 < \theta &\rightarrow 0 < \theta \\
0 \times w_1 + 1 \times w_2 > \theta &\rightarrow w_1 > \theta \\
1 \times w_1 + 0 \times w_2 > \theta &\rightarrow w_2 > \theta \\
1 \times w_1 + 1 \times w_2 < \theta &\rightarrow w_1 + w_2 < \theta
\end{aligned}$$

Obviously, we can't have both $w_1$ and $w_2$ greater than $\theta$ while their sum, $w_1 + w_2$, is less than $\theta$. There is a simple geometric interpretation of the class of problems that can be solved by a perceptron: It is the class of *linearly separable* functions. This can easily be illustrated for two dimensional problems such as XOR. Figure 5.1.1 shows a simple network with two inputs and a single output and illustrates three two-dimensional functions: the AND, the OR, and the XOR. The first two can be computed by the network; the third cannot. In these geometrical representations, the input patterns are represented as coordinates

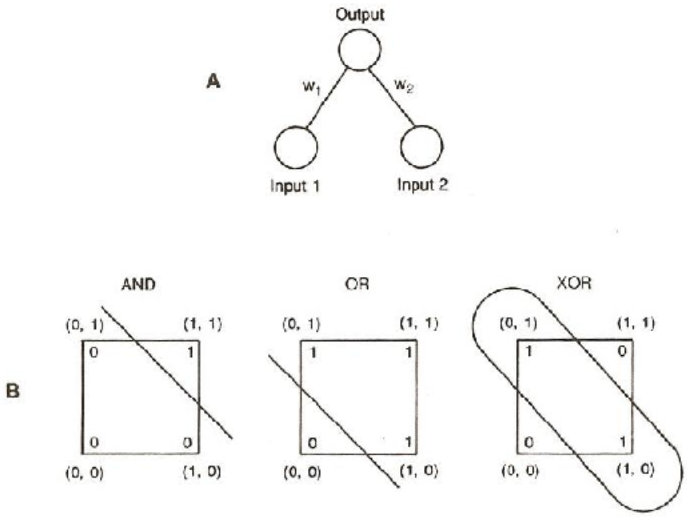
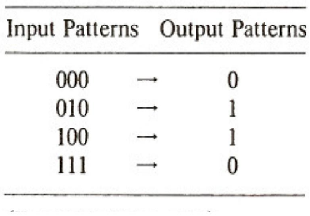


Figure 5.3: A. A simple network that can solve the AND and OR problems but cannot solve the XOR problem. B. Geometric representations of these problems. See test for explanation.

in space. In the case of a binary two-dimensional problem like XOR, these coordinates constitute the vertices of a square. The pattern 00 is represented at the lower left of the square, the pattern 10 as the lower right, and so on. The function to be computed is then represented by labeling each vertex with a 1 or 0 depending on which class the corresponding input pattern belongs to. The perceptron can solve any function in which a single line can be drawn through the space such that all of those labeled ”0” are on one side of the line and those labeled ”1” are on the other side. This can easily be done for AND and OR, but not for XOR. The line corresponds to the equation $i_1w_1 + i_2w_2 = \theta$. In three dimensions there is a plane, $i_1w_1 + i_2w_2 + i_3w_3 = \theta$, that corresponds to the line. In higher dimensions there is a corresponding hyperplane, $\sum_i w_ii_i = \theta$. All functions for which there exists such a plane are called *linearly separable*.

Now consider the function in Figure 5.4 and shown graphically in Figure 5.5. This is a three-dimensional problem in which the first two dimensions are identical to the XOR and the third dimension is the AND of the first two dimensions. (That is, the third dimension is 1 whenever both of the first two dimensions are 1, otherwise it is 0). Figure 5.5 shows how this problem can be represented in three dimensions. The figure also shows how the addition of the third dimension allows a plane to separate the patterns classified in category 0 from those in category 1. Thus, we see that the XOR is not solvable in two dimensions, but



| Input Patterns | | Output Patterns |
|---|---|---|
| 000 | → | 0 |
| 010 | → | 1 |
| 100 | → | 1 |
| 111 | → | 0 |

Figure 5.4: Adding an extra input makes it possible to solve the XOR problem. (From *PDP:8,* p. 319.)

if we add the appropriate third dimension, that is, the appropriate *new feature,* the problem *is* solvable. Moreover, as indicated in Figure 5.6, if you allow a multilayered perceptron, it is possible to take the original two-dimensional problem and convert it into the appropriate three-dimensional problem so it can be solved. Indeed, as Minsky and Papert knew, it is always possible to convert any unsolvable problem into a solvable one in a multilayer perceptron. In the more general case of multilayer networks, we categorize units into three classes: *input units,* which receive the input patterns directly; *output units,* which have associated *teaching* or *target* inputs; and *hidden units,* which neither receive inputs directly nor are given direct feedback. This is the stock of units from which new features and new internal representations can be created. The problem is to know which new features are required to solve the problem at hand. In short, we must be able to learn intermediate layers. The question is, how? The original perceptron learning procedure does not apply to more than one layer. Minsky and Papert believed that no such general procedure could be found. To examine how such a procedure can be developed it is useful to consider the other major one-layer learning system of the 1950s and early 1960s, namely, the *least-mean-square (LMS)* learning procedure of Widrow and Hoff (1960).

### 5.1.1 Minimizing Mean Squared Error

The LMS procedure makes use of the delta rule for adjusting connection weights; the perceptron convergence procedure is very similar, differing only in that linear threshold units are used instead of units with continuous-valued outputs. We use the term *LMS procedure* here to stress the fact that this family of learning rules may be viewed as minimizing a measure of the error in their performance. The LMS procedure cannot be directly applied when the output units are linear threshold units (like the perceptron). It has been applied most often with purely linear output units. In this case the activation of an output unit, $o_i$, is simply given by

$$o_i = \sum_j w_{ij} i_j + bias_i$$

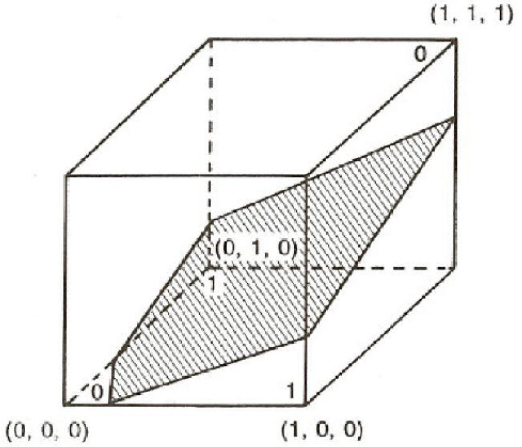


Figure 5.5: The three-dimensional solution of the XOR problem.

Note the introduction of the *bias* term which serves the same function as the threshold $\theta$ in the Perceptron. Providing a bias equal to $-\theta$ and setting the threshold to 0 is equivalent to having a threshold of $\theta$. The bias is also equivalent to a weight to the output unit from an input unit that is always on.

The error measure being minimised by the LMS procedure is the summed squared error. That is, the total error, $E$, is defined to be

$$E = \sum_p E_p = \sum_p \sum_i (t_{ip} - o_{ip})^2$$

where the index $p$ ranges over the set of input patterns, $i$ ranges over the set of output units, and $E_p$ represents the error on pattern $p$. The variable $t_{ip}$ is the desired output, or *target,* for the $i$th output unit when the $p$th pattern has been presented, and $o_{ip}$ is the actual output of the $i$th output unit when pattern $p$ has been presented. The object is to find a set of weights that minimizes this function. It is useful to consider how the error varies as a function of any given weight in the system. Figure 5.7 illustrates the nature of this dependence. In the case of the simple single-layered linear system, we always get a smooth error function such as the one shown in the figure. The LMS procedure finds the values of all of the weights that minimize this function using a method called *gradient descent.* That is, after each pattern has been presented, the error on that pattern is computed and each weight is moved "down" the error gradient toward its minimum value for that pattern. Since we cannot map out the entire error function on each pattern presentation, we must find a simple procedure for determining, for each weight, how much to increase or decrease each weight. The idea of gradient descent is to make a change in the weight proportional to the negative of the derivative of the error, as measured on the current pattern,

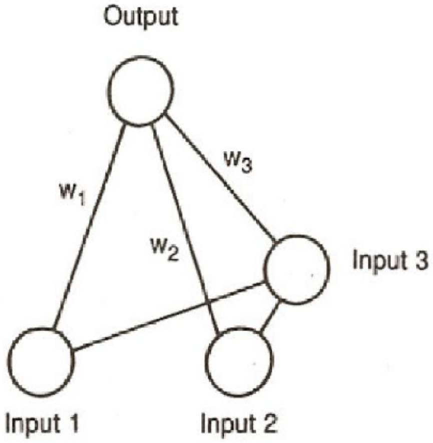


Figure 5.6: A multilayer network that converts the two-dimensional three-dimensional XOR problem into a three-dimensional linearly separable problem.

with respect to each weight.[1] Thus the learning rule becomes

$$\Delta w_{ij} = -k \frac{\partial E_p}{\partial w_{ij}}$$

where $k$ is the constant of proportionality. Interestingly, carrying out the derivative of the error measure in Equation 1 we get

$$\Delta w_{ij} = \epsilon \delta_{ip} i_{ip}$$

where $\epsilon = 2k$ and $\delta_{ip} = (t_{ip} - o_{ip})$ is the difference between the target for unit $i$ on pattern $p$ and the actual output produced by the network. This is exactly the delta learning rule described in Equation 15 from Chapter 4. It should also be noted that this rule is essentially the same as that for the perceptron. In the perceptron the learning rate was 1 (i.e., we made unit changes in the weights) and the units were binary, but the rule itself is the same: the weights are changed proportionally to the difference between target and output times the input. If we change each weight according to this rule, each weight is moved toward its own minimum and we think of the system as moving downhill in *weight-space* until it reaches its minimum error value. When all of the weights have reached their minimum points, the system has reached equilibrium. If the system is able to solve the problem entirely, the system will reach zero error

[1] It should be clear from Figure 5.7 why we want the negation of the derivative. If the weight is above the minimum value, the slope at that point is *positive* and we want to *decrease* the weight; thus when the slope is positive we add a negative amount to the weight. On the other hand, if the weight is too small, the error curve has a negative slope at that point, so we want to add a positive amount to the weight.

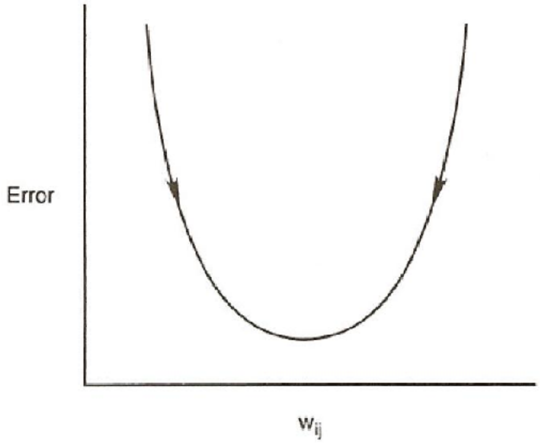


Figure 5.7: Typical curve showing the relationship between overall error and changes in a single weight in the network.

and the weights will no longer be modified. If the network is unable to get the problem exactly right, it will find a set of weights that produces as small an error as possible.

In order to get a fuller understanding of this process it is useful to carefully consider the entire error space rather than a one-dimensional slice. In general this is very difficult to do because of the difficulty of depicting and visualizing high-dimensional spaces. However, we can usefully go from one to two dimensions by considering a network with exactly two weights. Consider, as an example, a linear network with two input units and one output unit with the task of finding a set of weights that comes as close as possible to performing the function OR. Assume the network has just two weights and no bias terms like the network in Figure 5.1.1A. We can then give some idea of the shape of the space by making a contour map of the error surface. Figure 5.8 shows the contour map. In this case the space is shaped like a kind of oblong bowl. It is relatively flat on the bottom and rises sharply on the sides. Each equal error contour is elliptically shaped. The arrows around the ellipses represent the derivatives of the two weights at those points and thus represent the directions and magnitudes of weight changes at each point on the error surface. The changes are relatively large where the sides of the bowl are relatively steep and become smaller and smaller as we move into the central minimum. The long, curved arrow represents a typical trajectory in weight-space from a starting point far from the minimum down to the actual minimum in the space. The weights trace a curved trajectory following the arrows and crossing the contour lines at right angles.

The figure illustrates an important aspect of gradient descent learning. This

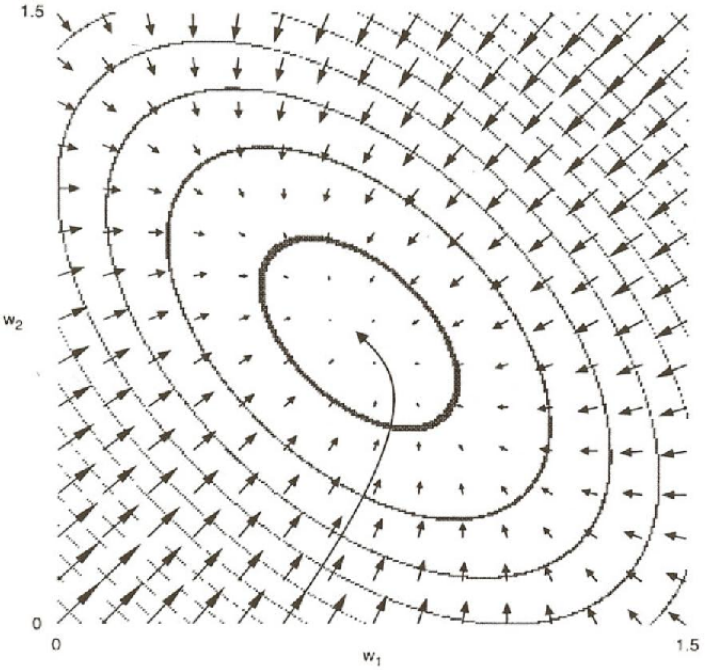


Figure 5.8: A contour map illustrating the error surface with respect to the two weights $w_1$ and $w_2$ for the OR problem in a linear network with two weights and no bias term. Note that the OR problem cannot be solved perfectly in a linear system. The minimum sum squared error over the four input-output pairs occurs when $w_1 = w_2 = 0.75$. (The input-output pairs are $00 - 0$,$01 - 1$,$10 - 1$, and $11 - 1$.)

is the fact that gradient descent involves making larger changes to parameters that will have the biggest effect on the measure being minimized. In this case, the LMS procedure makes changes to the weights proportional to the effect they will have on the summed squared error. The resulting total change to the weights is a vector that points in the direction in which the error drops most steeply.

### 5.1.2 The Back Propagation Rule

Although this simple linear pattern associator is a useful model for understanding the dynamics of gradient descent learning, it is not useful for solving problems such as the XOR problem mentioned above. As pointed out in *PDP:2,* linear systems cannot compute more in multiple layers than they can in a single layer. The basic idea of the back propagation method of learning is to combine a nonlinear perceptron-like system capable of making decisions with the objective error function of LMS and gradient descent. To do this, we must be able to readily compute the derivative of the error function with respect to *any weight in the network* and then change that weight according to the rule

$$\Delta w_{ij} = -k \frac{\partial E_p}{\partial w_{ij}}$$

How can this derivative be computed? First, it is necessary to use a differentiable output function, rather than the threshold function used in the perceptron convergence procedure. A common choice, and one that allows us to relate back-propagation learning to probabilistic inference, is to use the *logistic* function $f(net_i) = \frac{1}{1+exp(-net_i)}$. Given a choice of $f$, we can then determine the partial derivative of the Error with respect to a weight coming to an output unit $i$ from a unit $j$ that projects to it.

We can use the chain rule to compute the partial derivative of the error with respect to our particular weight $w_{ij}$:

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial a_{ip}} \frac{\partial a_{ip}}{\partial net_{ip}} \frac{\partial net_{ip}}{\partial w_{ij}}$$

Now, $E_p = \sum_i (t_{ip} - a_{ip})^2$, so that $\frac{\partial E_p}{\partial a_{ip}}$ is equal to $-2(t_{ip} - a_{ip})$. $a_i = f(net_i)$,[2] and, so that we can leave $f$ unspecified for the moment, we write $f'(net_i)$ to represent its derivative evaluated at $net_i$. Finally, $net_i = \sum_j a_j w_{ij}$, and its partial derivative of the net input with respect to $w_{ij}$ is just $a_j$. In sum, then we have

$$-\frac{\partial E_p}{\partial w_{ij}} = 2(t_{ip} - a_{ip}) f'(net_{ip}) a_{jp}$$

Let us use $\delta_{ip}$ to represent $(t_{ip} - a_{ip}) f'(net_{ip})$. $\delta_{ip}$ is proportional to (minus) the partial derivative of the error with respect to the net input to unit $i$ in pattern $p$, $\frac{\partial E}{\partial net_{ip}}$. Substituting this into the above expression, we can now write:

$$-\frac{\partial E_p}{\partial w_{ij}} \propto \delta_{ip} a_{jp}$$

This generalizes the delta rule from the LMS procedure to the case where there is a non-linearity applied to the output units, with the $\delta$ terms now defined so as to take this non-linearity into account.

Now let us consider a weight that projects from an input unit $k$ to a hidden unit $j$, which in turn projects to an output unit, $i$ in a very simple network consisting of only one unit at each of these three layers (see Figure 5.1.2). We can ask, what is the partial derivative of the error on the output unit $i$ with respect to a change in the weight $w_{jk}$ to the hidden unit from the input unit? It may be helpful to talk yourself informally through the series of effects changing

[2] In the networks we will be considering in this chapter, the output of a unit is equal to its activation. We use the symbol $a$ to designate this variable. This symbol can be used for any unit, be it an input unit, an output unit, or a hidden unit.

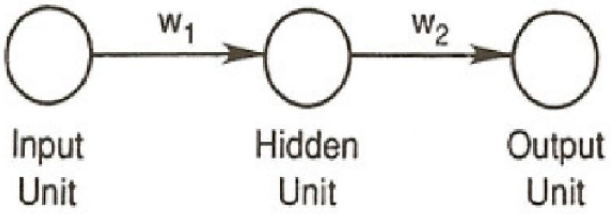


Figure 5.9: A 1:1:1 network, consisting of one input unit, one hidden unit, and one output unit. In the text discussing the chain of effects of changing the weight from the input unit to the hidden unit on the error at the output unit, the index $i$ is used for the output unit, $j$ for the hidden unit $j$, and $k$ for the input unit.

such a weight would have on the error in this case. It should be obvious that if you increase the weight to the hidden unit from the input unit, that will increase the net input to the hidden unit $j$ by an amount that depends on the activation of the input unit $k$. If the input unit were inactive, the change in the weight would have no effect; the stronger the activation of the input unit, the stronger the effect of changing the weight on the net input to the hidden unit. This change, you should also see, will in turn increase the activation of the hidden unit; the amoung of the increase will depend on the slope (derivative) of the unit's activation function evaluated at the current value of its net input. This change in the activation with then affect the net input to the output unit $i$ by an amount depending on the current value of the weight to unit $i$ from unit $j$. This change in the net input to unit $i$ will then affect the activation of unit $i$ by an amount proportional to the derivative of its activation function evaluated at the current value of its net input. This change in the activation of the output unit will then affect the error by an amount proportional to the difference between the target and the current activation of the output unit.

The above is an intuitive account of corresponding to the series of factors you get when you apply the chain rule to unpack the partial derivative of the error at the output unit with respect to a change in the weight to the hidden unit from the input unit. Applying this to the case of the error on pattern $p$, we would write

$$\frac{\partial E_p}{\partial w_{jk}} = \frac{\partial E_p}{\partial a_{ip}} \frac{\partial a_{ip}}{\partial net_i} \frac{\partial net_{ip}}{\partial a_{jp}} \frac{\partial a_{ip}}{\partial net_j} \frac{\partial net_{jp}}{\partial w_{jk}}$$

The factors in the chain are given in the reverse order from the verbal description above since this is how they will actually be calculated using back propagation. The first two factors on the right correspond to the last two links of the chain described above, and are equal to the $\delta$ term for output unit $i$ as previously discussed. The third factor is equal to the weight to output unit $i$ from hidden unit $j$; and the fourth factor corresponds to the derivative of the activation function of the hidden unit, evaluated at its net input given the current pattern $p$, $f'(net_{jp})$. Taking these four factors together they correspond to (minus) the partial derivative of the error at output unit $i$ with respect to the net input to hidden unit $j$.

Now, if there is more than one output unit, the partial derivative of the error across all of the output units is just equal to the sum of the partial derivatives of the error with respect to each of the output units:

$$\delta_{jp} = f'(net_{jp}) \sum_i w_{ij} \delta_{ip}. \qquad \text{BP Equation}$$

The equation above is the core of the back propagation process and we call it the *BP Equation* for future reference.

Because $\frac{\partial net_{jp}}{\partial w_{jk}}$ equals $a_{kp}$, the partial derivative of the error with respect to the weight then becomes:

$$-\frac{\partial E_p}{\partial w_{jk}} = \delta_{jp} a_{kp} \ .$$

Although we have not demonstrated it here, it is easy to show that, with more layers, the correct $\delta$ term for each unit $j$ in a given layer of a feed-forward network is always equal to the derivative of the activation function of the unit evaluated at the current value of its net input, times the sum over the forward connections from that unit of the product of the weight on each forward connection times the delta term at the receiving end of that connection; i.e., the $\delta$ terms for all layers are determined by applying the BP Equation.

Thus, once delta terms have been computed at the output layers of a feed-forward network, the BP equation can be used iteratively to calculate $\delta$ terms backward across many layers of weights, and to specify how each weight should be changed to perform gradient decent. Thus we now have a generalized version of the delta rule that specifies a procedure for changing all the weights in all laters of a feed forward network: If we adjust the weight to each output unit $i$ from each unit $j$ projecting to it, by an amount proportional to $\delta_{ip} a_{jp}$, where $\delta_j$ is defined recursively as discussed above, we will be performing gradient descent in E: We will be adjusting each weight in proportion to (minus) the effect that its adjustment would have on the error.

The application of the back propagation rule, then, involves two phases: During the first phase the input is presented and propagated forward through the network to compute the output value $a_{ip}$ for each unit. This output is then compared with the target, and scaled by the derivative of the activation function, resulting in a $\delta$ term for each output unit. The second phase involves a backward pass through the network (analogous to the initial forward pass) during which the $\delta$ term is computed for each unit in the network. This second, backward pass allows the recursive computation of $\delta$ as indicated above. Once these two phases are complete, we can compute, for each weight, the product of the $\delta$ term associated with the unit it projects to times the activation of the unit it projects from. Henceforth we will call this product the *weight error derivative* since it is proportional to (minus) the derivative of the error with respect to the weight. As will be discussed later, these weight error derivatives can then be used to compute actual weight changes on a pattern-by-pattern basis, or they may be accumulated over the ensemble of patterns with the accumulated sum of its weight error derivatives then being applied to each of the weights.

*Adjusting bias weights.* Of course, the generalized delta rule can also be used to learn biases, which we treat as weights from a special "bias unit" that is always on. A bias weight can project from this unit to any unit in the network, and can be adjusted like any other weight, with the further stipulation that the activation of the sending unit in this case is always fixed at 1.

*The activation function.* As stated above, the derivation of the back propagation learning rule requires that the derivative of the activation function, $f'(net_i)$ exists. It is interesting to note that the linear threshold function, on which the perceptron is based, is discontinuous and hence will not suffice for back propagation. Similarly, since a network with linear units achieves no advantage from hidden units, a linear activation function will not suffice either. Thus, we need a continuous, nonlinear activation function. In most of our work on back propagation and in the program presented in this chapter, we have used the *logistic* activation function.

In order to apply our learning rule, we need to know the derivative of this function with respect to its net input. It is easy to show that this derivative is equal to $a_{ip}(1 - a_{ip})$. This expression can simply be substituted for $f'(net)$ in the derivations above.

It should be noted that $a_{ip}(1 - a_{ip})$ reaches its maximum when $a_{ip} = 0.5$ and goes to 0 as $a_{ip}$ approaches 0 or 1 (see Figure 5.1.2). Since the amount of change in a given weight is proportional to this derivative, weights will be changed most for those units that are near their midrange and, in some sense, not yet committed to being either on or off. This feature can sometimes lead to problems for backpropagation learning, and the problem can be especially serious at the output layer. If the weights in a network at some point during learning are such that a unit that should be on is completely off (or a unit that should be off is completely on) the error at that unit is large but paradoxically the delta term at that unit is very small, and so no error signal is propagated back through the network to correct the problem.

*An improved error measure.* There are various ways around the problem

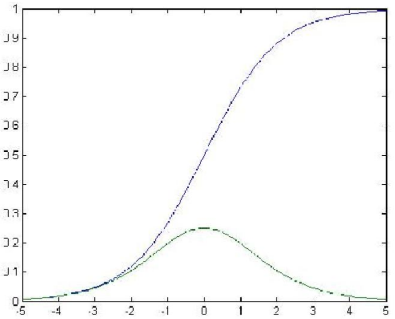


Figure 5.10: The logistic function and its derivative.

just noted above. One is to simply leave the $f'(net_i)$ term out of the calculation of delta terms at the output units. In practice, this solves the problem, but it seems like a bit of a hack.

Interestingly, however, if the error measure E is replaced by a different measure, called the 'cross-entropy' error, here called $CE$, we obtain an elegant result. The cross-entropy error for pattern $p$ is defined as

$$CE_p \;=\; -\sum_i [t_{ip} \log(a_{ip}) + (1 - t_{ip}) \log(1 - a_{ip})]$$

If the target value $t_{ip}$ is thought of as a binary random variable having value one with probability $p_{ip}$, and the activation of the output unit $a_{ip}$ is construed as representing the network's estimate of that probability, the cross-entropy measure corresponds to the negative logarithm of the probability of the observed target values, given the current estimates of the $p_{ip}$'s. Minimizing the cross-entropy error corresponds to maximizing the probability of the observed target values. The maximum is reached when for all $i$ and $p$, $a_{ip} = p_{ip}$.

Now very neatly, it turns out that the derivative of $CE_p$ with respect to $a_{ip}$ is

$$-\left[\frac{t_{ip}}{a_{ip}} - \frac{(1 - t_{ip})}{(1 - a_{ip})}\right].$$

When this is multiplied times the derivative of the logistic function evaluated at the net input, $a_{ip}(1 - a_{ip})$, to obtain the corresponding $\delta$ term $\delta_{ip}$, several things cancel out and we are left with

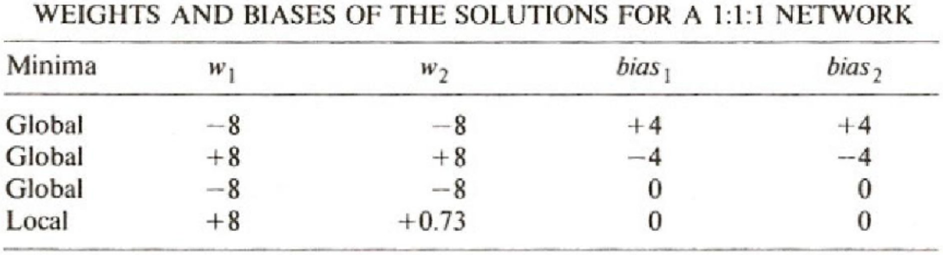
$$\delta_{ip} = t_{ip} - a_{ip}.$$

This is the same expression for the $\delta$ term we would get using the standard sum squared error measure E, if we simply ignored the derivative of the activation function! Because using cross entropy error seems more appropriate than summed squared error in many cases and also because it often works better, we provide the option of using the cross entropy error in the **pdptool** back propagation simulator.

Even using cross-entropy instead of sum squared error, it sometimes happens that hidden units have strong learned input weights that 'pin' their activation against 1 or 0, and in that case it becomes effectively impossible to propagate error back through these units. Different solutions to this problem have been proposed. One is to use a small amount of weight decay to prevent weights from growing too large. Another is to add a small constant to the derivative of the activation function of the hidden unit. This latter method works well, but is often considered a hack, and so is not implemented in the **pdptool** software. Weight decay is available in the software, however, and is described below.

*Local minima.* Like the simpler LMS learning paradigm, back propagation is a gradient descent procedure. Essentially, the system will follow the contour of the error surface, always moving downhill in the direction of steepest descent. This is no particular problem for the single-layer linear model. These systems always have bowl-shaped error surfaces. However, in multilayer networks there is the possibility of rather more complex surfaces with many minima. Some of the minima constitute complete solutions to the error minimization problem, in the sense at these minima the system has reached a completely errorless state. All such minima are *global* minima. However, it is possible for there to be some residual errror at the bottom of some of the minima. In this case, a gradient descent method may not find the *best* possible solution to the problem at hand.

Part of the study of back propagation networks and learning involves a study of how frequently and under what conditions local minima occur. In networks with many hidden units, local minima seem quite rare. However with few hidden units, local minima can occur. The simple 1:1:1 network shown in Figure 5.1.2 can be used to demonstate this phenomenon. The problem posed to this network is to copy the value of the input unit to the output unit. There are two basic ways in which the network can solve the problem. It can have positive biases on the hidden unit and on the output unit and large negative connections from the input unit to the hidden unit and from the hidden unit to the output unit, or it can have large negative biases on the two units and large positive weights from the input unit to the hidden unit and from the hidden unit to the output unit. These solutions are illustrated in Figure 5.11. In the first case, the solution works as follows: Imagine first that the input unit takes on a value of 0. In this case, there will be no activation from the input unit to the hidden unit, but the bias on the hidden unit will turn it on. Then the hidden unit has a *strong negative* connection to the output unit so it will be turned off, as required in this case. Now suppose that the input unit is set to 1. In this case, the strong inhibitory connection from the input to the hidden unit will turn the hidden unit off. Thus, no activation will flow from the hidden unit to the output unit. In this case, the positive bias on the output unit will turn it on and the problem will be solved. Now consider the second class of solutions. For this case, the connections among units are positive and the biases are negative. When the input unit is off, it cannot turn on the hidden unit. Since the hidden unit has a negative bias, it too will be off. The output unit, then, will not receive any



WEIGHTS AND BIASES OF THE SOLUTIONS FOR A 1:1:1 NETWORK

| Minima | $w_1$ | $w_2$ | $bias_1$ | $bias_2$ |
|---|---|---|---|---|
| Global | −8 | −8 | +4 | +4 |
| Global | +8 | +8 | −4 | −4 |
| Global | −8 | −8 | 0 | 0 |
| Local | +8 | +0.73 | 0 | 0 |

Figure 5.11:

input from the hidden unit and since its bias is negative, it too will turn off as required for zero input. Finally, if the input unit is turned on, the strong positive connection from the input unit to the hidden unit will turn on the hidden unit. This in turn will turn on the output unit as required. Thus we have, it appears, two symmetric solutions to the problem. Depending on the random starting state, the system will end up in one or the other of these *global* minima.

Interestingly, it is a simple matter to convert this problem to one with one local and one global minimum simply by setting the biases to 0 and not allowing them to change. In this case, the minima correspond to roughly the same two solutions as before. In one case, which is the global minimum as it turns out, both connections are large and negative. These minima are also illustrated in Figure 5.11. Consider first what happens with both weights negative. When the input unit is turned off, the hidden unit receives no input. Since the bias is 0, the hidden unit has a net input of 0. A net input of 0 causes the hidden unit to take on a value of 0.5. The 0.5 input from the hidden unit, coupled with a large negative connection from the hidden unit to the output unit, is sufficient to turn off the output unit as required. On the other hand, when the input unit is turned on, it turns off the hidden unit. When the hidden unit is off, the output unit receives a net input of 0 and takes on a value of 0.5 rather than the desired value of 1.0. Thus there is an error of 0.5 and a squared error of 0.25. This, it turns out, is the best the system can do with zero biases. Now consider what happens if both connections are positive. When the input unit is off, the hidden unit takes on a value of 0.5. Since the output is intended to be 0 in this case, there is pressure for the weight from the hidden unit to the output unit to be small. On the other hand, when the input unit is on, it turns on the hidden unit.

Since the output unit is to be on in this case, there is pressure for the weight to be large so it can turn on the output unit. In fact, these two pressures balance off and the system finds a compromise value of about 0.73. This compromise yields a summed squared error of about 0.45—a local minimum.

Usually, it is difficult to see why a network has been caught in a local minimum. However, in this very simple case, we have only two weights and can produce a contour map for the error space. The map is shown in Figure 5.1.2. It is perhaps difficult to visualize, but the map roughly shows a saddle shape. It is high on the upper left and lower right and slopes down toward the center. It then slopes off on each side toward the two minima. If the initial values of the weights begin one part of the space, the system will follow the contours down and to the left into the minimum in which both weights are negative. If, however, the system begins in another part of the space, the system will follow the slope into the upper right quadrant in which both weights are positive. Eventually, the system moves into a gently sloping valley in which the weight from the hidden unit to the output unit is almost constant at about 0.73 and the weight from the input unit to the hidden unit is slowly increasing. It is slowly being sucked into a local minimum. The directed arrows superimposed on the map illustrate thelines of force and illustrate these dynamics. The long arrows represent two trajectories through weightspace for two different starting points.

It is rare that we can create such a simple illustration of the dynamics of weight-spaces and see how clearly local minima come about. However, it is likely that many of our spaces contain these kinds of saddle-shaped error surfaces. Sometimes, as when the biases are free to move, there is a global minimum on either side of the saddle point. In this case, it doesn't matter which way you move off. At other times, such as in Figure 5.1.2, the two sides are of different depths. There is no way the system can sense the depth of a minimum from the edge, and once it has slipped in there is no way out. Importantly, however, we find that high-dimensional spaces (with many weights) have relatively few local minima.

*Momentum.* Our learning procedure requires only that the change in weight be proportional to the weight error derivative. True gradient descent requires that infinitesimal steps be taken. The constant of proportionality, $\epsilon$, is the learning rate in our procedure. The larger this constant, the larger the changes in the weights. The problem with this is that it can lead to steps that overshoot the minimum, resulting in a large increase in error. For practical purposes we choose a learning rate that is as large as possible without leading to oscillation. This offers the most rapid learning. One way to increase the learning rate without leading to oscillation is to modify the back propagation learning rule to include a *momentum* term. This can be accomplished by the following rule:

$$\Delta w_{ij}(n+1) \ = \ \epsilon(\delta_{ip}a_{jp}) + \alpha \Delta w_{ij}(n)$$

where the subscript $n$ indexes the presentation number and $\alpha$ is a constant that determines the effect of past weight changes on the current direction of movement in weight space. This provides a kind of momentum in weight-space that

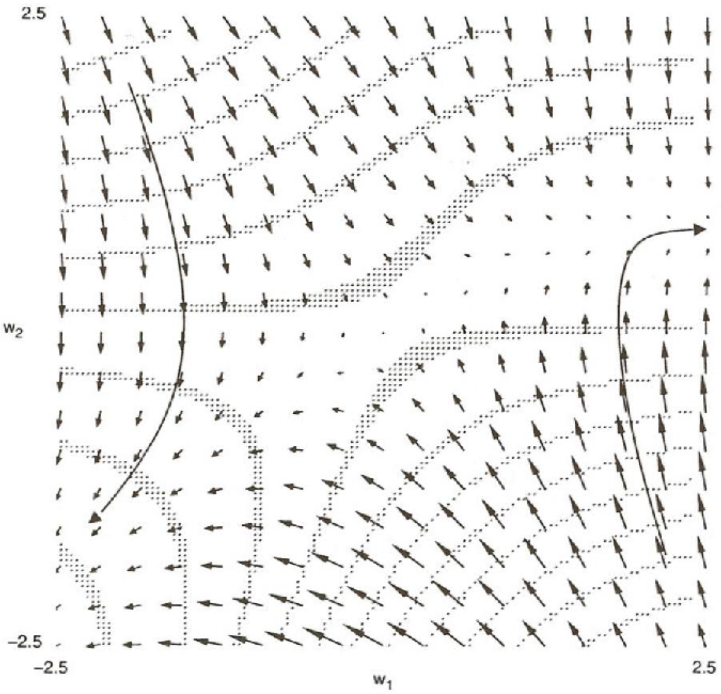


Figure 5.12: A contour map for the 1:1:1 identity problem with biases fixed at 0. The map show a local minimum in the positive quadrant and a global minimum in the lower left-hand negative quadrant. Overall the error surface is saddle-shaped. See the text for further explanation.

effectively filters out high-frequency variations of the error surface in the weight-space. This is useful in spaces containing long ravines that are characterized by steep walls on both sides of the ravine and a gently sloping floor. Such situations tend to lead to divergent oscillations across the ravine. To prevent these it is necessary to take very small steps, but this causes very slow progress along the ravine. The momentum tends to cancel out the tendency to jump across the ravine and thus allows the effective weight steps to be bigger. In most of the simulations reported in *PDP:8,* $\alpha$ was about 0.9. Our experience has been that we get the same solutions by setting $\alpha = 0$ and reducing the size of $\epsilon$, but the system learns much faster overall with larger values of $\alpha$ and $\epsilon$.

*Symmetry breaking.* Our learning procedure has one more problem that can be readily overcome and this is the problem of symmetry breaking. If all weights start out with equal values and if the solution requires that unequal weights be developed, the system can never learn. This is because error is propagated back through the weights in proportion to the values of the weights. This means that all hidden units connected directly to the output units will get identical error signals, and, since the weight changes depend on the error signals, the weights from those units to the output units must always be the same. The system is starting out at a kind of unstable equilibrium point that keeps the weights equal, but it is higher than some neighboring points on the error surface, and once it moves away to one of these points, it will never return. We counteract this problem by starting the system with small random weights. Under these conditions symmetry problems of this kind do not arise. This can be seen in Figure 5.1.2. If the system starts at exactly (0,0), there is no pressure for it to move at all and the system will not learn; if it starts virtually anywhere else, it will eventually end up in one minimum or the other.

*Weight decay.* One additional extension of the back propagation model that we will consider here is the inclusion of weight decay. Weight decay is simply a tendency for weights to be reduced very slightly every time they are updated. If weight-decay is non-zero, then the full equation for the change to each weight becomes the following:

$$\Delta w_{ij}(n+1) \;=\; \epsilon(\delta_{ip}a_{jp}) - \omega w_{ij}(n) + \alpha\Delta w_{ij}(n)$$

where $\omega$ is a positive constant representing the strength of the weight decay. Weight decay can be seen as a procedure for minimizing the total magnitude of the weights, where the magnitude is the sum of the squares of the weights. It should be noted that minimizing the sum of the squares of the weights can be in competition with minimizing the error measure, and so if weight decay is too strong it can often interfere with reaching an acceptable performance criterion.

*Learning by pattern or by epoch.* The derivation of the back propagation rule presupposes that we are taking the derivative of the error function summed over all patterns. In this case, we might imagine that we would present all patterns and then sum the derivatives before changing the weights. Instead, we can compute the derivatives on each pattern and make the changes to the weights after each pattern rather than after each epoch. Although the former approach can be implemented more efficiently (weight error derivatives for each pattern can be computed in parallel over separate processors, for example) the latter approach may be more plausible from a human or biological learning perspective, where it seems that learning does occur "on line". Also, if the training set is large, consistent weight error derivatives across patterns can add up and produce a huge overshoot in the change to a connection weight. The **bp** program allows weight changes after each pattern or after each epoch. In fact, the user may specify the size of a batch of patterns to be processed before weights are updated.[3]

[3] We moving between these options, it is important to note that weight decay is applied each time weights are updated. If weights are updated after each pattern, a smaller value of weight decay should be used than if they are updated after a batch of n patterns or a whole epoch.

## 5.2 IMPLEMENTATION

The **bp** program implements the back propagation process just described. Networks are assumed to be feedforward only, with no recurrence. An implementation of backpropagation for recurrent networks is described in a later chapter.

The network is specified in terms of a set of pools of units. By convention, pool(1) contains the single bias unit, which is always on. Subsequent pools are declared in an order that corresponds to the feed-forward structure of the network. Since activations at later layers depend on the activations at earlier layers, the activations of units must be processed in correct order, and therefore the order of specification of pools of units is important. Indeed, since deltas at each layer depend on the delta terms from the layers further forward, the backward pass must also be carried out in the correct order. Each pool has a type: it can be an input pool, an output pool, or a hidden pool. There can be more than one input pool and more than one output pool and there can be 0 or more hidden pools. Input pools must all be specified before any other pools and all hidden pools must be specified before any output pools.

Connections among units are specified by projections. Projections may be from any pool to any higher numbered pool; since the bias pool is pool(1) it may project to any other pool, although bias projections to input pools will have no effect since activations of input units are clamped to the value specified by the external input. Projections from a layer to itself are not allowed.

Weights in a projection can be constrainted to be positive or negative. These constraints are imposed both at initialization and after each time the weights are incremented during processing. Two other constraints are imposed only when weights are initialized; these constraints specify either a fixed value to which the weight is initialized, or a random value. For weights that are random, if they are constrained to be positive, they are initialized to a value between 0 and the value of a parameter called *wrange*; if the weights are constrained to be negative, the initialization value is between -*wrange* and 0; otherwise, the initialization value is between *wrange*/2 and -*wrange*/2. Weights that are constrained to a fixed value are initialized to that value.

The program also allows the user to set an individual learning rate for each projection via a layer-specific *lrate* parameter. If the value of this layer-specific *lrate* is unspecified, the network-wide *lrate* variable is used.

The **bp** program also makes use of a list of pattern pairs, each pair consisting of a name, an input pattern, and a target pattern. The number of elements in the input pattern should be equal to the total number of units summed across all input pools. Similarly, the number of elements of the target pattern should be equal to the total number of output units summed across all output pools.

Processing of a single pattern occurs as follows: A pattern pair is chosen, and the pattern of activation specified by the input pattern is clamped on the input units; that is, their activations are set to whatever numerical values are specified in the input pattern. These are typically 0's and 1's but may take any real value.

Next, activations are computed. For each noninput pool, the net inputs to each unit are computed and then the activations of the units are set. This occurs in the order that the pools are specified in the network specification, which must be specified correctly so that by the time each unit is encountered, the activations of all of the units that feed into it have already been set. The routine performing this computation is called *compute_output.* Once the output has been computed some summary statistics are computed in a routine called *sumstats.* First it computes the pattern sum of squares *(pss),* equal to the squared error terms summed over all of the output units. Analogously, the *pce* or pattern cross entropy, the sum of the cross entropy terms accross all the output units, is calculated. Then the routine adds the *pss* to the total sum of squares *(tss),* which is just the cumulative sum of the *pss* for all patterns thus far processed within the current epoch. Similarly the *pce* is added to the *tce*, or total cross entropy measure.

Next, *error* and *delta* terms are computed in a routine called *compute_error.* The *error* for a unit is equivalent to (minus) the partial derivative of the error with respect to a change in the *activation* of the unit. The *delta* for the unit is (minus) the partial derivative of the error with respect to a change in the *net input* to the unit. First, the *error* terms are calculated for each output unit. For these units, *error* is the difference between the target and the obtained activation of the unit. After the error has been computed for each output unit, we get to the "heart" of back propagation: the recursive computation of *error* and *delta* terms for hidden units. The program iterates backward over the layers, starting with the last output layer. The first thing it does in each layer is set the value of *delta* for the units in the current layer; this is equal to the *error* for the unit times the derivative of the activation function as described above. Then, once it has the *delta* terms for the current pool, the program passes this back to all pools that project to the current pool; this is the actual back propagation process. By the time a particular pool becomes the current pool, all of the units that it projects to will have already been processed and its total error will have been accumulated, so it is ready to have its *delta* computed.

After the backward pass, the weight error derivatives are then computed from the *deltas* and *activations* in a routine called *compute_weds.* Note that this routine adds the weight error derivatives occasioned by the present pattern into an array where they can potentially be accumulated over patterns.

Weight error derivatives actually lead to changes in the weights when a routine called *change_weights* is called. This may be called after each pattern has been processed, or after each batch of $n$ patterns, or after all patterns in the training set have been processed. When this routine is called, it cycles through all the projections in the network. For each, the new *delta weight* is first calculated. The delta weight is equal to (1) the accumulated weight error derivative scaled by the *lrate*, minus the weight decay scaled by *wdecay*, plus a fraction of the previous delta weight where the fraction is the value of the *momentum* parameter. Then, this delta weight is added into the weight, so that the weight's new value is equal to its old value plus the delta weight. At the end of processing each projection, the weight error derivative terms are all set to 0, and constraints on the values of the weights are imposed in the routine *constrain_weights*.

Generally, learning is accomplished through a sequence of *epochs*, in which all pattern pairs are presented for one trial each during each epoch. The presentation is either in sequential or permuted order. It is also possible to test the processing of patterns, either individually or by sequentially cycling through the whole list, with learning turned off. In this case, *compute_output*, *compute_error*, and *sumstats* are called, but *compute_wed* and *change_weights* are not called.

## 5.3 RUNNING THE PROGRAM

The **bp** program is used much like earlier programs in this series, particularly **pa.** Like the other programs, it has a flexible architecture that is specified using a *.net* file, and a flexible screen layout that is specified in a *.tem* file. The program also makes use of a *.pat* file, in which the pairs of patterns to be used in training and testing the network are listed.

When networks are initialized, the weights are generally assigned according to a pseudo-random number generator. As in **pa** and **iac,** the *reset* command allows the user to repeat a simulation run with the same initial configuration used just before. (Another procedure for repeating the previous run is described in Ex. 5.1.) The *newstart* command generates a new random seed and seeds the random number generator with it before using the seed in generating a new set of random initial values for the connection weights.

Control over learning occurs by setting options in the training window or via the *settrainopts* function. The number of epochs *nepochs* to train each time "run" is called can be specified. The user can also specify whether weights are updated after every pattern, after n patterns, or after every epoch, and whether the mode of pattern presentation is sequential *strain* or permuted *ptrain.* The learning rate, weight decay, and momentum parameters can all be specified as well. The user can also set a stopping criterion for training called "ecrit". The variable *wrange*, the range of values allowed when weights are initialized or re-initialized, can also be adjusted.

*fast mode for training.* The bp program's inner loops have been converted to fast mex code (c code that interfaces with MATLAB). To turn on the fast mode for network training, click the 'fast' option in the train window. This option can also be set by typing the following command on the MATLAB command prompt:

```
runprocess('process','train','granularity','epoch','count',1,'fastrun',1);
```

There are some limitations in executing fast mode of training.In this mode, network output logs can only be written at epoch level. Any smaller output frequency if specified, will be ignored.When running with fast code in gui mode, network viewer will only be updated at epoch level.Smaller granularity of display update is disabled to speed up processing. The 'cascade' option of building activation gradually is also not available in fast mode.

There is a known issue with running this on linux OS.MATLAB processes running fast version appear to be consuming a lot of system memory. This memory management issue has not been observed on windows. We are currently investigating this and will be updating the documentation as soon as we find a fix.

In the **bp** program the principle measures of performance are the pattern sum of squares *(pss)* and the total sum of squares *(tss)*, and the pattern cross-entropy *pce* and the total cross entropy *tce.* The user can specify whether the error measure used in computing error derivatives is the sum squared error or the cross entropy. Because of its historical precedence, the sum squared error is used by default. The user may optionally also compute an additional measure, the vector correlation of the present weight error derivatives with the previous weight error derivatives. The set of weight error derivatives can be thought of as a vector pointing in the steepest direction downhill in weight space; that is, it points down the error gradient. Thus, the vector correlation of these derivatives across successive epochs indicates whether the gradient is staying relatively stable or shifting from epoch to epoch. For example, a negative value of this correlation measure (called *gcor* for *gradient correlation*) indicates that the gradient is changing in direction. Since the *gcor* can be thought of as following changes in the direction of the gradient, the check box for turning on this computation is called *follow*

Control over testing is straightforward. With the "test all" box checked, the user may either click run to carry out a complete pass through the test set, or click step to step pattern by pattern, or the user may uncheck the "text all" button and select an individual pattern by clicking on it in the network viewer window and then clicking *run* or *step.*

There is a special mode available in the **bp** program called *cascade* mode. This mode allows activation to build up gradually rather than being computed in a single step as is usually the case in *bp.* A discussion of the implementation and use of this mode is provided later in this chapter.

As with other **pdptool** programs the user may adjust the frequency of display updating in the train and test windows. It is also possible to log and create graphs of the state of the network at the pattern or epoch level using *create/edit logs* within the training and testing *options* panels.

## 5.4 EXERCISES

We present four exercises using the basic back propagation procedure. The first one takes you through the XOR problem and is intended to allow you to test and consolidate your basic understanding of the back propagation procedure and the gradient descent process it implements. The second allows you to explore the wide range of different ways in which the XOR problem can be solved; as you will see the solution found varies from run to run initialized with different starting weights. The third exercise suggests minor variations of the basic back propagation procedure, such as whether weights are changed pattern by pattern or epoch by epoch, and also proposes various parameters that may be explored. The fourth exercise suggests other possible problems that you might want to

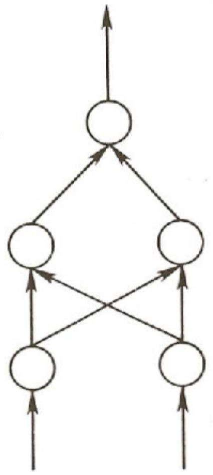


Figure 5.13: Architecture of the XOR network used in the exercises (From *PDP:8*, p.332.)

explore using back-propagation.

## Ex5.1. The XOR Problem

The XOR problem is described at length in *PDP:8*. Here we will be considering one of the two network architectures considered there for solving this problem. This architecture is shown in Figure 5.13. In this network configuration there are two input units, one for each "bit" in the input pattern. There are also two hidden units and one output unit. The input units project to the hidden units, and the hidden units project to the output unit; there are no direct connections from the input units to the output units.

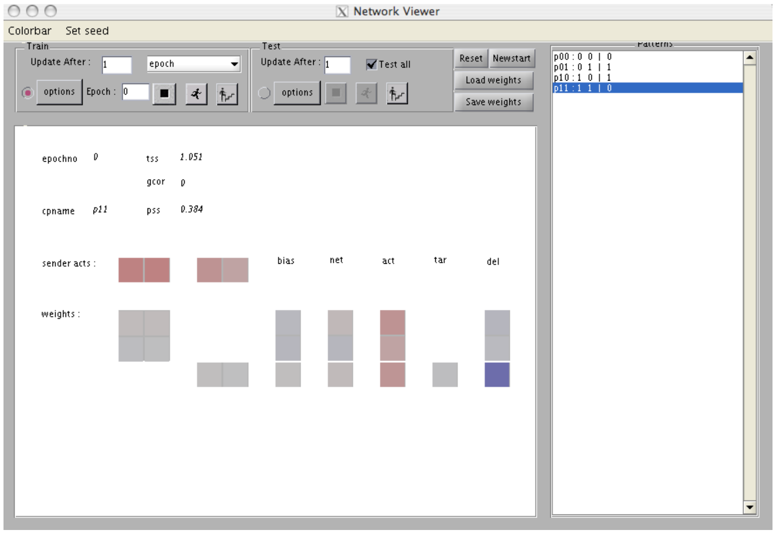
All of the relevant files for doing this exercise are contained in the *bp* directory; they are called *xor.tem, xor.net, xor.pat*, and *xor. wts*.

Once you have downloaded the latest version of the software, started MATLAB, set your path to include the *pdptool* directory and all of its children, and changed to the *bp* directory, you can simply type

*bpxor*

to the MATLAB command-line prompt. This file instructs the program to set up the network as specified in the *xor.net* file and to read the patterns as specified in the *xor.pat* file; it also initializes various variables. Then it reads in an initial set of weights to use for this exercise. Finally, a test of all of the patterns in the training set is performed. The network viewer window that finally appears shows the state of the network at the end of this initial test of all of the patterns. It is shown in Figure 5.14.

The display area in the network viewer window shows the current epoch



```
Train   Update After : 1   epoch   Epoch : 0
Test    Update After : 1   ☑ Test all

epochno  0        tss   1.051
                  gcor  0
cpname   p11      pss   0.384

p00 : 0  0  |  0
p01 : 0  1  |  1
p10 : 1  0  |  1
p11 : 1  1  |  0
```

Figure 5.14: The display produced by the *bp* program, initialized for XOR.

number and the total sum of of squares *(tss)* resulting from testing all four patterns. The next line contains the value of the gcor variable, currently 0 since no error derivatives have yet been calculated. Below that is a line containing the current pattern name and the pattern sum of squares *pss* associated with this pattern. To the right in the "patterns" panel is the set of input and target patterns for XOR. Back in the main network viewer window, we now turn our attention to the area to the right and below the label "sender acts". The colored squares in this row shows the activations of units that send their activations forward to other units in the network. The first two are the two input units, and the next two are the two hidden units. Below each set of sender activations are the corresponding projections, first from the input to the hidden units, and below and to the right of that, from the hidden units to the single output unit. The weight in a particular column and row represents the strength of the connection from a particular sender unit indexed by the column to the particular receiver indexed by the row.

To the right of the weights is a column vector indicating the values of the bias terms for the *receiver* units-that is, all the units that receive input from other units. In this case, the receivers are the two hidden units and the output unit.

To the right of the biases is a column for the net input to each receiving unit. There is also a column for the activations of each of these receiver units. (Note that the hidden units' activations appear twice, once in the row of senders and once in this column of receivers.) The next column contains the target vector, which in this case has only one element since there is only one output unit. Finally, the last column contains the delta values for the hidden and output units.

Note that shades of red are used to represent positive values, shades of blue are used for negative values, and a neutral gray color is used to represent 0. The color scale for weights, biases, and net inputs ranges over a very broad range, and values less than about .5 are very faint in color. The color scale for activations ranges over somewhat less of a range, since activations can only range from 0 to 1. The color scale for deltas ranges over a very small range since delta values are very small. Even so, the delta values at the hidden level show very faintly compared with those at the output level, indicating just how small these delta values tend to be, at least at this early stage of training. You can inspect the actual numberical values of each variable by moving your mouse over the corresponding colored square.

The display shows what happened when the last pattern pair in the file *xor.pat* was processed. This pattern pair consists of the input pattern (1 1) and the target pattern (0). This input pattern was clamped on the two input units. This is why they both have activation values of 1.0, shown as a fairly saturated red in the first two entries of the sender activation vector. With these activations of the input units, coupled with the weights from these units to the hidden units, and with the values of the bias terms, the net inputs to the hidden units were set to 0.60 and -0.40, as indicated in the *net* column. Plugging these values into the logistic function, the activation values of 0.64 and 0.40 were obtained for these units. These values are shown both in the sender activation vector and in the receiver activation vector (labeled act, next to the net input vector). Given these activations for the hidden units, coupled with the weights from the hidden units to the output unit and the bias on the output unit, the net input to the output unit is 0.48, as indicated at the bottom of the *net* column. This leads to an activation of 0.61, as shown in the last entry of the *act* column. Since the target is 0.0, as indicated in the target column, the *error*, or *(target - activation)* is -0.61; this error, times the derivative of the activation function (that is, *activation* (1 - *activation))* results in a delta value of -0.146, as indicated in the last entry of the final column. The delta values of the hidden units are determined by back propagating this delta term to the hidden units, using the back-propagation equation.

Q.5.1.1.

> Show the calculations of the values of *delta* for each of the two hidden units, using the activations and weights as given in this initial screen display, and the BP Equation. Explain why these values are so small.

At this point, you will notice that the total sum of squares before any learning has occurred is 1.0507. Run another *tall* to understand more about what is happening.

Q.5.1.2.

> Report the output the network produces for each input pattern and explain why the values are all so similar, referring to the strengths of

> the weights, the logistic function, and the effects of passing activation forward through the hidden units before it reaches the output units.

Now you are ready to begin learning. Acivate the training panel. If you click run (don't do that yet), this will run 30 epochs of training, presenting each pattern sequentially in the order shown in the patterns window within each epoch, and adjusting the weights at the end of the epoch. If you click step, you can follow the *tss* and *gcor* measures as they change from epoch to epoch. A graph will also appear showing the *tss*. If you click run after clicking step a few times, the network will run to the 30 epoch milestone, then stop.

You may find in the course of running this exercise that you need to go back and start again. To do this, you should use the *reset* command, followed by clicking on the load weights button, and selecting the file *xor.wts*. This file contains the initial weights used for this exercise. This method of reinitializing guarantees that all users will get the same starting weights.

After completing the first 30 epochs, stop and answer this question.

Q.5.1.3.

> The total sum of squares is smaller at the end of 30 epochs, but is only a little smaller. Describe what has happened to the weights and biases and the resulting effects on the activation of the output units. Note the small sizes of the deltas for the hidden units and explain. Do you expect learning to proceed quickly or slowly from this point? Why?

Run another 90 epochs of training (for a total of 120) and see if your predictions are confirmed. As you go along, watch the progression of the *tss* in the graph that should be displayed (or keep track of this value at each 30 epoch milestone by recording it manually). You might find it interesting to observe the results of processing each pattern rather than just the last pattern in the four-pattern set. To do this, you can set the update after selection to 1 pattern rather than 1 epoch, and use the step button for an epoch or two at the beginning of each set of 30 epochs.

At the end of another 60 epochs (total: 180), some of the weights in the network have begun to build up. At this point, one of the hidden units is providing a fairly sensitive index of the number of input units that are on. The other is very unresponsive.

Q.5.1.4.

> Explain why the more responsive hidden unit will continue to change its incoming weights more rapidly than the other unit over the next few epochs.

Run another 30 epochs. At this point, after a total of 210 epochs, one of the hidden units is now acting rather like an OR unit: its output is about the same for all input patterns in which one or more input units is on.

Q.5.1.5.

> Explain this OR unit in terms of its incoming weights and bias term. What is the other unit doing at this point?

Now run another 30 epochs. During these epochs, you will see that the second hidden unit becomes more differentiated in its response.

Q.5.1.6.

> Describe what the second hidden unit is doing at this point, and explain why it is leading the network to activate the output unit most strongly when only one of the two input units is on.

Run another 30 epochs. Here you will see the *tss* drop very quickly.

Q.5.1.7.

> Explain the rapid drop in the *tss*, referring to the forces operating on the second hidden unit and the change in its behavior. Note that the size of the *delta* for this hidden unit at the end of 270 epochs is about as large in absolute magnitude as the size of the *delta* for the output unit. Explain.

Click the run button one more time. Before the end of the 30 epochs, the value of *tss* drops below *ecrit*, and so training stops. The XOR problem is solved at this point.

Q.5.1.8.

> Summarize the course of learning, and compare the final state of the weights with their initial state. Can you give an approximate intuitive account of what has happened? What suggestions might you make for improving performance based on this analysis?

## Ex5.2. Learning XOR with different initial weights

Run the XOR problem several more times, each time using newstart to get a new random configuration of weights. Write down the value of the random seed after each newstart (you will find it by clicking on *Set Seed* in the upper left hand corner of the network viewer window). Then run for up to 1000 epochs, or until the *tss* reaches the criterion (you can set nepochs in the test window to 1000, and set *update after* to, say, 50 epochs). We will call cases in which the *tss* reaches the criterion *successful* cases. Continue til you find two successful runs that reach different solutions than the one found in Exercise 5.1. Read on for further details before proceeding.

Q.5.2.1.

> At the end of each run, record, after each random seed, the final epoch number, and the final tss. Create a table of these results to turn in as part of your homework. Then, run through a test, inspecting the activations of each hidden unit and the single output unit obtained for each of the four patterns. Choose two successful runs that seem to have reached different solutions than the one reached in Exercise 5.1, as evidenced by qualitative differences in the hidden unit activation patterns. For these two runs, record the hidden and output unit activations for each of the four patterns, and include these results in a second table as part of what you turn in. For each case, state what logical predicate each hidden unit appears to be calculating, and how these predicates are then combined to determine the activation of the output unit. State this in words for each case, and also use the notation described in the *hint* below to express this information succinctly.

*Hint.* The question above may seem hard at first, but should become easier as you consider each case. In Excercise 5.1, one hidden unit comes on when either input unit is on (i.e., it acts as an OR unit), and the other comes on when both input units are on (i.e., it acts as an AND unit). For this exercise, we are looking for qualitatively different solutions. You might find that one hidden unit comes on when the first input unit is on and the second is off (This could be called 'A and not B'), and the other comes on when the first is on and the second is off ('B and not A'). The weights from the hidden units to the output unit will be different from what they were in Exercise 5.1, reflecting a difference in the way the predicates computed by each hidden unit are combined to solve the XOR problem. In each case you should be able to describe the way the problem is being solved using logical expressions. Use A for input unit 1, B for input unit 2, and express the whole operation as a compound logical statement, using the additional logical terms 'AND' 'OR' and 'NOT'. Use square brackets around the expression computed by each hidden unit, then use logical terms to express how the predicates computed by each hidden unit are combined. For example, for the case in Exercise 5.1, we would write: [A OR B] AND NOT [A AND B].

## Ex5.3. Effects of parameters on XOR Learning.

There are several further studies one can do with XOR. You can study the effects of varying:

1. One of the parameters of the model *(lrate, wrange, momentum).*
2. Frequency of weight updating: once per *pattern* or *epoch.*
3. The training regime: *permuted* vs. *sequential* presentation. This makes a difference only when the frequency of weight updating is equal to *pattern.*
4. The magnitude of the initial random weights (determined by *wrange,*).
5. The error measure: cross-entropy vs. sum squared error.

You are encouraged to do your own exploration of one of these parameters, trying to examine its effects of the rate and outcome of learning. We don't want to prescribe your experiments too specifically, but one thing you could do would be the following. Re-run each of the eight runs that you carried out in the previous exercise under the variation that you have chosen. To do this, you first set the training option for your chosen variation, then you set the random seed to the value from your first run above, then click reset. The network will now be initialized exactly as it was for that first run, and you can now test the effect of your chosen variation by examining whether it effects the time course or the outcome of learning. You could repeat these steps for each of your runs, exploring how the time course and outcome of learning are affected.

Q.5.3.1.

> Describe what you have chosen to vary, how you chose to vary it, and present the results you obtained in terms of the rate of learning, the evolution of the weights, and the eventual solution achieved. Explain as well as you can why the change you made had the effects you found.

For a thorough investigation, you might find it interesting to try several different values along the dimension you have chosen to vary, and see how these parametric variations affect your solutions. Sometimes, in such explorations, you can find that things work best with an intermediate value of some parameter, and get worse for both larger and smaller values.

## Ex5.4. Other Problems for Back Propagation

This exercise encourages you to construct a different problem to study, either choosing from those discussed in *PDP:8* or choosing a problem of your own. Set up the appropriate network, template, pattern, and start-up files, and experiment with using back propagation to learn how to solve your problem.

Q.5.4.1.

> Describe the problem you have chosen, and why you find it interesting. Explain the network architecture that you have selected for the problem and the set of training patterns that you have used. Describe the results of your learning experiments. Evaluate the back propagation method for learning and explain your feelings as to its adequacy, drawing on the results you have obtained in this experiment and any other observations you have made from the readings or from this exercise.

*Hints.* To create your own network, you will need to create the necessary .net, .tem, and .pat files yourself; once you've done this, you can create a script file (with .m extension) that reads these files and launches your network. The steps you need to take to do this are described in Appendix B, *How to create* *your own network*. More details are available in the *PDPTool User's Guide*, Appendix C.

In general, if you design your own network, you should strive to keep it simple. You can learn a lot with a network that contains as few as five units (the XOR network considered above), and as networks become larger they become harder to understand.

To achieve success in training your network, there are many parameters that you may want to consider. The exercises above should provide you with some understanding of the importance of some of these parameters. The learning rate (*lrate*) of your network is important; if it is set either too high or too low, it can hinder learning. The default 0.1 is fine for some simple networks (e.g., the 838 encoder example discussed in Appendix B), but smaller rates such as 0.05, 0.01 or 0.001 are often used, especially in larger networks. Other parameters to consider are momentum, the initial range of the weights (*wrange*), the weight update frequency variable, and the order of pattern presentation during training (all these are set through the train options window).

If you are having trouble getting your network to learn, the following approach may not lead to the fastest learning but it seems fairly robust: Set momentum to 0, set the learning rate fairly low (.01), set the update frequency to 1 pattern, set the training regime to permuted (*ptrain*), and use cross-entropy error. If your network still doesn't learn make sure your network and training patterns are specified correctly. Sometimes, it may also be necessary to add hidden units, though it is surprising how few you can get away with in many cases, though with the minimum number, as we know from XOR, you can get stuck sometimes.

The range of the initial random weights can hinder learning if it is set too high or too low. A range that is too high (such as 20) will push the hidden units to extreme activation values (0 or 1) before the network has started learning, which can harm learning (why?). If this parameter is too small (such as .01), learning can also be very slow since the weights dilute the back propagation of error. The default wrange of 1 is ok for smaller networks, but it may be too big for larger networks. Also, it may be worth noting that, while a smaller *wrange* and learning rate tends to lead to slower learning, it tends to produce more consistent results across different runs (using different initial random weights).

*Other pre-defined bp networks.* In addition to XOR, there are two further examples provided in the PDPTool/bp directory. One of these is the 4-2-4 encoder problem described in PDP:8. The files 424.tem, 424.net, 424.pat, and FourTwoFour.m are already set up for this problem just type FourTwoFour at the command prompt to start up this network. The network viewer window is layed out as with XOR, such that the activations of the input and hidden units are shown across the top, and the bias, net input, activations, targets and deltas for the hidden and output units are shown in vertical columns to the right of the two arrays of weights.

Another network that is also ready to run is Rumelhart's Semantic Network, described in Rumelhart and Todd (1993), Rogers and McClelland (2004) (Chapters 2 and 3), and McClelland and Rogers (2003). The files for this are called semnet.net, semnet.tem, EightThings.pat and semnet.m. The exercise can be started by typing semnet to the command prompt. Details of the simulation are close to those used in McClelland and Rogers (2003). Learning takes on the order of 1000 epochs for all patterns to reach low *pss* values with the given parameters.

# Chapter 6

# Competitive Learning

In Chapter 5 we showed that multilayer, nonlinear networks are essential for the solution of many problems. We showed one way, the back propagation of error, that a system can learn appropriate features for the solution of these difficult problems. This represents the basic strategy of pattern association—to search out a representation that will allow the computation of a specified function. There is a second way to find useful internal features: through the use of a *regularity detector*, a device that discovers useful features based on the stimulus ensemble and some a priori notion of what is important. The competitive learning mechanism described in *PDP:5* is one such regularity detector. In this section we describe the basic concept of competitive learning, show how it is implemented in the **cl** program, describe the basic operations of the program, and give a few exercises designed to familiarize the reader with these ideas.

## 6.1 SIMPLE COMPETITIVE LEARNING

### 6.1.1 Background

The basic architecture of a competitive learning system (illustrated in Figure 6.1) is a common one. It consists of a set of hierarchically layered units in which each layer connects, via excitatory connections, with the layer immediately above it, and has inhibitory connections to units in its own layer. In the most general case, each unit in a layer receives an input from each unit in the layer immediately below it and projects to each unit in the layer immediately above it. Moreover, within a layer, the units are broken into a set of inhibitory clusters in which all elements within a cluster inhibit all other elements in the cluster. Thus the elements within a cluster at one level compete with one another to respond to the pattern appearing on the layer below. The more strongly any particular unit responds to an incoming stimulus, the more it shuts down the other members of its cluster.

There are many variants to the basic competitive learning model. von der Malsburg (1973), Fukushima (1975), and Grossberg (1976), among others, have

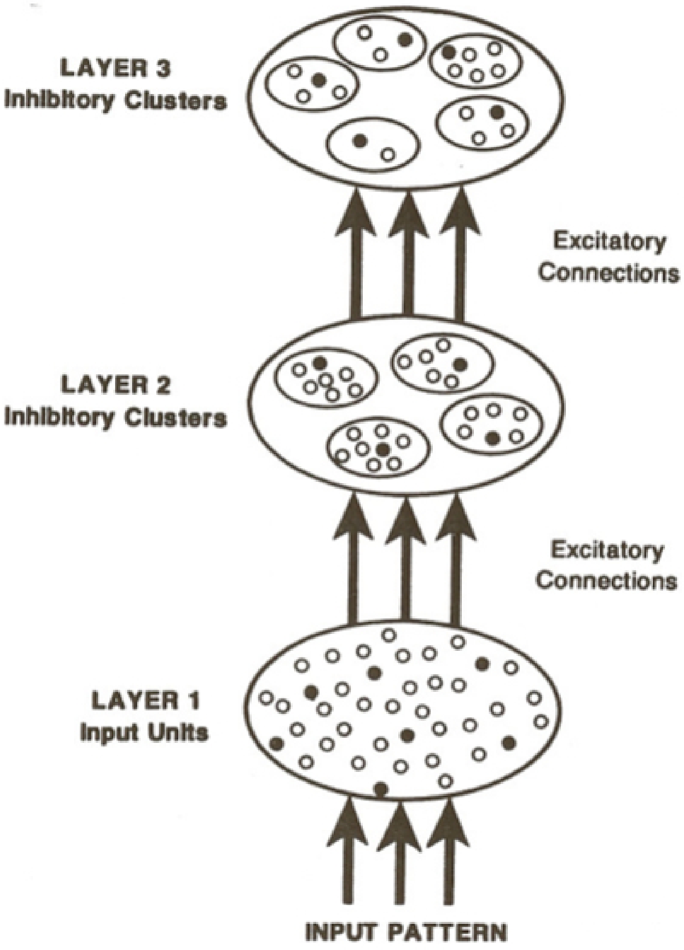


Figure 6.1: The architecture of the competitive learning mechanism. Competitive learning takes place in a context of sets of hierarchically layered units. Units are represented in the diagram as dots. Units may be active or inactive. Active units are represented by filled dots, inactive ones by open dots. In general, a unit in a given layer can receive inputs from all of the units in the next lower layer and can project outputs to all of the units in the next higher layer. Connections between layers are excitatory and connections within layers are inhibitory. Each layer consists of a set of clusters of mutually inhibitory units. The units within a cluster inhibit one another in such a way that only one unit per cluster may be active. We think of the configuration of active units on any given layer as representing the input pattern for the next higher level. There can be an arbitrary number of such layers. A given cluster contains a fixed number of units, but different clusters can have different numbers of units. (From "Feature Discovery by Competitive Learning" by D. E. Rumelhart and D. Zipser, 1985, *Cognitive Science*, 9, 75-112. Copyright 1985 by Ablex Publishing. Reprinted by permission.)

developed competitive learning models. In this section we describe the simplest of the many variations. The version we describe was first proposed by Grossberg (1976) and is the one studied by Rumelhart and Zipser (also in *PDP:5*). This version of competitive learning has the following properties:

- The units in a given layer are broken into several sets of nonoverlapping clusters. Each unit within a cluster inhibits every other unit within a cluster. Within each cluster, the unit receiving the largest input achieves its maximum value while all other units in the cluster are pushed to their minimum value.[1] We have arbitrarily set the maximum value to 1 and the minimum value to 0.
- Every unit in every cluster receives inputs from all members of the same set of input units.
- A unit learns if and only if it wins the competition with other units in its cluster.
- A stimulus pattern $S_j$ consists of a binary pattern in which each element of the pattern is either *active* or *inactive*. An active element is assigned the value 1 and an inactive element is assigned the value 0.
- Each unit has a fixed amount of weight (all weights are positive) that is distributed among its input lines. The weight on the line connecting to unit $i$ on the upper layer from unit $j$ on the lower layer is designated $w_{ij}$. The fixed total amount of weight for unit $j$ is designated $\sum_j w_{ij} = 1$. A unit learns by shifting weight from its inactive to its active input lines. If a unit does not respond to a particular pattern, no learning takes place in that unit. If a unit wins the competition, then each of its input lines gives up some portion $\epsilon$ of its weight and that weight is then distributed equally among the active input lines. Mathematically, this learning rule can be stated

$$\Delta w_{ij} = \begin{cases} 0 & \text{if unit } i \text{ loses on stimulus } k \\ \epsilon \frac{active_{jk}}{nactive_k} - \epsilon w_{ij} & \text{if unit } i \text{ wins on stimulus } k \end{cases} \tag{6.1}$$

  where $active_{jk}$ is equal to 1 if in stimulus pattern $S_k$, unit $j$ in the lower layer is active and is zero otherwise, and $nactive_k$ is the number of active units in pattern $S_k$ (thus $nactive_k = \sum_j active_{jk}$).[2]

Figure 6.2 illustrates a useful geometric analogy to this system. We can consider each stimulus pattern as a vector. If all patterns contain the same

[1] A simple circuit, employed by Grossberg (1976) for achieving this result, is attained by having each unit activate itself and inhibit its neighbors. Such a network can readily be employed to *choose* the maximum value of a set of units. In our simulations, we do not use this mechanism. We simply compute the maximum value directly.

[2] Note that for consistency with the other chapters in this book we have adopted terminology here that is different from that used in the *PDP:5*. Here we use $\epsilon$ where $g$ was used in *PDP:5*. Also, here the weight to unit $i$ from unit $j$ is designated $w_{ij}$. In *PDP:5*, $i$ indexed the sender not the receiver, so $w_{ij}$ referred to the weight from unit $i$ to unit $j$.

number of active lines, then all vectors are the same length and each can be viewed as a point on an $N$-dimensional hypersphere, where $N$ is the number of units in the lower level, and therefore, also the number of input lines received by each unit in the upper level. Each × in Figure 6.2A represents a particular pattern. Those patterns that are very similar are near one another on the sphere, and those that are very different are far from one another on the sphere. Note that since there are $N$ input lines to each unit in the upper layer, its weights can also be considered a vector in $N$-dimensional space. Since all units have the same total quantity of weight, we have $N$-dimensional vectors of approximately fixed length for each unit in the cluster.[3] Thus, properly scaled, the weights themselves form a set of vectors that (approximately) fall on the surface of the same hypersphere. In Figure 6.2B, the ○'s represent the weights of two units superimposed on the same sphere with the stimulus patterns. Whenever a stimulus pattern is presented, the unit that responds most strongly is simply the one whose weight vector is nearest that for the stimulus. The learning rule specifies that whenever a unit wins a competition for a stimulus pattern, it moves a fraction $\epsilon$ of the way from its current location toward the location of the stimulus pattern on the hypersphere. Suppose that the input patterns fell into some number, $M$, of "natural" groupings. Further, suppose that an inhibitory cluster receiving inputs from these stimuli contained exactly $M$ units (as in Figure 6.2C). After sufficient training, and assuming that the stimulus groupings are sufficiently distinct, we expect to find one of the vectors for the $M$ units placed roughly in the center of each of the stimulus groupings. In this case, the units have come to detect the grouping to which the input patterns belong. In this sense, they have "discovered" the structure of the input pattern sets.

### 6.1.2 Some Features of Competitive Learning

There are several characteristics of a competitive learning mechanism that make it an interesting candidate for study, for example:

- Each cluster classifies the stimulus set into $M$ groups, one for each unit in the cluster. Each of the units captures roughly an equal number of stimulus patterns. It is possible to consider a cluster as forming an $M$-valued feature in which every stimulus pattern is classified as having exactly one of the $M$ possible values of this feature. Thus, a cluster containing two units acts as a binary feature detector. One element of the cluster responds when a particular feature is present in the stimulus pattern, otherwise the other element responds.

- If there is structure in the stimulus patterns, the units will break up the patterns along structurally relevant lines. Roughly speaking, this means

[3] It should be noted that this geometric interpretation is only approximate. We have used the constraint that $\sum_j w_{ij} = 1$ rather than the constraint that $\sum_j w_{ij}^2 = 1$. This latter constraint would ensure that all vectors are in fact the same length. Our assumption only assures that they will be approximately the same length.

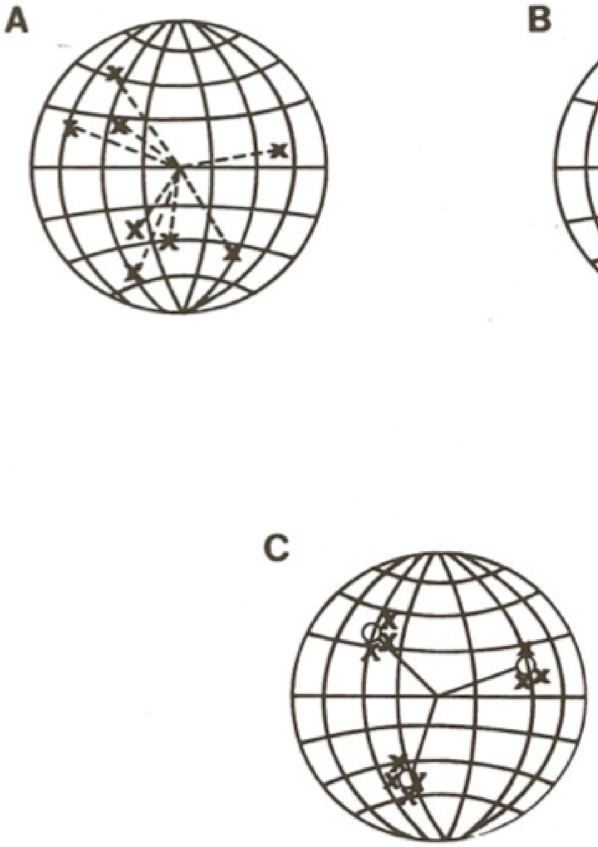


Figure 6.2: A geometric interpretation of competitive learning. *A*: It is useful to conceptualize stimulus patterns as vectors whose tips all lie on the surface of a hypersphere. We can then directly see the similarity among stimulus patterns as distance between the points on the sphere. In the figure, a stimulus pattern is represented as an ×. The figure represents a population of eight stimulus patterns. There are two clusters of three patterns and two stimulus patterns that are rather distinct from the others. *B*: It is also useful to represent the weights of units as vectors falling on the surface of the same hypersphere. Weight vectors are represented in the figure as ○'s. The figure illustrates the weights of two units falling on rather different parts of the sphere. The response rule of this model is equivalent to the rule that whenever a stimulus pattern is presented, the unit whose weight vector is closest to that stimulus pattern on the sphere wins the competition. In the figure, one unit would respond to the cluster in the northern hemisphere and the other unit would respond to the rest of the stimulus patterns. *C*: The learning rule of this model is roughly equivalent to the rule that whenever a unit wins the competition (i.e., is closest to the stimulus pattern), that weight vector is moved toward the presented stimulus. The figure shows a case in which there are three units in the cluster and three natural groupings of the stimulus patterns. In this case, the weight vectors for the three units will each migrate toward one of the stimulus groups. (From "Feature Discovery by Competitive Learning" by D. E. Rumelhart and D. Zipser, 1985, *Cognitive Science*, 9, 75-112. Copyright 1985 by Ablex Publishing. Reprinted by permission.)

that the system will find clusters if they are there.

- If the stimuli are highly structured, the classifications are highly stable. If the stimuli are less well structured, the classifications are more variable, and a given stimulus pattern will be responded to first by one and then by another member of the cluster. In our experiments, we started the weight vectors in random directions and presented the stimuli randomly. In this case, there is rapid movement as the system reaches a relatively stable configuration (such as one with a unit roughly in the center of each cluster of stimulus patterns). These configurations can be more or less stable. For example, if the stimulus points do not actually fall into nice clusters, then the configurations will be relatively unstable and the presentation of each stimulus will modify the pattern of responding so that the system will undergo continual evolution. On the other hand, if the stimulus patterns fall rather nicely into clusters, then the system will become very stable in the sense that the same units will always respond to the same stimuli.[4]

- The particular grouping done by a particular cluster depends on the starting value of the weights and the sequence of stimulus patterns actually presented. A large number of clusters, each receiving inputs from the same input lines can, in general, classify the inputs into a large number of different groupings or, alternatively, discover a variety of independent features present in the stimulus population. This can provide a kind of distributed representation of the stimulus patterns.

- To a first approximation, the system develops clusters that minimize within-cluster distance, maximize between-cluster distance, and balance the number of patterns captured by each cluster. In general, tradeoffs must be made among these various forces and the system selects one of these tradeoffs.

### 6.1.3 Implementation

The competitive learning model is implemented in the **cl** program. The model implements a single input (or lower level) layer of units, each connected to all members of a single output (or upper level) layer of units. The basic strategy for the **cl** program is the same as for **bp** and the other learning programs. Learning occurs as follows: A pattern is chosen and the pattern of activation specified by the input pattern is clamped on the input units. Next, the net input into each of the output units is computed. The output unit with the largest input

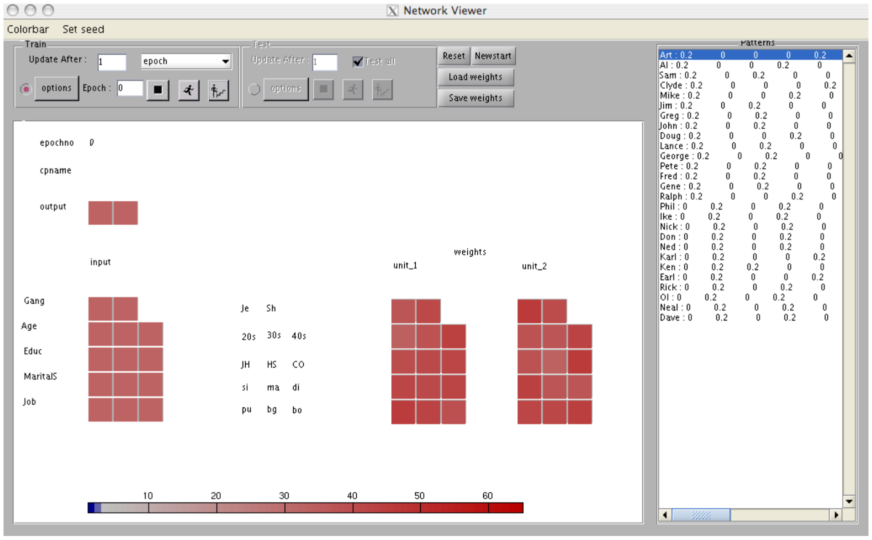
[4] Grossberg (1976) has addressed this problem in his very similar system. He has proved that if the patterns are sufficiently sparse and/or when there are enough units in the cluster, then a system such as this will find a perfectly stable classification. He also points out that when these conditions do not hold, the classification can be unstable. Most of our work is with cases in which there is no perfectly stable classification and the number of patterns is much larger than the number of units in the inhibitory clusters.

is determined to be the winner and its activation value is set to 1. All other units have their activation values set to 0. The routine that carries out this computation is

```
function compute_output()

% intialize the output activation
% ---------------------------------
net.pool(‘output’).activation = zeros(1,noutput);

% compute the netinput for each output unit i
% -------------------------------------------
% ‘netinput’ is a [1 x noutput] array
% ‘weight’ is a [noutput x ninput] matrix
net.pool(‘output’).netinput = net.pool(‘input’).activation * ...
                               net.pool(‘output’).proj(1).weight’;

% find the winner and set its activation to 1
% --------------------------------------------
[maxnet, winindex] = max(net.pool(‘output’).netinput);
net.pool(‘output’).activation(winindex)=1;
net.pool(‘output’).winner=winindex;
end
```

After the activation values are determined for each of the output units, the weights must be adjusted according to the learning rule. This involves increasing the weights from the active input lines to the winner and decreasing the weights from the inactive lines to the winner. This routine assumes that each input pattern sums to 1.0 across units, keeping the total amount of weight equal to 1.0 for a given output unit. If we do not want to make this assumption, the routine could easily be modified by implementing Equation 6.1 instead.

```
function change_weights()

% find the weight vector to be updated (belonging to the winning output unit)
% ------------------------------------------------------------------------
wt = net.pool(‘output’).proj(1).weight(net.pool(‘output’).winner,:);

% adjusting the winner’s weights, assuming that the input
% activation pattern sums to 1
% ---------------------------------------------------------------------
wt = wt + (lrate  .* (net.pool(‘input’).activation - wt));
net.pool(‘output’).proj(1).weight(net.pool(‘output’).winner,:) = wt;
end
```



Update After : 1 epoch

Epoch : 0

Update After : 1 ☑ Test all

epochno 0

```
Art : 0.2       0       0       0       0.2
Al : 0.2        0       0       0.2     0
Sam : 0.2       0       0.2     0       0
Clyde : 0.2     0       0       0       0.2
Mike : 0.2      0       0       0.2     0
Jim : 0.2       0       0.2     0       0
Greg : 0.2      0       0.2     0       0
John : 0.2      0       0.2     0       0
Doug : 0.2      0       0       0.2     0
Lance : 0.2     0       0.2     0       0
George : 0.2    0       0.2     0       0
Pete : 0.2      0       0.2     0       0
Fred : 0.2      0       0.2     0       0
Gene : 0.2      0       0.2     0       0
Ralph : 0.2     0       0       0.2     0
Phil : 0        0.2     0       0.2     0
Ike : 0         0.2     0       0.2     0
Nick : 0        0.2     0       0.2     0
Don : 0         0.2     0       0.2     0
Ned : 0         0.2     0       0.2     0
Karl : 0        0.2     0       0       0.2
Ken : 0         0.2     0.2     0       0
Earl : 0        0.2     0       0       0.2
Rick : 0        0.2     0       0.2     0
Ol : 0          0.2     0       0.2     0
Neal : 0        0.2     0       0.2     0
Dave : 0        0.2     0       0.2     0
```

Figure 6.3: Initial screen display for the **cl** program running the Jets and Sharks example with two output units.

### 6.1.4 Overview of Exercises

We provide an exercise for the **cl** program. It uses the Jets and Sharks data base to explore the basic characteristics of competitive learning.

## Ex6.1. Clustering the Jets and Sharks

The Jets and Sharks data base provides a useful context for studying the clustering features of competitive learning. There are two scripts, **jets2** and **jets3**, where the 2 or 3 in the name indicates that the network has an output cluster of 2 or 3 units. The file *jets.pat* contains the feature specifications for the 27 gang members. The pattern file is set up as follows: The first column contains the name of each individual. The next two tell whether the individual is a Shark or a Jet, the next three columns correspond to the age of the individual, and so on. Note that there are no inputs corresponding to name units; the name only serves as a label for the convenience of the user. To run the program type

*jets2*

The resulting screen display (shown in Figure 6.3) shows the epoch number, the name of the current pattern, the output vector, the inputs, and the weights from the input units to each of the output units. Between the inputs and the weights is a display indicating the labels of each feature.

The inputs and weights are configured in a manner that mirrors the structure of the features. In this case, the pattern for Art is the current pattern, and patterns sum to 1 across the input units. The first row of inputs indicate the gang to which the individual belongs. In the case of Art, we have a .2 on the left and a 0 on the right. This represents the fact that Art is a Jet and not a Shark. Note that there is at most one .2 in each row. This results from the fact that the values on the various dimensions are mutually exclusive. Art has a .2 for the third value of the *Age* row, indicating that Art is in his 40s. The rest of the values are similarly interpreted. The weights are in the same configuration as the inputs. The corresponding weight value is displayed below each of the two output unit labels (*unit_1* and *unit_2*). Each cell contains the weight from the corresponding input unit to that output unit. Thus the upper left-hand value for the weights is the initial weight from the Jet unit to output unit 1. Similarly, the lower right-hand value of the weight matrix is the initial weight from *bookie* to unit 2. The initial values of the weights are random, with the constraint that the weights for each unit sum to 1.0. (Due to scaling and roundoff, the actual values displayed should sum to a value somewhat less than 1.0.) The *lrate* parameter is set to 0.05. This means that on any trial 5% of the winner's weight is redistributed to the active lines.

Now try running the program by clicking the run button in the train window. Since *nepochs* is set to 20, the system will stop after 20 epochs. Look at the new values of the weights. Try several more runs, using the *newstart* command to reinitialize the system each time. In each case, note the configuration of the weights. You should find that usually one unit gets about 20% of its weight on the *jets* line and none on the *sharks* line, while the other unit shows the opposite pattern.

Q.6.1.1.

> What does this pattern mean in terms of the system's response to each of the separate patterns? Explain why the system usually falls into this pattern.

*Hint.*

> You can find out how the system responds to each subpattern by stepping through the set of patterns in test mode — noting each time which unit wins on that pattern (this is indicated by the output activation values displayed on the screen).

Q.6.1.2.

> Examine the values of the weights in the other rows of the weight matrix. Explain the pattern of weights in each row. Explain, for example, why the unit with a large value on the *Jet* input line has the largest weight for the 20s value of age, whereas the unit with a large value on the *Shark* input line has its largest weight for the 30s value of the age row.

Now repeat the problem and run it several more times until it reaches a rather different weight configuration. (This may take several tries.) You might be able to find such a state faster by reducing *lrate* to a smaller value, perhaps 0.02.

Q.6.1.3.

> Explain this configuration of weights. What principle is the system now using to classify the input patterns? Why do you suppose reducing the learning rate makes it easier to find an unusual weight pattern?

We have prepared a pattern file, called *ajets.pat*, in which we have deleted explicit information about which gang the individuals represent. Load this file by going to Train options / Pattern file and clicking "Load new." The same should be done for Test options.

Q.6.1.4.

> Repeat the previous experiments using these patterns. Describe and discuss the differences and similarities.

Thus far the system has used two output units and it therefore classified the patterns into two classes. We have prepared a version with three output units. First, *close* the pdptool windows. Then access the program by the command:

> *jets3*

Q.6.1.5.

> Repeat the previous experiments using three output units. Describe and discuss differences and similarities.

## 6.2 SELF-ORGANIZING MAP

A simple modification to the competitive learning model gives rise to a powerful new class of models: the Self-Organizing Map (SOM). These models were pioneered by Kohonen (1982) and are also referred to as Kohonen Maps.

The SOM can be thought of as the simple competitive learning model with a neighborhood constraint on the output units. The output units are arranged in a spatial grid; for instance, 100 output units might form a 10x10 square grid. Sticking with the hypersphere analogy (Figure 6.2), instead of just moving the winning output unit weights towards the input pattern, the winning unit and its neighbors in the grid are adjusted. The amount of adjustment is determined by the distance in the grid of a given output unit from the winning unit. The effect of this constraint is that neighboring output units tend to respond to similar input pattern, producing a topology preserving map (also called a topographic map) from input space to the output space. This property can be used to visualize structure in high-dimensional input data.

### 6.2.1 The Model

In terms of the architecture illustrated in Figure 6.1, the SOM model presented here is a layer of input units feeding to a single inhibitory output cluster. Each input unit has a weighted connection to each output unit. However, both the input units and the output units are arranged in two-dimensional grids, adding spatial structure to the network.

Activation of the outputs units is calculated in the much the same way as the simple competitive learning model, with the addition of the winner's spread of activation to neighbors. First, the net input to each output unit is calculated, and the unit with the largest net input is selected as the winner. The activations of the units are then set according to the Gaussian in Equation 6.2.[5]

$$active_o = \frac{1}{2\pi\sigma^2} e^{\frac{-1}{2\sigma^2}((o_x - winner_x)^2 + (o_y - winner_y)^2)} \tag{6.2}$$

where $o_x$ and $o_y$ are the grid x and y coordinates of output unit $o$ and $\sigma^2$ is a spread parameter. The larger the spread, the further learning spreads to the neighbors of the winning unit. The total amount of learning remains approximately constant for different values of $\sigma^2$, which generally starts off larger and is reduced so that global order of the map can be established early during learning.

In contrast to simple competitive learning, all of the network weights are updated for a given input pattern. The weight $w_{ij}$ to output unit $i$ is updated according to Equation 6.3.

$$\Delta w_{ij} = \epsilon active_i (active_j - w_{ij}) \tag{6.3}$$

Thus as with simple competitive learning, the output weights are pulled towards the input pattern. This pull is proportional to the activation of the particular output unit and the learning rate $\epsilon$.

### 6.2.2 Some Features of the SOM

- The simple competitive learning algorithm at the beginning of the chapter was described to cluster input patterns along structurally relevant lines. If there are clusters in the input patterns, the algorithm will usually find them. Also, to a first approximation, the algorithm develops clusters that minimize within-cluster distance, maximize between-cluster distance, and balance the number of patterns captured by each cluster — striking some form of tradeoff between these constraints. The SOM still tries to satisfy these constraints, along with the additional constraint that neighboring output units should cluster similar types of inputs. Thus, the structure found by the map not only reflects input clusters but also attempts to preserve the topology of those clusters.

[5] This is the 2D Gaussian density function with mean $\begin{pmatrix} winner_x & winner_y \end{pmatrix}$ and covariance matrix $\begin{pmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{pmatrix}$

- In the SOM presented, the net input for an output unit is the inner product of the input vector and the weight vector, as is common for neural networks. This type of model is discussed in Hertz et al. (1991). However, SOMs usually use Euclidean distance between the vectors instead. In this case, the winning output unit would have the smallest distance from the input pattern instead of the largest inner product.

- Hertz et al. (1991) provide a useful analogy, describing the SOM as an "elastic net in input space that wants to come as close as possible to the inputs; the net has the topology of the output array... and the points of the net have the weights as coordinates."

  To illustrate this point, a simple simulation was run using the Euclidean distance version of the SOM. Input patterns were two dimensional, drawn equally often from one of two circular Gaussians. There was a 5x5 grid of output units, with the weights initialized randomly in a tangled mess (Figure 6.4A). After 250 input patterns, the network dragged the output units towards the patterns, but the network was still a bit tangled and did not span the input space well (Figure 6.4B). After 1000 input patterns, the network spread out to cover more of the patterns, partitioning the output units over the two Gaussians (Figure 6.4C). The "elastic net" was evidently stretched in the middle, since inputs in this region were very sparse.

### 6.2.3 Implementation

The SOM is also implemented in the **cl** program. Learning is very similar to simple competitive learning. A pattern is chosen and clamped onto the input units. Using the same routine as simple competitive learning, the output unit with the largest net input is chosen as the winner. Unique to the SOM, the activation of each output unit is set according to the Gaussian function based on distance from the winner. The routine that carries out this computation is:

```
function compute_output()

% same routine as simple competitive learning
% ------------------------------------------
net.pool('output').activation = zeros(1,noutput);
net.pool('output').netinput = net.pool('input').activation * ...
                              net.pool('output').proj(1).weight';
[maxnet, winindex] = max(net.pool('output').netinput);
net.pool('output').activation(winindex)=1;
net.pool('output').winner=winindex;

% get the (x,y) coordinate of the winning output
% unit in the grid
```

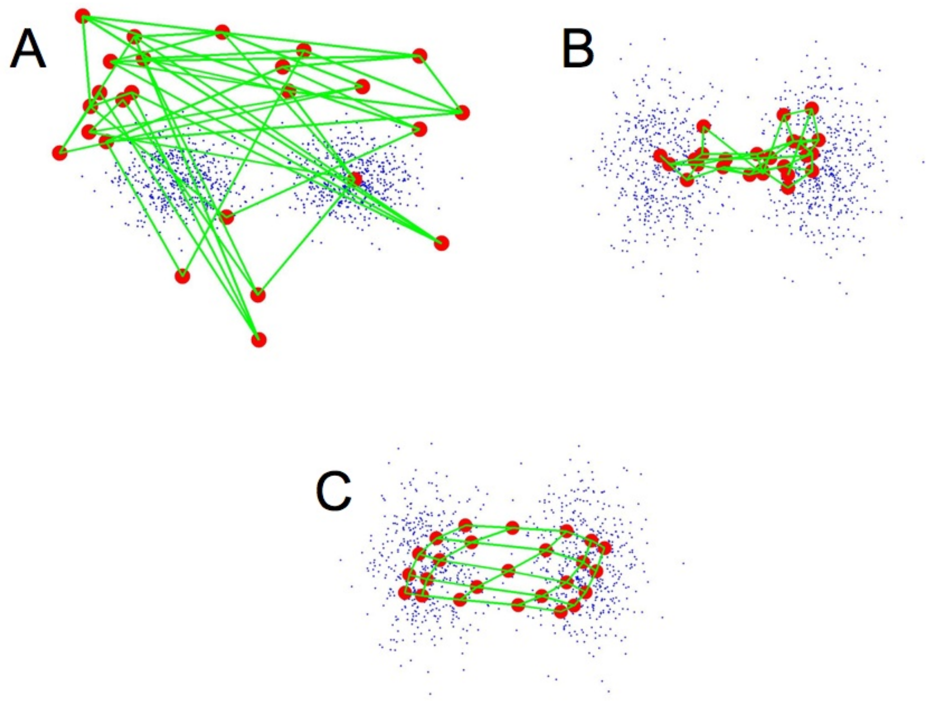


Figure 6.4: The evolution of the network is illustrated at 1 input pattern ($A$), 250 input patterns ($B$), and 1000 input patterns ($C$). In the plots, the blue points are the 1000 input points to be presented. The red points are the weights for each of the 5x5 output units, and adjacent output units are connected by green lines. At initialization in $A$, there is little order, with neighboring output units in the grid spread to opposite ends of the space. After 250 patterns in $B$, the map is compressed between the two Gaussians. Order is starting to emerge since neighboring grid units seem to be nearby in input space. However, coverage of the Gaussians is still poor. In $C$, the output units form a clear grid, illustrating the elastic net analogy. The output units are crowded in the center of each Gaussian where the density of input patterns is concentrated, avoiding the sparse gap between the Gaussians. This illustrates the constraints on the model: concentrating clusters in areas of high density, maximizing between-cluster distance, and retaining the input topology by keeping neighboring output units as neighbors in input space.

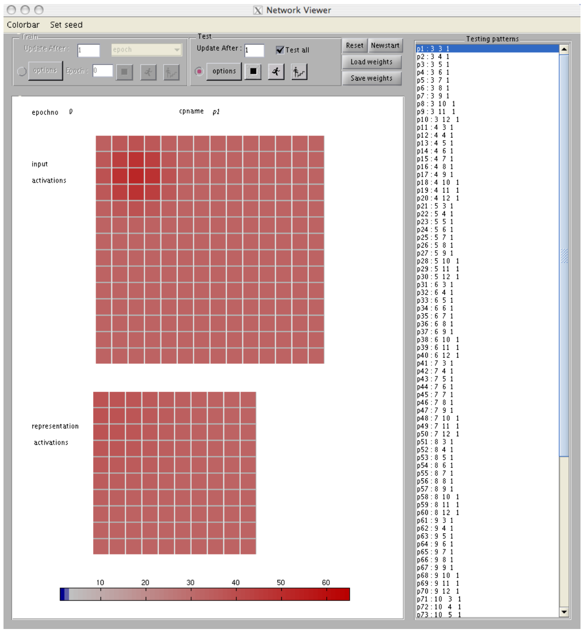
```
% ---------------------------------------------
[xwin, ywin] = ind2sub(net.pool(‘output’).geometry,net.pool(‘output’).winner);

% get the (x,y) coordinates of each output unit in
% the grid, stored in vectors ri and rj
% -------------------------------------------------
[ri,rj]= ind2sub(net.pool(‘output’).geometry,(1:net.pool(‘output’).nunits));

% Gaussian function that distributes activation
% amongst the neighbors of the winner, with
% the spread parameter lrange
% --------------------------------------------
dist = ((xwin - ri) .^ 2 ) + ((ywin - rj) .^ 2);
net.pool(‘output’).activation = exp(-dist ./...
                        (2*lrange^2));
net.pool(‘output’).activation = net.pool(‘output’).activation ./...
                        (2*pi*lrange^2);
end
```

After the activation values are determined for each of the units, the weights are updated. In contrast with simple competitive learning, not just the winner's weights are updated. Each of the output units is pulled towards the input pattern in proportion to its activation and the learning rate. This is done with the following routine:

```
function change_weights()

% get the weight matrix, which has dimensions [noutput x ninput]
% --------------------------------------------------------------------
wt = net.pool(‘output’).proj(1).weight;

% for each output unit, in proportion to the activation of that output unit,
% adjust the weights in the direction of the input pattern
% ---------------------------------------------------------------------
for k =1 :size(wt,1)
   wt(k,:) = wt(k,:) + (lrate .* (net.pool(‘output’).activation(k)*...
    (net.pool(‘input’).activation - wt(k,:))));
end
net.pool(‘output’).proj(1).weight = wt;
end
```

### 6.2.4 Overview of Exercises

We provide an exercise with the SOM in the **cl** program, showing how the SOM could be applied as a model of topographic maps in the brain and illustrating some of the basic properties of SOMs.



Train: Update After : 1 epoch, Epoch : 0

Test: Update After : 1 ☑ Test all

epochno 0 cpname p1

Testing patterns

```
p1 : 3  3  1
p2 : 3  4  1
p3 : 3  5  1
p4 : 3  6  1
p5 : 3  7  1
p6 : 3  8  1
p7 : 3  9  1
p8 : 3  10  1
p9 : 3  11  1
p10 : 3  12  1
p11 : 4  3  1
p12 : 4  4  1
p13 : 4  5  1
p14 : 4  6  1
p15 : 4  7  1
p16 : 4  8  1
p17 : 4  9  1
p18 : 4  10  1
p19 : 4  11  1
p20 : 4  12  1
p21 : 5  3  1
p22 : 5  4  1
p23 : 5  5  1
p24 : 5  6  1
p25 : 5  7  1
p26 : 5  8  1
p27 : 5  9  1
p28 : 5  10  1
p29 : 5  11  1
p30 : 5  12  1
p31 : 6  3  1
p32 : 6  4  1
p33 : 6  5  1
p34 : 6  6  1
p35 : 6  7  1
p36 : 6  8  1
p37 : 6  9  1
p38 : 6  10  1
p39 : 6  11  1
p40 : 6  12  1
p41 : 7  3  1
p42 : 7  4  1
p43 : 7  5  1
p44 : 7  6  1
p45 : 7  7  1
p46 : 7  8  1
p47 : 7  9  1
p48 : 7  10  1
p49 : 7  11  1
p50 : 7  12  1
p51 : 8  3  1
p52 : 8  4  1
p53 : 8  5  1
p54 : 8  6  1
p55 : 8  7  1
p56 : 8  8  1
p57 : 8  9  1
p58 : 8  10  1
p59 : 8  11  1
p60 : 8  12  1
p61 : 9  3  1
p62 : 9  4  1
p63 : 9  5  1
p64 : 9  6  1
p65 : 9  7  1
p66 : 9  8  1
p67 : 9  9  1
p68 : 9  10  1
p69 : 9  11  1
p70 : 9  12  1
p71 : 10  3  1
p72 : 10  4  1
p73 : 10  5  1
```

Figure 6.5: Initial screen display for the **cl** program running the topographic map.

## Ex6.2. Modeling Hand Representation

There are important topology preserving maps in the brain, such as the map between the skin surface and the somatosensory cortex. Stimulating neighboring areas on the skin activate neighboring areas in the cortex, providing a potential application of the SOM model. In this exercise, we apply the SOM to this mapping, inspired by Merzenich's studies of hand representation. Jenkins et al. (1990) studied reorganization of this mapping (between skin surface and cortex) due to excessive stimulation. Monkeys were trained to repeatedly place their fingertips on a spinning, grooved wheel in order to receive food pellets. After many days of such stimulation, Jenkins et al. found enlargement of the cortical representation of the stimulated skin areas. Inspired by this result, a simulation was set up in which:

- Initially quite random (although biased) projections would be organized by experience to create a smooth and orderly topographic map.
- Concentrating exposure in one part of input space, while receiving input deprivation in another part of space, would lead to re-organization of the map. Presumably, the representation will expand in areas of concentrated input.

To run the software:

1. Start MATLAB, make sure the pdptool path is set, and change to the pdptool/cl directory.
2. At the Matlab prompt, type “topo.” This will bring up two square arrays of units, the upper one representing an input layer (like the skin surface) and the lower one representing an internal representation (like the cortical sheet) This window is displayed in Figure 6.5.
3. Start by running a test to get your bearings. Note that there are training and testing windows, train on the left and testing on the right. To test, click the selector button next to ‘options’ under test. Then select test all (so that it is checked) and click run.

The program will step through 100 input patterns, each producing a blob of activity at a point in the input space. The edges of the input space are used only for the flanks of the blobs, their centers are restricted to a region of 10x10 units. The centers of the blobs will progress across the screen from left to right, then down one row and across again, etc. In the representation layer you will see a large blob of activity that will jump around from point to point based on the relatively random initial weights (more on this in Part 3).

Note that the input patterns are specified in the pattern file with a name, the letter x, then three numerical entries. The first is the x position on the input grid ($pat_x$), the second is the y position ($pat_y$), and the third is a spread parameter ($\sigma$), determining the width (standard deviation) of the Gaussian blob. All spreads have been set to 1. The activation of an input unit $i$, at grid coordinates ($i_x$,$i_y$), is determined by:

$$active_i = \frac{1}{2\pi\sigma^2} e^{\frac{-1}{2\sigma^2}((i_x - pat_x)^2 + (i_y - pat_y)^2)} \tag{6.4}$$

which is the same Gaussian function (Equation 6.2) that determines the output activations, depending on the winning unit’s grid position.

The pool structure of the network is as follows:

Pool(1) is not used in this model.
Pool(2) is the input pool.
Pool(3) is the representation pool.

There is only one projection in the network, net.pool(3).proj(1), which contains the weights in the network.

### Part 1: Training the Network

Now you are ready to try to train the network. First, type “net.pool(3).lrange = 2.8” in the Command Window, to set the output activation spread to be initially wide.

To begin training, select the training panel (click the button next to options under train). The network is set up to run 200 epochs of training, with a learning rate (lrate) of .1. The “ptrain” mode is set, meaning that you will be training the network with the patterns presented in a randomly permuted order within each epoch (each pattern is presented once per epoch). The display will update once per epoch, showing the last pattern presented in the epoch in the display. You can reduce the frequency of updating if you like to, say, once per 10 or 20 epochs in the update after window.

Now if you test again, you may see some order beginning to emerge. That is, as the input blob progresses across and then down when you run a test all, the activation blob will also follow the same course. It will probably be jerky and coarse at the point, and sometimes the map comes out twisted. If it is twisted at this stage it is probably stuck.

If it is not twisted, you can proceed to refining the map. This is done by a process akin to annealing, in which you gradually reduce the lrange variable. A reasonable choice reducing it every 200 epochs of training in the following increments: 2.8 for the first 200 epochs, 2.1 for the second 200 epochs, 1.4 for the third 200 epochs, and 0.7 for the last 200 epochs. So, you've trained for 200 epochs as lrange = 2.8, so set "net.pool(3).lrange = 2.1."

Then, run 200 more epochs (just click run) and test again. At this stage the network seems to favor the edges too much (a problem that lessens but often remains throughout the rest of training). Then set net.pool(3).lrange to 1.4 at the command prompt; then run another 200 epochs, then test again, then set it to 0.7, run another 200, then finally test again.

You may or may not have a nice orderly net at this stage. To get a sense of how orderly, you can log your output in the following manner. In test options, click "write output" then "set write options." Click "start new log" and use the name "part1_log.mat." Click "Save" and you will return to the set write output panel. In this panel, go into network variables, click net, it will open, click pool(3), it will open, click "winner" in pool(3), then click "add." The line "pool(3).winner" will then appear under selected. Click "OK." NOTE: you must also click OK on the Testing options popup for the log to actually be opened for use.

Now run a test again. The results will be logged as a vector showing the winners for each of the 100 input patterns. At the matlab command window you can now load this information into matlab:

```
mywinners = load('part1_log');
```

Then if you type (without forgetting the transpose operator '):

```
reshape(mywinners.pool3_winner,10,10)'
```

you will get a 10x10 array of the integer indexes of the winners in your command window. The numbers in the array correspond to the winning output unit. The output unit array (the array of colored squares you see on the gui) is column major, meaning that you count vertically 1-10 first and then 11 starts from the next column, so that 1, 11, 21, 31, 41 etc. are on the same horizontal line. In the matrix printed in your command window, the spatial position corresponds to position of the test pattern centers. Thus, a perfect map will be numbered down then across such that it would have 1-10 in the first column, 11-21 in the second column, etc.

$$\begin{pmatrix} 1 & 11 & 21 & 31 & \dots \\ 2 & 12 & 22 & 32 & \dots \\ 3 & 13 & 23 & 33 & \dots \\ 4 & 14 & 24 & 34 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

** The above array is your first result. Bring this (in printed form) to class for discussion. If your results are not perfect, which is quite likely, what is “wrong” with yours?

NOTE: The log currently stays open and logs all subsequent tests until you shut it off. To do this, click “test options” / “set write options” / and the click “log status off.” You should probably start a new log file each time you want to examine the results, since the contents of the log file will require parsing otherwise. Also the file will then be available to reload and you can examine its contents easily.

### Part 2: Changing the seed

The results depend, as in the necker cube, on the value of the random seed. Try the network again with a seed that works pretty well. Go to the very top left of the network viewer, click “set seed,” enter “211579,” then click ok. Then in the upper right find “Reset” and click that. Just setting the seed does not reset the weights. You must click “Reset.” Note that “Reset” resets the system without changing the seed; if you click “Newstart” you will get a different seed.

To save time during training, we have provided a script to automate the process of reducing the lrange parameter. It is provided in “auto_som.m,” and is run by typing “auto_som” in the command prompt.

After training, *save* your weights after training, for the purposes of Part 4. To do this, click on the “Save weights” button in the upper right of the window. You will load these weights later in this exercise, so remember the folder and name you chose.

After training, the results should be pretty good. Go through the logging steps above, call your log file “part2_log.mat,” and display your results.

**The above array is your second result. Bring this also to class for discussion.

### Part 3: Topographic bias in the initial weights

The neighborhood function causes the winning output unit and its neighbors to move their weights in the same direction, developing the property that neighboring output units respond to similar stimuli. However, for the input grid to align properly with the output grid, there must be some underlying topographic bias in the initial weights. Otherwise, the neighborhood function might help create an orderly response grid, but rotated 90 degrees, twisted, or perhaps worse, a jumbled mess. We will explore this initial topographic weight bias in this exercise.

Note that when the network is initialized, the weights are assigned according to the explanation below:

> Create a set of weights, weight_rand(r,i), which are drawn randomly from a uniform distribution from 0 to 1. Then, normalize the weights such that
>
> ```
> sum(weight_rand(r,:)) = 1
> ```
>
> for any output unit r.
>
> Create a set of weights, weight_topo(r,i), such that
>
> ```
>  weight_topo(r,i) = exp(-d/(2*net.pool(i).proj(j).ispread^2)) ...
> ./ (2*pi*net.pool(i).proj(j).ispread^2)
> ```
>
> where ‘d’ is the distance between unit r and unit i in their respective grids (aligned such that the middle 10x10 square of the input grid aligns with the 10x10 output grid). Thus, the weights have a Gaussian shape, such that the input units connect most strongly with output units that share a similar position in their respective grids. Also,
>
> ```
>  sum(weight_topo(r,:)) = 1 %approximately
> ```
>
> due to the Gaussian function.
>
> Then set the initial weights to:
>
> ```
> net.pool(3).proj(1).weight(r,i) =
>    (1. - topbias).* weight_rand(r,i) +
>    topbias .* weight_topo(r,i);
> ```

Note that if topbias is 0 there is no topographic bias at all in the initial weights, and the weights are random. On the other hand if topbias is 1 the weights are pre-initialized to have a clear topographic (Gaussian) shape, governed by standard deviation “ispread” (stands for “initial spread of weights”). These parameters are associated with net.pool(3).proj(1), and their values are:

```
% we make the problem hard initially!
 net.pool(3).proj(1).topbias = .1
% One standard deviation of the Gaussian covers 4 units,
% so there’s an initially wide spread of the connections.
 net.pool(3).proj(1).ispread = 4
```

Q.6.2.1.

> Set net.pool(3).proj(1).topbias = 1 in the Matlab console and click "reset," and then run through the test patterns without any training. How is the network responding, and why is this the case? Be brief in your response, and there is no need to log the result.

Q.6.2.2.

> Set net.pool(3).proj(1).topbias = 0 in the Matlab console, set the seed to 211579 (as in Part 2), and click "reset." Now, run the standard training regimen (the auto_som.m file). How is the network responding? Do some adjacent input patterns activate adjacent response units?

** Bring these printed answers to class.

### Part 4: Map reorganization through modified stimulation

The purpose of Part 4 is to demonstrate that after the map reaches a reasonable and stable organization, the map can be reorganized given a different distribution of inputs, like the monkey finger tips in Jenkins et al. (1990), with expanded representation in areas of high stimulation.

In this final part of the homework, a developed map will be selectively stimulated in the bottom half of the input space, receiving no input in the upper half. This, of course, is not analogous to the experiment, since the monkeys would also receive stimulation in other areas of the hand not in contact with the wheel. This exercise is rather a simplification to suggest how reorganization is possible in the SOM.

First, load the weights for the trained network that you saved in Part 2. Do this by clicking the Load weights button and select your saved weights. After the weights are loaded, do a test cycle to ensure that the map is already trained.

After this, go into training options. Where it says Pattern file: Click "Load new." Select "topo_half.pat." In case you want to switch the patterns back, topo.pat is the standard pattern file that you have been using.

Set net.pool(3).lrange = 0.7, and train the network for 400 epochs. After training, test the network. Changing the patterns before only applies to the training set, and you will want to change the test patterns as well. Thus, go to test options, and click "Load New," and select topo_half.pat. Also, start a new test log as described in Part 1 and create the log "part4_log.mat." Test the patterns and observe how the network now responds to this selected set of lower-half test patterns. Then type (without forgetting the transpose operator after the second line):

```
mywinners = load('part4_log');
reshape(mywinners.pool3_winner,10,5)'
```

This grid corresponds to only the winners for the bottom 50 input patterns. The first row was most telling of the reorganization. Compare it with the 6th row of your saved array from Part 2. The perfect map in Part 2 would have this row numbered 6, 16, 26, . . . , 96. If the ones digits in the row are numbers other than 6, this would be indicative of reorganization. Specifically, if the ones digits are 4 or 5, that means the representation of this input row has creeped upwards, taking over territory that would previously have responded to the upper half of the patterns.

** This array is your third result. Bring this also to class for discussion.

NOTE: You can either test with the same patterns you train with, or with the original set of 100 patterns. The program generally allows different sets of patterns for training and testing.

# Chapter 7

# The Simple Recurrent Network: A Simple Model that Captures the Structure in Sequences

Since the publication of the original pdp books (Rumelhart et al., 1986; McClelland et al., 1986) and back-propagation algorithm, the **bp** framework has been developed extensively. Two of the extensions that have attracted the most attention among those interested in modeling cognition have been the Simple Recurrent Network (SRN) and the recurrent back-propagation (RBP) network. In this and the next chapter, we consider the cognitive science and cognitive neuroscience issues that have motivated each of these models, and discuss how to run them within the PDPTool framework.

## 7.1 BACKGROUND

### 7.1.1 The Simple Recurrent Network

The Simple Recurrent Network (SRN) was conceived and first used by Jeff Elman, and was first published in a paper entitled *Finding structure in time* (Elman, 1990). The paper was ground-breaking for many cognitive scientists and psycholinguists, since it was the first to completely break away from a prior commitment to specific linguistic units (e.g. phonemes or words), and to explore the vision that these units might be emergent consequences of a learning process operating over the latent structure in the speech stream. Elman had actually implemented an earlier model in which the input and output of the network was a very low-level spectrogram-like representation, trained using a spectral information extracted from a recording of his own voice saying 'This is

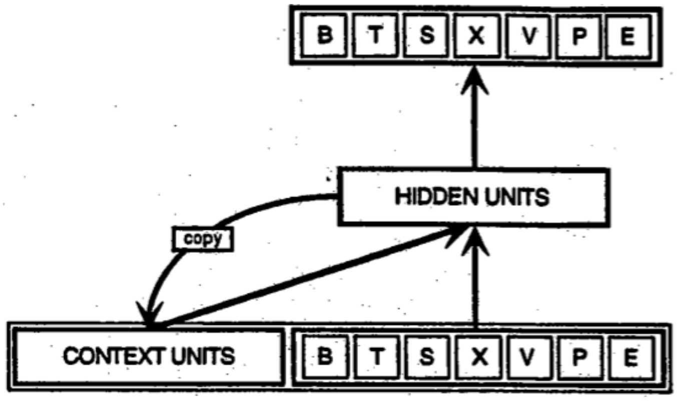


Figure 7.1: The SRN network architecture. Each box represents a pool of units and each forward arrow represents a complete set of trainable connections from each sending unit to each receiving unit in the next pool. The backward arrow, from the hidden layer to the context layer denotes a copy operation. To see how this architecture is represented in the PDPTool implementation, see Figure 7.7. Reprinted from Figure 2, p. 163, of Servan-Schreiber et al. (1991).

the voice of the neural network'. We will not discuss the details of this network, except to note that it learned to produce this utterence after repeated training, and contained no explicit feature, phoneme, syllable, morpheme, or word-level units.

In Elman's subsequent work he stepped back a little from the raw-stimulus approach used in this initial unpublished simulation, but he retained the fundamental commitment to the notion that the real structure is not in the symbols we as researchers use but in the input stream itself. In *Finding structure in time*, Elman presented several simulations, one addressing the emergence of words from a stream of sub-lexical elements (he actually used the letters making up the words as the elements for this), and the other addressing the emergence of sentences from a stream of words. In both models, the input at any given time slice comes from a small fixed alphabet; interest focuses on what can be learned in a very simple network architecture, in which the task posed to the network is to predict the next item in the sting, using the item at time t, plus an internal representation of the state of a set of hidden units from the previous time step.

An SRN of the kind Elman employed is illustrated in Figure 7.1. We actually show the network used in an early follow-up study by Servan-Schreiber et al. (1991), in which a very small alphabet of elements is used (this is the particular network provided with the PDPTool software, and it will be described in more detail later).

The beauty of the SRN is its simplicity. In fact, it is really just a three-layer, feed-forward back propagation network. The only proviso is that one of the two parts of the input to the network is the pattern of activation over the network's own hidden units at the previous time step.

In Elman's simulations, input to the network consisted of an unbroken stream of tokens from the alphabet. The simulation starts with small random connection weights in all feed-forward projections. The first token in the stream is presented, typically setting the values of the units in the input layer to a pattern of ones and zeros. Activation is propagated forward, as in a standard feed-forward back-propagation network. The teaching input or target pattern for the output layer is simply the next item in the stream of tokens. The output is compared to the target, delta terms are calculated, and weights are updated before stepping along to the next item in the stream. Localist input and output patterns are often used in SRN's, because this choice makes it easy from the modeler to understand the network's output, as we shall see later. However, success at learning and discovery of useful and meaningful internal representations does not depend on the use of localist input and output representations, as Elman showed in several of his simulations.

After processing the first item in the stream, the critical step—the one that makes the SRN a type of recurrent network—occurs. The state of the hidden units is 'copied back' to the context units, so that it becomes available as part of the input to the network on the next time step. The arrow labeled 'copy' in the figure represents this step. Perhaps it is worth noting that this copy operation is really just a conceptual convenience; one can think of the context units as simply providing a convenient way of allowing the network to rely on the state of the hidden units from the previous time step.

Once the copy operation is completed, the input sequence is stepped forward one step. The input now becomes the item that was the target at the previous time step, and the target now becomes the next item in the sequence. Activation is propagated forward, the output is compared to the target, delta terms are calculated via back-propagation, and weights are updated. Note that the weights that are updated include those from the context (previous hidden state) to the hidden units (current hidden state). This allows the network to learn, not only to use the input, but also to use the context, to help it make predictions.

In most of Elman's investigations, he simply created a very long stream of tokens from the alphabet, according to some generative scheme, and then repeatedly swept through it, wrapping around to the beginning when he reached the end of the sequence. After training with many presentations, he stopped training and tested the network.

Here we briefly discuss three of the findings from Elman (1990). The first relates to the notion of 'word' as linguistic unit. The concept 'word' is actually a complicated one, presenting considerable difficulty to anyone who feels they must decide what is a word and what is not. Consider these examples: 'line drive', 'flagpole', 'carport', 'gonna', 'wanna', 'hafta', 'isn't' and 'didn't' (often pronounced "dint"). How many words are involved in each case? If more than one word, where are the word boundaries? Life might be easier if we did not have to decide where the boundaries between words actually lie. Yet, we have intuitions that there are points in the stream of speech sounds that correspond

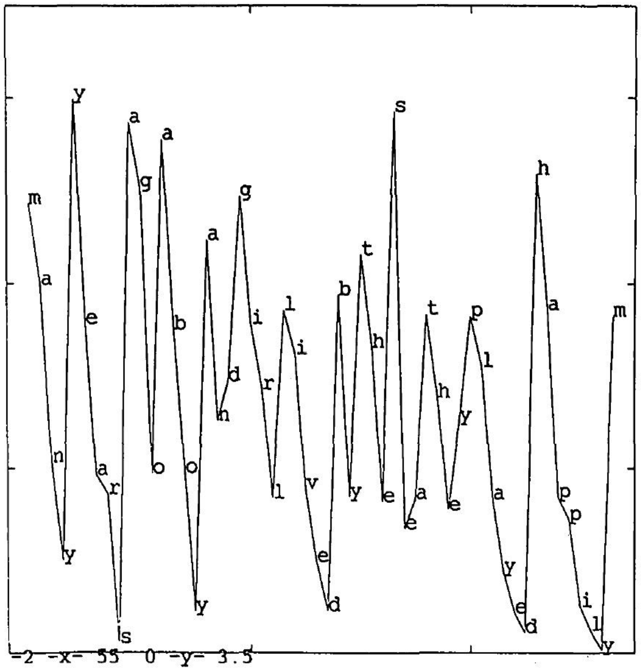


Figure 7.2: Root mean squared error in predicting each of the indicated letters from Elman's letter-in-word prediction experiment. The letters shown are the first 55 letters in the text used for training the network. Reprinted from Figure 6, p. 194, of Elman (1990).

to places where something ends and something else begins. One such place might be between 'fifteen' and 'men' in a sentence like 'Fifteen men sat down at a long table', although there is unlikely to be a clear boundary between these words in running speech.

Elman's approach to these issues, as previously mentioned, was to break utternances down into a sequence of elements, and present them to an SRN. In his letter-in-word simulation, he actually used a stream of sentences generated from a vocabulary of 15 words. The words were converted into a stream of elements corresponding to the letters that spelled each of the words, with no spaces. Thus, the network was trained on an unbroken stream of letters. After the network had looped repeatedly through a stream of about 5,000 elements, he tested its predictions for the first 50 or so elements of the training sequence. The results are seen in Figure 7.2.

What we see is that the network tends to have relatively high prediction error for the first letter of each word. The error tends to drop throughout the word, and then suddenly to shoot up again at the first letter of the next word. This is not always true – sometimes, afer a few downward steps, there is an uptick within a word, but such uptics generally correspond to places where there might be the end of what we ordinarily call a word. Thus, the network has learned something that corresponds at least in part with our intuitive notion of 'word', without building in the concept of word or ever making a categorical decision about the locations of word boundaries.

The other two findings come from a different simulation, in which the elements of the sequences used corresponded to whole words, strung together again to form simple sentences. The set of words Elman used corresponded to several familiar nouns and verbs. Each sentence involved a verb, and at least one noun as subject, with an optional subsequent noun as direct object. Verbs and nouns fell into different sub-types, – there were, for example, verbs of perception (which require an animate subject but can take any noun as object) and verbs of consumption, which require something consumable, and verbs of descruction, each of which had different restrictions on the nouns that could occur with it as subject and object. Crucially, the input patterns representing the nouns and verbs were randomly assigned, and thus did not capture in any way the coocurrence structure of the domain. Over the course of learning, however, the network came to assign each input its own internal representation. In fact, the hidden layer reflected both the input and the context; as a result, the patterns the network learned to assign provided a highly context-sensitive form of lexical representation.

The next two figures illustrate findings from this simulation. The first of these (Figure 7.3) shows a cluster analysis based on the average pattern over the hidden layer assigned to each of the different words in the corpus. What we see is that the learned average internal representations indicate that the network has been able to learn the category structure and even the sub-category structure of the "lexicon" of this simple artificial language. The reason for this is largely that the predictive consequences of each word correspond closely to the syntactic category and sub-category structure of the language. One may note, in fact, that the category structure encompasses distinctions that are usually treated as syntactic (noun or verb, and within verbs, transitive vs intransitive) as well as distinctions that are usually treated as semantic (fragile-object, food item), and at least one distinction that is clearly semantic (animate vs. inanimate) but is also often treated as a syntactically relevant "subcategorization feature" in linguistics. The second figure (Figure 7.4) shows a cluster analysis of the patterns assigned two of the words (BOY and GIRL) in each of many different contexts. The analysis establishes that the overall distinction between BOY and GIRL separates the set of context-sensitive patterns into two highly similar subtrees, indicating that the way context shades the representation of BOY is similar to the way in which it shades the representation of GIRL.

Overall, these three simulations from Elman (1990) show how both segmentation of a stream into larger units and assignment of units into a hierarchical similarity structure can occur, without there actually being any enumerated list of units or explicit assignment to syntactic or semantic categories.

Elman continued his work on SRN's through a series of additional important and interesting papers. The first of these (Elman, 1991) explored the

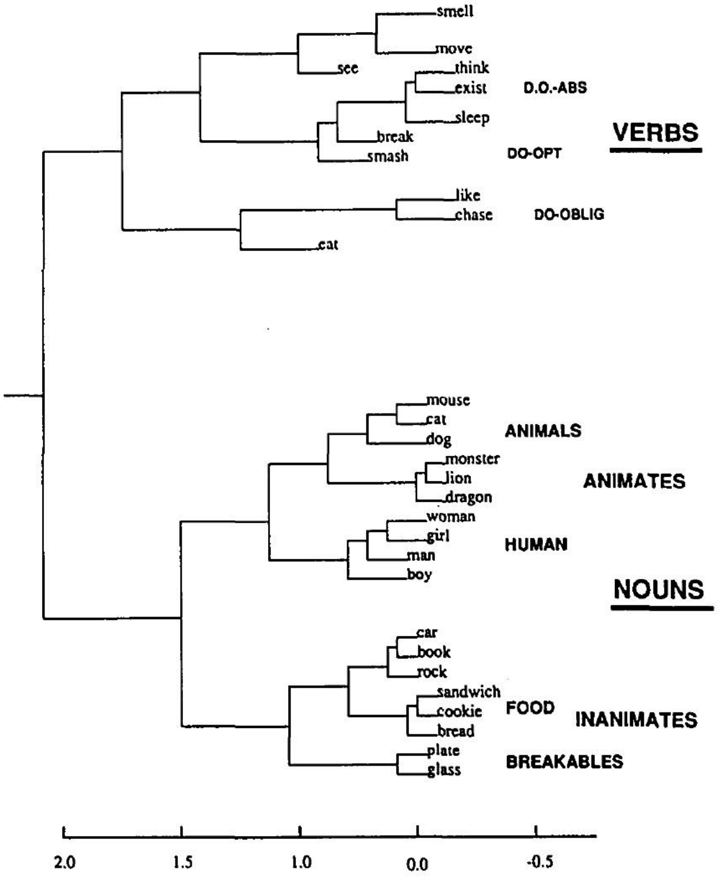


Figure 7.3: Result of clustering the average pattern over the hidden units for each of the words used in Elman's (1990) sentence-structure simulation. Noun and verb categories are cleanly separated. Within nouns, there is strong clustering by animacy, and within animates, by human vs animal; then within animal, by predator vs prey. Inanimates cluster by type as well. Within verbs, clustering is largely based on whether the verb is trainsitive (DO-OBLIG) intransitive (DO-ABS), or both (DO-OPT), although some verbs are not perfectly classified. Reprinted from Figure 7, p. 200, of Elman (1990)

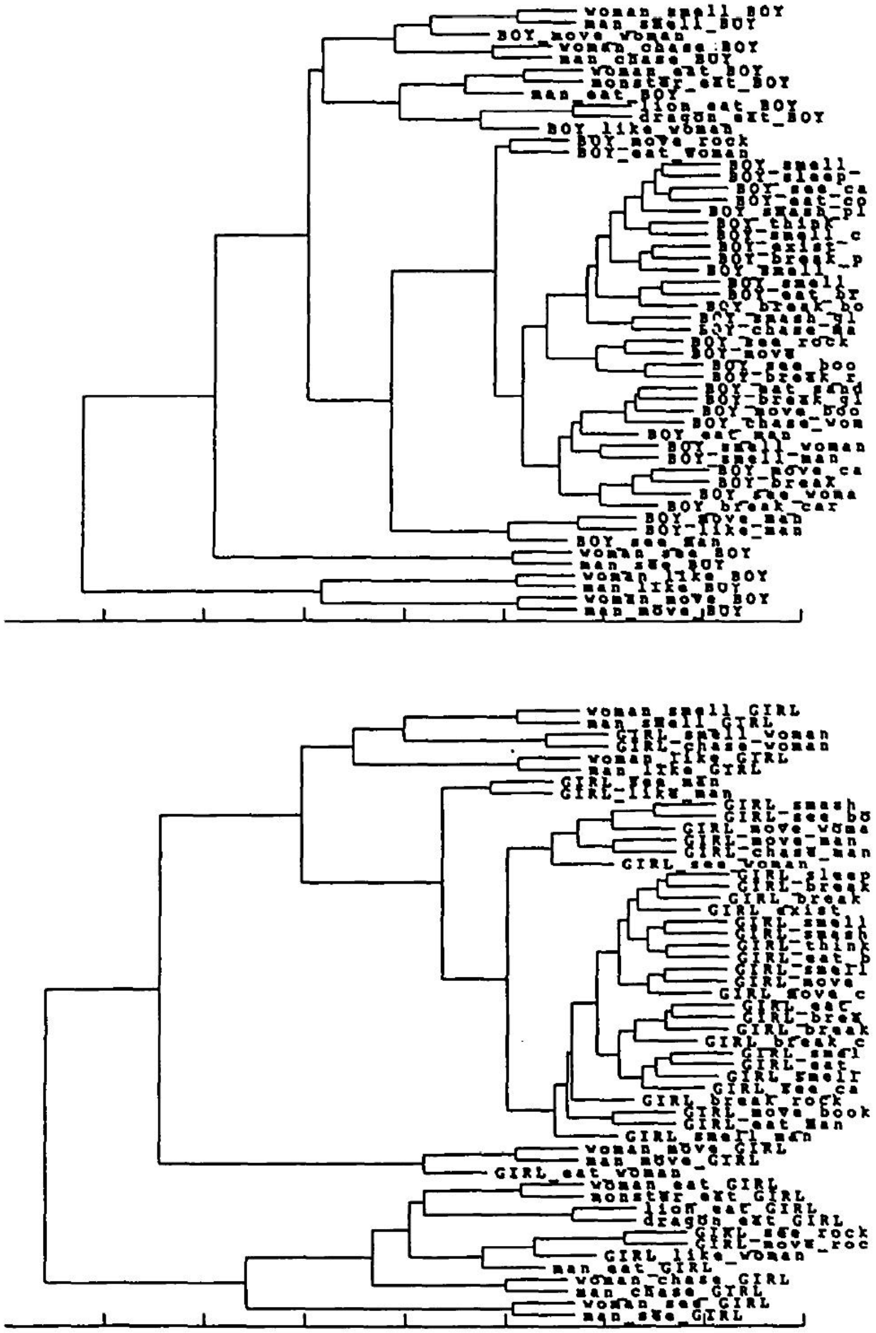


Figure 7.4: Hierarchical cluster diagram of hidden unit activation vectors in response to some occurrences of the inputs BOY and GIRL. Upper-case labels indicate the actual input; lowercase labels indicate the context. Note in both cases the large cluster in the middle corresponding to BOY or GIRL as subject, and the large clusters flanking these above and below correponding primarily to cases in which BOY or GIRL is the sentential object. Reprinted from Figure 9, p. 206, from Elman (1990).

learning of sentences with embedded clauses, illustrating that this was possible in this simple network architecture, even though the network did not rely on the computational machinery (an explicitly recursive computational structure, including the ability to 'call' a computational process from within itself) usually thought to be required to deal with imbeddings. A subsequent and highly influential paper (Elman, 1993) reported that success in learning complex embedded structures depended on starting small – either starting with simple sentences, and gradually increasing the number of complex ones, or limiting the network's ability to exploit context over long sequences by clearing the context layer after every third element of the training sequence. However, this later finding was later revisited by Rohde and Plaut (1999). They found in a very extensive series of investigations that starting small actually hurt eventual performance rather than helped it, except under very limited circumstances. A number of other very interesting investigations of SRN's have also been carried our by Tabor and collaborators, among other things using SRN's to make predictions about participants reading times as they read word-by-word through sentences (Tabor et al., 1997).

### 7.1.2 Graded State Machines

The simple recurrent network introduced by Elman (1990) also spawed investigations by Servan-Schreiber et al. (1991). We describe this work in greater detail than some of the other work following up on Elman (1990) because the exercise we provide is based on the Servan-Schreiber et al. (1991) investigations.

These authors were especially interested in exploring the relationship between SRNs an classical automata, including Finite State Transition Networks, and were also interested in the possibility that SRNs might provide useful models of implicit sequence learning in humans (Cleeremans and McClelland, 1991). Servan-Schreiber et al. (1991) investigated the learning of sequences that could be generated by a simple finite-state transition network grammar of the kind used to generate stimuli used in human implicit learning experiments by Rebur (1976), and illustrated in Figure 7.5. Sequence generation occurs as follows. Each sequence begins with the symbol 'B', and enters the state associated with the node in the diagram marked '#0'. From there, a random choice is made to follow one of the two links leading out of node 0. After the random choice, the symbol written beside the link is added as the second element of the sequence, and the process transitions to the state at the end of the chosen link. The choice could, for example, be the upper link leaving node 0, producing a T and shifting to node 1. The process iterates as before, choosing one of two links out of the current node, adding the symbol written next to the link as the next element of the seqeuence, and shifting to the node at the end of the link. When node 5 is reached, there is only one remaining possibility; the symbol E is written, and the sequence then ends.

In their simulation of learning sequences generated by the Finite State Transition Network, Servan-Schreiber et al. (1991) assumed that the learner is trained with a series of sequences, which may be generated at random by the sequence-

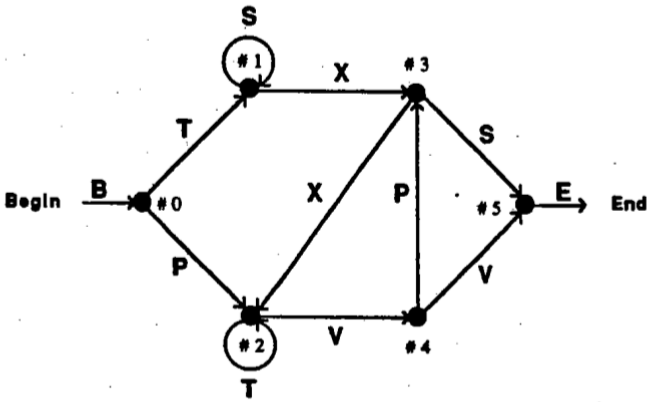


Figure 7.5: The stochastic finite-state transition network used in the **gsm** simulation. Strings are generated by transintioning between nodes connected by links, and emitting the symbol associated with each link. Where two links leave the same node, one is chosen at random with equal probability. Reprinted from Figure 3, p. 60, of Servan-Schreiber et al. (1991), based on the network used earlier by Rebur (1976).

generating finite-state transition network. At the beginning of each sequence, the context representation is reset of a neutral initial state, and the initial element B is presented on the input. The network then tries to predict the next element of the sequence, which then is presented as the target. Small adjustments to the weights are made, and then the process steps forward through the sequence. Now the second element of the sequence (the one after the B) is presented at the the input, and the third item in the sequence becomes the target. The process of propagating activation forward, adjusting the the connections, and stepping forward to the next element continues, until the last element of the sequence (the E symbol) is reached. At that point, processing of the sequence ends. The process then begins again with the next sequence.

In their paper, Servan-Schreiber et al. (1991) demonstrated that a network like the one shown in Figure 7.1 could essentially learn to become the transition network grammar through a completely gradual and continuous learning process. That is, as a result of training, the patterns at the hidden layer came to correspond closely to the nodes of the finite state transition network, in the sense that there was essentially one hidden pattern corresponding to each of the nodes in the network, regardless of the sequence of elements leading to that node. However, the network could also learn to 'shade' its representation of each node to a certain extent with the details of the prior context, so that it would eventually learn to capture subtle idiosyncratic constraints when trained repeatedly on a fixed corpus of legal example sentences. The authors also went on to show that the network could learn to use such subtle shading to carry information forward over an embedded sequence, but this will only happen in the SRN if the material needed at the end of the embedding is also of some relevance within the sequence.

The reader is referred to the paper by Servan-Schreiber et al. (1991) for further details of these investigations. Here we concentrate on describing the simulation model that allows the reader to explore the SRN model, using the same network and one of the specific training sets used by Servan-Schreiber et al. (1991).

## 7.2 THE SRN PROGRAM

The **srn** is a specific type of back-propagation network. It assumes a feed-forward architecture, with units in input, hidden, and output pools. It also allows for a special type of hidden layer called a "context" layer. A context layer is a hidden layer that receives a single special type of projection from another layer containing the same number of units, typically a hidden layer. This special type of projection (called a 'copy-back projection') allows the pattern of activation left over on the sending units from the last input pattern processed to be copied onto the receiving units for use as context in processing the next input pattern.

### 7.2.1 Sequences

The **srn** network type also allows provides a construct called the "sequence", which consists of one or more input-output pairs to be presented in a fixed order.

The idea is that one might experience a series of sequences, such that each sequence has a fixed structure, but the order in which the sequences appear can be random (permuted) within each epoch. In the example provided, a sequence is a sequence of characters beginning with a $B$ and ending with an $E$.

Sequences can be defined in the pattern file in two different ways:

**Default** The *default* method involves beginning the specification of each sequence with a *pname*, followed by a series of input-output pairs, followed by 'end' (see the file gsm21_s.pat for an example). When the file is read, a data structure element is created for the sequence. At the beginning of each sequence, the state of the context units is initialized to all .5's at the same time that the first input pattern is presented on the input. At each successive step through the sequence, the state of the context units is equal to the state of the hidden units determined during the previous step in the sequence.

**SeqLocal** The *SeqLocal* method of specifying a sequence works only for a restricted class of possible cases. These are cases where (a) each input and target pattern involves a single active unit; all other inputs and targets are 0; and (b) the target at step n is the input at step n+1 (except for the last element of the sequence). For such cases, the .pat file must begin with a line like this:

```
SeqLocal  b t s x v p e
```

This line specifies that the following entries will be used to construct actual input output pattern pairs, as follows. The character strings following *SeqLocal* are treated as labels for both the input and output units, with the first label being used for the first input unit and the first output unit etc. Single characters are used in the example but strings are supported.

Specific sequences are then specified by lines like the following:

```
p05 b t x s e
```

Here each line begins with a *pname*, followed by a series of instances of the different labels previously defined. The case shown above generates four input-output pattern pairs; the first input contains a 1 in the first position and 0's elsewhere and the first target contains a 1 in the second position and 0's elsewhere. The second input is the same as the first target, etc. Thus, in this model the user specifies the actual symbolic training sequences and these are translated into actual input and output patterns for the user.

*Using the* ***last target*** *from the previous sequence as the first element if the input for the next sequence.* If the first label in the string of labels is the special label '#', the last target pattern from the previous sequence is used as the first input pattern for the current sequence. In this case, we also copy the state of the hidden units from the network's prediction of the last target to the context layer, instead of resetting the context to the *clearval*.

### 7.2.2 New Parameters

**mu** The srn program introduces one new parameter called *mu*. This parameter specifies a scale factor used to determine how large a fraction of the pattern of activation from the context layer at time step n is added to the pattern copied back from the hidden layer. In the **gsm** simulation, *mu* is set to 0. Non-zero values of *mu* force some trace of the earlier context state to stick around in the current state of the context units.

**clearval** At the beginning of each training and testing sequence, the activations of the context units are set to the value of this parameter, represented as *clear value* in the train and test options windows. By default, this paramter is set to .5. If a negative value is given, the states of the units are not reset at the beginning of a new sequence; instead the state from the last element of the preceding sequence is used.

### 7.2.3 Network specification

The .net file specifies the specific architecture of the model as in other PDPtool programs. To have a pool of units treated as a 'context' layer, it should be of type ‘hidden’ and should receive a single projection from some other layer (typically a hidden layer) which has its constraint_type field set to ‘copyback’. When the constraint type is ‘copyback’, there are no actual weights, the state of the ‘sending’ units is simply copied to the ‘receiving’ units at the same time that the next target input is applied (except at the beginning of a new sequence, where the state of each of the context units is set to the *clearval*, which by default is set to .5).

### 7.2.4 Fast mode for training

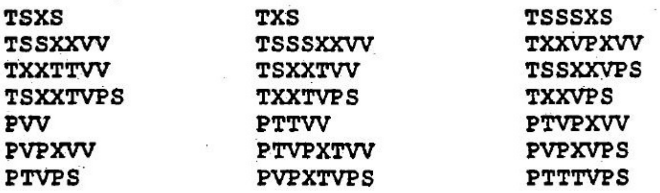
The srn program’s inner loops have been converted to fast mex code (c code that interfaces with MATLAB). To turn on the fast mode for network training, click the ’fast’ option in the train window. This option can also be set by typing the following command on the MATLAB command prompt:
*runprocess(’process’,’train’,’granularity’,’epoch’,’count’,1,’fastrun’,1);*

There are some limitations in executing fast mode of training.In this mode, network output logs can only be written at epoch level. Any smaller output frequency if specified, will be ignored.When running with fast code in gui mode, network viewer will only be updated at epoch level.Smaller granularity of display update is disabled to speed up processing.

There is a known issue with running this on linux OS.MATLAB processes running fast version appear to be consuming a lot of system memory. This memory management issue has not been observed on windows. We are currently investigating this and will be updating the documentation as soon as we find a fix.

## 7.3 EXERCISES

The exercise is to replicate the simulation discussed in Sections 3 and 4 of Servan-Schreiber et al. (1991). The training set you will use is described in more detail in the paper, but is presented here in Figure 7.6. This particular set contains 21 patterns varying in length from 3 to 8 symbols (plus a B at the beginning and an E at the end of each one).

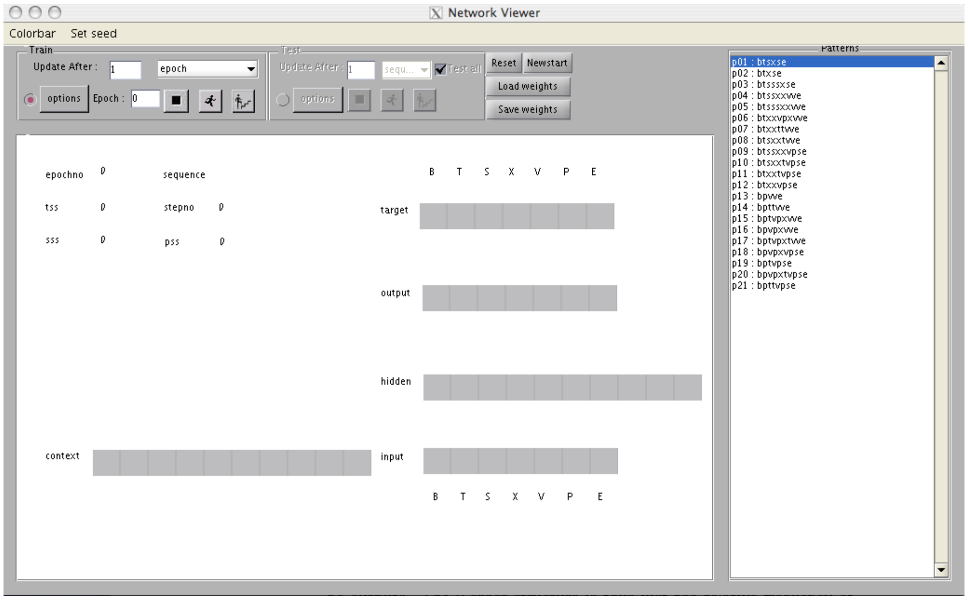


TSXS
TSSXXVV
TXXTTVV
TSXXTVPS
PVV
PVPXVV
PTVPS

TXS
TSSSXXVV
TSXXTVV
TXXTVPS
PTTVV
PTVPXTVV
PVPXTVPS

TSSSXS
TXXVPXVV
TSSXXVPS
TXXVPS
PTVPXVV
PVPXVPS
PTTTVPS

Figure 7.6: The training patterns used in the **gsm** exercise. A *B* is added to the beginning of each sequence and an *E* is added to the end in the **gsm** simulation. Reprinted from Figure 12, p. 173, of Servan-Schreiber et al. (1991).



Train: Update After : 1 epoch
Epoch : 0
Test: Update After 1 sequ... ☑ Test all

epochno 0
sequence
tss 0
stepno 0
sss 0
pss 0

p01 : btsxse
p02 : btxse
p03 : btsssxse
p04 : btssxxvve
p05 : btsssxxvve
p06 : btxxvpxvve
p07 : btxxttvve
p08 : btsxxtvve
p09 : btssxxvpse
p10 : btsxxtvpse
p11 : btxxtvpse
p12 : btxxvpse
p13 : bpvve
p14 : bpttvve
p15 : bptvpxvve
p16 : bpvpxvve
p17 : bptvpxtvve
p18 : bpvpxvpse
p19 : bptvpse
p20 : bpvpxtvpse
p21 : bpttvpse

Figure 7.7: Network view showing the layout for the **srn** network.

To run the exercise, download the latest version of pdptool, set your path to include pdptool and all of its children, and change to the pdptool/srn directory. Type ‘gsm’ (standing for “Graded State Machine”) at the matlab prompt. After everything loads you will see a display showing (at the right of the screen) the input, hidden and output units, and a vector representing the target for the output units (see Figure 7.7). To the left of the input is the context. Within the input, output, and target layers the units are layed out according to Figure 7.1. Your exercise will be to test the network after 10, 50, 150, 500, and 1000 epochs of training. The parameters of the simulation are a little different from the parameters used in the published article (it is not clear what values were actually used in the published article in some cases), and the time course of learning is a little extended relative to the results reported in the paper, but the same basic pattern appears.

One thing to be clear about at the outset is that the training and testing is organized at the sequence level. Each sequence corresponds to a string that could be generated by the stochastic finite state automaton shown in Figure 7.5. The *ptrain* option (which is the one used for this exercise) permutes the order of presentation of sequences but presents the elements of the sequence in its cannonical sequential order. Each sequence begins with a *B* and ends with an *E*, and consists of a variable number N of elements. As describe above, the sequence is broken down into N-1 input-target pairs, the first of which has a *B* as input and the successor of *B* in the sequence as its target, and the last of which has the next to last symbol as its input and *E* as its target. When the $B$ symbol is presented the context is reset to .5's. It makes sense to update the display during testing at the pattern level, so that you can step through the patterns within each sequence. During training I update the display at the epoch level or after 10 epochs.

Before you begin, consider:

- What is the 0-order structure of the sequences? That is, if you had no idea about even the current input, what could you predict about the output? This question is answered by noting the relative frequency of the various outputs. Note that $B$ is never an output, but all the other characters can be outputs. The 0-order structure is thus just the relative frequency of each character in the output.

- What is the 1st order structure of the sequences? To determine this approximately, consult the network diagram (last page of this handout) and note which letters can occur after each letter. Make a little grid four yourself of seven rows each containing seven cells. The row stands for the current input, the cell within a row for a possible successor. So, consulting the network diagram, you will find that $B$ can be followed only by $T$ or $P$. So, place an $X$ in the second ($T$) cell of the first row and the 6th ($P$) cell of the first row. Fill in the rest of the rows, being careful to attend to the direction of the arrows coming out of each node in the diagram and the label on each arc. You should find (unless I made a mistake) that three of the letters actually have the exact same set of possible successors. Check yourself carefully to make sure you got this correct.

OK, now run through a test, before training. You should find that the network produces near uniform output over the output units at each step of testing.

NOTE: For testing, set update to occur after 1 pattern in the test window. Use the single step mode and, in this case, quickly step through, noticing how little the output changes as a function of the input. The weights are initialized in a narrow range, making the initial variation in output unit activation rather tiny. You can examine the activations of the output units at a given point by typing the following to the matlab console:

```
net.pool(5).activation
```

When you get tired or stepping through, hit run in the test window. The program will then quickly finish up the test.

The basic goal of this exercise is to allow you to watch the network proceed to learn about the 0th, 1st, and higher-order structure of the training set. You have already examined the 0th and 1st order structure; the higher-order structure is the structure that depends on knowing something about what happened before the current input. For example, consider the character $V$. What can occur after a $V$, where the $V$ is preceded by a $T$? What happens when the $V$ is preceeded by an $X$? By a $P$? By another $V$? Similar questions can be asked about other letters.

Q.7.0.3.

Set *nepochs* in the training options panel to ten, and run ten epochs of training, then test again.

What would you say has been learned at this point? Explain your answer by referring to the pattern of activation across the output units for different inputs and for the the same input at different points in the sequence.

Continue training, testing after a total of 50, 150, 500, and 1000 epochs. *Answer the same question as above, for each test point point.*

Q.7.0.4.

Summarize the time course of learning, drawing on your results for specific examples as well as the text of section 4 of the paper. How do the changes in the representations at the hidden and context layers contribute to this process?

Q.7.0.5.

Write about 1 page about the concept of the SRN as a graded state machine and its relation to various types of discrete-state automata, based on your reading of the entire article (including especially the section on spanning embedded sequences).

Try to be succinct in each of your answers. You may want to run the whole training sequence twice to get a good overall sense of the changes as a function of experience.

# Chapter 8

# Recurrent Backpropagation: Attractor network models of semantic and lexical processing

Recurrent back-propagation networks came into use shortly after the development of the back-propagation algorithm was first developed, and there are many variants of such networks. Williams and Zipser (1995) provide a thorough review of the recurrent back-propagation computational framework. Here we descibe a particular variant, used extensively in PDP models the effects of brain injury on lexical and semantic processing (Plaut and Shallice, 1993; Plaut et al., 1996; Rogers et al., 2004; Dilkina et al., 2008).

## 8.1 BACKGROUND

A major source of motivation for the use of recurrent backpropagation networks in this area is the intuition that they may provide a way of understanding the pattern of degraded performance seen in patients with neuropsychological deficits. Such patients make a range of very striking errors. For example, some patients with severe reading impairments make what are called semantic errors – misreading APRICOT as PEACH or DAFFODIL as TULIP. Other patients, when redrawing pictures they saw only a few minutes ago, will sometimes put two extra legs on a duck, or draw human-like ears on an elephant.

In explaining these kinds of errors, it has been tempting to think of the patient as having settled into the wrong basin of attraction in a semantic attractor network. For cases where the patient reads 'PEACH' instead of 'APRICOT', the idea is that there are two attractor states that are 'near' each other in a semantic space. A distortion, either of the state space itself, or of the mapping

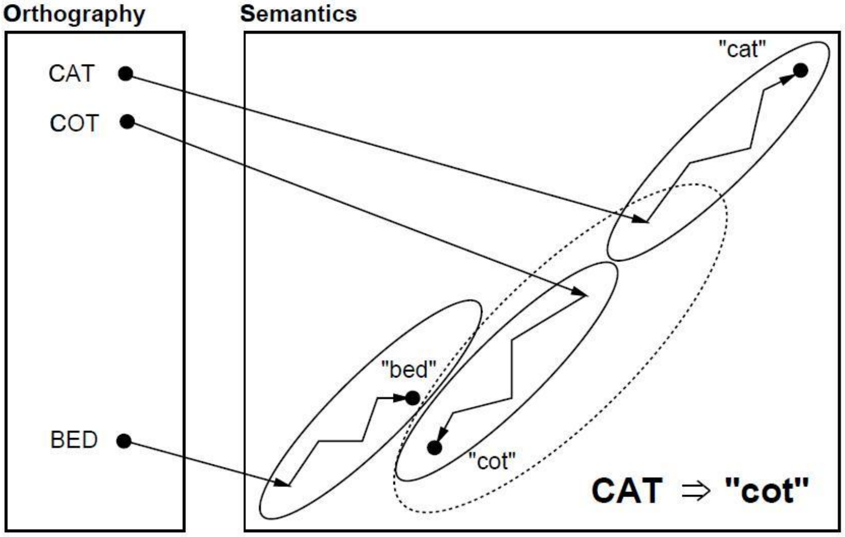


Figure 8.1: Conceptual illustration of a semantic state-space containing basins of attraction (solid ellipses) for $CAT$, $COT$. and $BED$. Distortion (dotted ellipse) of the basin for $COT$ can result in $CAT$ falling into it. A different distortion, not shown, could allow $COT$ to fall into the basin for $BED$. From Figure 4 p. 394 of Plaut and Shallice (1993).

into that space, can result in an input that previously settled to one attractor state settling into the neighboring attractor. Interestingly, patients who make these sorts of semantic errors also make visual errors, such as misreading 'cat' as 'cot', or even what are called 'visual-then-semantic' errors, mis-reading 'sympathy' as 'orchestra'. All three of these types of errors have been captured using PDP models that rely on the effects of damage in networks containing learned semantic attractors (Plaut and Shallice, 1993). Figure 8.1 from Plaut and Shallice (1993) illustrates how both semantic and visual errors can occur as a result of damage to an attractor network that has learned to map from orthography (a representation of the spelling of a word) to semantics (a represenation of the word's meaning), taking printed words and mapping them to basins of attraction within a recurrent semantic network.

The use of networks with learned semantic attractors has an extensive history in work addressing semantic and lexical deficits, building from the work of Plaut and Shallice (1993) and other early work (Farah and McClelland, 1991; Lambon Ralph et al., 2001). Here we focus on a somewhat more recent model introduced to address a progressive neuropsychological condition known as *se-*

*mantic dementia* by Rogers et al. (2004). In this model, the 'semantic' representation of an item is treated as an attractor state over an population of neurons thought to be located in a region known as the 'temporal pole' or anterior temporal lobe. The neurons in this integrative layer receive input from, and project back to, a number of different brain regions, each representing a different type of information about an item, including what it looks like, how it moves, what it sounds like, the sound of its name, the spelling of the word for it, etc. The architecture of the model is sketched in Figure 8.2 (top). Input coming to any one of the visible layers can be used to activate the remaining kinds of information, via the bi-directional connections among the visible layers and the integrative layer and the recurrent connections among the units in the integrative layer. According to the theory behind the model, progressive damage to the neurons in the integrative layer and/or to the connections coming into and out of this integrative layer underlies the progressive deterioration of semantic and abilities in semantic dementia patients (**?**).

## 8.2 THE RBP PROGRAM

In the version of recurrent back propagation that we consider here, activations of units are thought of as being updated continuously over some number of time intervals. The network is generally set to an initial state (corresponding to time 0), in which some units are clamped to specific values by external inputs, while others are initialized to a default starting value. Processing then proceeds in what is viewed conceptually as a continuous process for the specified number of time intervals. At any point along the way, target patterns can be provided for some of the pools of units in the network. Typically, these networks are trained to settle to a single final state, which is represented by a target to be matched over a subset of the pools in the network, over the last few time intervals.

At the end of the forward activation process, error signals are "back propagated through time" to calculate delta terms for all units in the network across the entire time span of the settling process. A feature of recurrent backpropagation is that the same pool of units can have both external inputs (usually provided over the first few time intervals) and targets (usually specified for the last few intervals). These networks are generally thought of as settling to attractor states, in which there is a pattern of activation over several different output pools. In this case, the target pattern for one of these pools might also be provided as the input. This captures the idea that I could access all aspects of my concept of, for example, a clock, either from seeing a clock, hearing a clock tick, hearing or reading the word clock, etc.

One application of these ideas is in the semantic network model of Rogers et al. (2004). A schematic replica of this model is shown in Figure 8.2 (top). There are three sets of visible units, corresponding to the name of the object, other verbal information about the object, and the visual appearance of the object. Whichever pattern is provided as the input, the task of the network is to settle to the complete pattern, specifying the name, the other verbal information about the object, and the visual percept. Thus the model performs pattern completion, much like, for example, the Jets-and-Sharks **iac** model from Chapter 2. A big difference is that the Rogers et al. (2004) model uses learned distributed representations rather than instance units for each known concept.

### 8.2.1 Time intervals, and the partitioning of intervals into ticks

The actual computer simulation model, called **rbp**, treats time, as other networks do, as a sequence of discrete steps spanning some number of cannonical time intervals. The number of such intervals is represented by the variable *nintervals*. In **rbp**, the discrete time steps, called *ticks*, can slice the whole span of time very finely or very coursely, depending on the value of a parameter called *dt*. The number of ticks per interval is $1/dt$, and the total number of ticks of processing is equal to $nintervals/dt$. The number of states of the network is one larger than the number of ticks; there is an initial state, at time 0, and a state at the end of each tick — so that the state at the end of tic 1 is state 1, etc.

### 8.2.2 Visualizing the state space of an rbp network

To understand the recurrent backpropagation network, it is useful to visualize the states of all of the units in the network laid out on a series of planes, with one plane per state (Figure 8.2). Within each plane all of the pools of units in the network are represented. Projections are thought of as going forward from one plane to the next. Thus if pool(3) of a network receives a projection from pool(2), we can visualize nticks separate copies of this projection, one to the plane for state 1 from the plane for state 0, one to the plan for state 2 from the plan for state 2, etc.

### 8.2.3 Forward propagation of activation.

Processing begins after the states of all of the units have been initialized to their starting values for state 0. Actually, both the activations of units, and their net inputs, must be initialized. The net inputs are set to values that correspond to the activation (by using the inverse logistic function to calculate the starting net input from the starting activation). For cases where the unit is clamped at 0 or 1, the starting net input is based on the correct inverse logistic value for 2.0e-8 and 1-(2.0e-8). [CHECK]

After the net inputs and activations have been established for state 0, the net inputs are calculated for state 1. The resulting net input value is essentially a running average, combining what we will call the newnet, based on the activations at the previous tick with the old value of the net input:

$$newnet_i(t) = bias_i + ext_i(t) + \sum_j a_j(t-1)w_{ij}$$

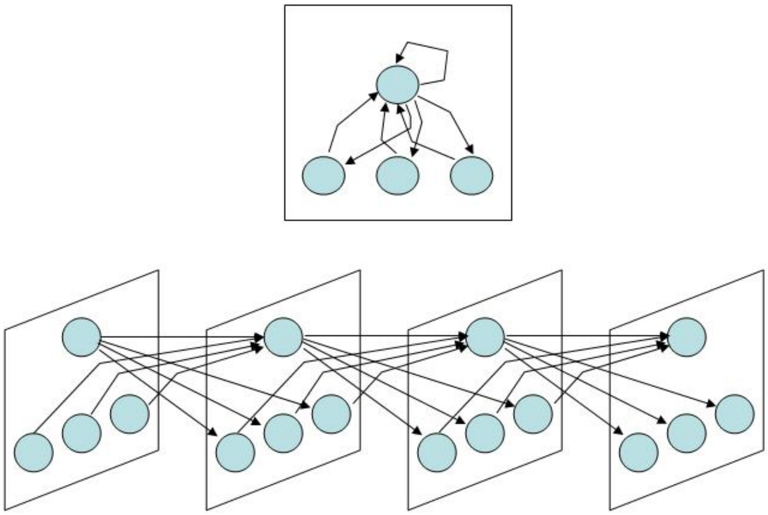


Figure 8.2: Standard (top) and unfolded (bottom) visualization of a recurrent neural network like the one used in the Rogers et al. (2004) model. The unfolded network makes clear the fact that activation at one time influences the activations at the next time step. In this unfolded form, the network is equivalent to a feed-forward network, where forward refers now to time. Activation feeds forward in time, and error signals (delta terms) feed back.

$$net_i(t) = dt * newnet_i(t) + (1 - dt) * net_i(t-1)$$

Note that if $dt = 1$ (the largest allowable value), the net input is not time averaged, and $net(t)$ simply equals $newnet(t)$.

After net inputs for state $t$ have been computed, activation values for state $t$ are then computed based on each unit's net input for state $t$. A variant on this procedure (not available in the **rbp** simulator) involves using the newnet to calculate a newact value, and then time averaging is then applied on the activations:

$$a_i(t) = dt * newact_i(t) + (1 - dt) * a_i(t-1)$$

Time-averaging the net inputs is preferred because net input approximates a neuron's potential while activation approximates its firing rate, and it is generally thought that it is the potential that is subject to time averaging ('temporal summation'). Also, time-averaging the net input allows activation states to change quickly if the weights are large. The dynamics seems more damped when time-averaging is applied to the activations.

Once activation values have been computed, error measures and delta terms (called *dEdnet*) are calculated, if a target has been specified for the given state of the unit. The error is thought of as spread out over the ticks that make up each interval over which the target is applied, and so the magnitude of the error at each time step is scaled by the size of the time step. Either sum squared error (*sse*) or cross entropy error (*cee*) can be used as the error measure whose derivative drives learning. In either case, we compute both measures of error, since this is fast and easy. The *sse* for a given unit at a given time step is

$$sse_i(t) = dt * (tgt_i(t) - a_i(t))^2,$$

where $tgt_i(t)$ is the externally supplied target at tick $t$. The *cce* for a given unit at a given time step is

$$cce_i(t) = -dt * (tgt_i(t) * log(a_i(t)) + (1 - tgt_i(t)) * log(1 - a_i(t))),$$

In the forward pass we also calculate a quantity we will here call the 'direct' *dEdnet* for each time step. This is that portion of the partial derivative of the error with respect to the net input of the unit that is directly determined by the presence of a target for the unit at time step $t$. If squared error is used, the direct *dEdnet* is given by

$$directdEdnet_i(t) = (tgt_i(t) - a_i(t)) * a_i(t) * (1 - a_i(t))$$

Note that the direct *dEdnet* will eventually be scaled by $dt$, but this is applied during the backward pass as discussed below. Of course if there is no target the $directdEdnet_i(t)$ is 0.

If cross-entropy error is used instead, we have the following simpler expression, due to the cancellation of part of the gradient of the activation function with part of the derivative of the cross entropy error:

$$directdEdnet_i(t) = (tgt_i(t) - a_i(t))$$

The process of calculating net inputs, activations, the two error measures, and the direct *dEdnet* takes place in the forward processing pass. This process continues until these quantities have been computed for the final time step.

The overall error measure for a specific unit is summed over all ticks for which a target is specified:

$$e_i = \sum_{t*} e_i(t)$$

This is done separately for the sse and the cce measures.

The activations of each unit in each tick are kept in an array called the activation history. Each pool of units keeps its own activation history array, which has dimensions [nticks+1, nunits]

### 8.2.4 Backward propagation of error

We are now ready for the backward propagation of the dEdnet values. We can think of the *dEdnet* values as being time averaged in the backwards pass, just as the net inputs are in the forward pass. Each state propagates back to the preceding state, both through the time averaging and by back-propagation through the weights, from later states to earlier states:

$$dEdnet_i(t) = dt * newdEdnet_i(t) + (1 - dt) * dEdnet_i(t + 1),$$

where

$$newdEdnet_i(t) = a_i(t) * (1 - a_i(t)) * \sum_k w_{ki} * dEdnet_k(t+1) + directdEdnet_i(t).$$

The subscript $k$ in the summation above indexes the units that receive connections from unit $i$. Note that state 0 is thought of as immutable, so deltas need not be calculated for that state. Note also that for the last state (the state whose index is $nticks + 1$), there is no future to inherit error derivatives from, so in that case we have simply have

$$dEdnet_i(nticks + 1) = dt * directdEdnet_i(nticks + 1).$$

For the backward pass calculation, $t$ starts at the next-to-last state (whose index is $nticks$) and runs backward to $t = 1$; however, for units and ticks where the external input is hard clamped, the value of *dEdnet* is kept at 0.

All of the $dEdnet_i(t)$ values are maintained in an array called dEdnethistory. As with the activations, there is a separate dEdnet history for each pool of units, which like the activation history array, has dimensions [nticks+1,nunits]. (In practice, the values we are calling *directdEdnet* scaled by $dt$ and placed in this history array on the forward pass, and the contents of that array is thus simply incremented during the backward pass).

### 8.2.5 Calculating the weight error derivatives

After the forward pass and the backward pass, we are ready to calculate the weight error derivatives arising from this entire episode of processing in the network. This calculation is very simple. For each connection weight, we simply add together the weight error derivative associated with each processing tick. The weight error derivative for each tick is just the product of the activation of the sending unit from the time step on the input side of the tick times the *dEdnet* value of the receiving unit on the receiving side of the tick:

$$wed_{ij} = \sum_{t=1}^{nticks+1} dEdnet_i(t) * a_j(t-1)$$

### 8.2.6 Updating the weights.

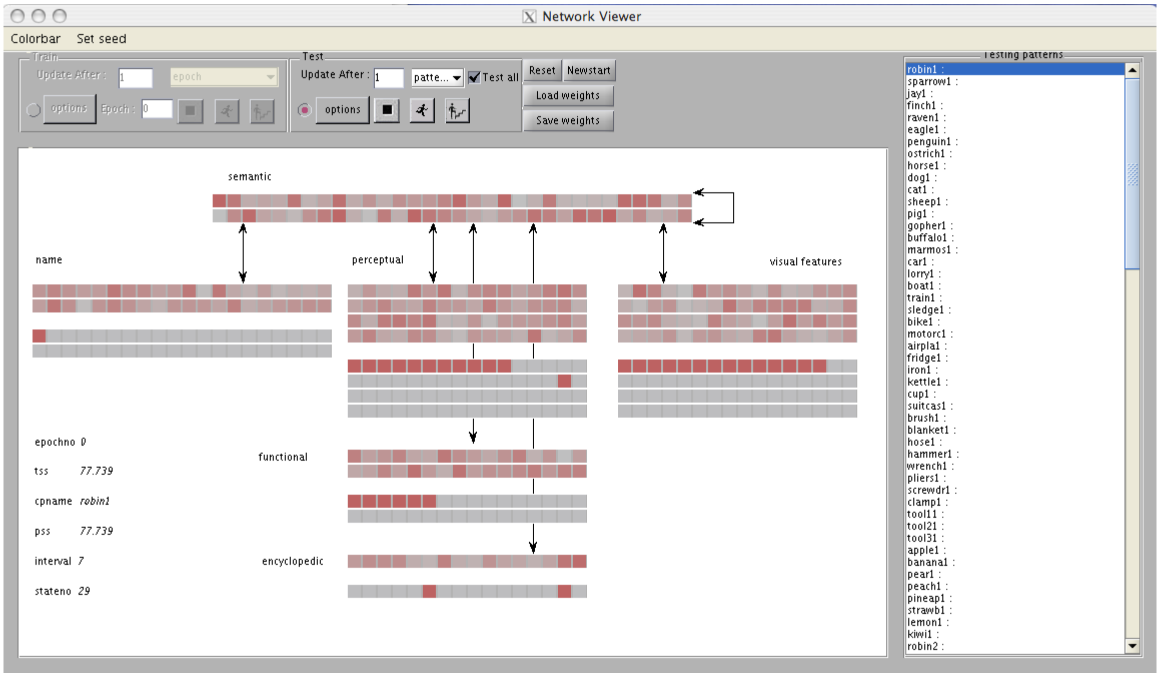
Once we have the weight error derivatives, we proceed exactly as we do in the back propagation algorithm as implemented in the **bp** program. As in standard back propagation, we can update the weights once per pattern, or once per N patterns, or once per epoch. Patterns may be presented in sequential, or permuted order, again as in the **bp** program. Momentum and weight decay can also be applied.

## 8.3 Using the rbp program with the rogers network

Training and testing in **rbp** work much the same as in other networks. The rogers network has been set up in advance for you, and so you can launch the program to run the rogers example by simply typing rogers at the MATLAB commmand prompt while in the **rbp** directory. See Figure 8.3 for the screen layout of this network.

During testing, the display can be updated (and the state of the network can be logged) at several different granularities: At the tick level, the interval level, and the pattern level. When tick level or interval level is specified, state 0 is shown first, followed by the state at the end of the first tick or interval, until the end of the final tick is reached. The template for the rogers model, *rogers.m*, also displays the target (if any) associated with the state, below the activations of each pool of units in the network. With updating at the pattern level, the state is updated only once at the end of processing each pattern.

Back propagation of error occurs only during training, although during training the display update options are limited to the pattern and the epoch level. The user can log activations forward and deltas back at the tick level via the set write options button in the train options panel (select *backtick* for ‘frequency’ of logging). Otherwise logging only occurs in the forward direction.



Train: Update After : 1 epoch

○ Epoch : 0

Test: Update After : 1 patte... ☑ Test all

●

epochno *0*

tss *77.739*

cpname *robin1*

pss *77.739*

interval *7*

stateno *29*

Testing patterns

robin1 :
sparrow1 :
jay1 :
finch1 :
raven1 :
eagle1 :
penguin1 :
ostrich1 :
horse1 :
dog1 :
cat1 :
sheep1 :
pig1 :
gopher1 :
buffalo1 :
marmos1 :
car1 :
lorry1 :
boat1 :
train1 :
sledge1 :
bike1 :
motorc1 :
airpla1 :
fridge1 :
iron1 :
kettle1 :
cup1 :
suitcas1 :
brush1 :
blanket1 :
hose1 :
hammer1 :
wrench1 :
pliers1 :
screwdr1 :
clamp1 :
tool11 :
tool21 :
tool31 :
apple1 :
banana1 :
pear1 :
peach1 :
pineap1 :
strawb1 :
lemon1 :
kiwi1 :
robin2 :

Figure 8.3: Architecture of the Rogers et al. (2004) model. Verbal descriptors (names, perceptual, functional, and encyclopedic) and visual feature units receive input directly from the environment. The environmental input is displayed directly below the corresponding pool activations.

### 8.3.1 rbp fast training mode.

A great deal of computing is required to process each pattern, and thus it takes quite a long time to run one epoch in the **rbp** program. To ameliorate this problem, we have implemented the pattern processing steps (forward pass, backward pass, and computing the weight error derivatives) within a 'mex' file (essentially, efficient C code that iterfaces with Matlab data formats and structures).Fast mode is approximately 20 times faster than the regular mode. To turn on the fast mode for network training, click the 'fast' option in the train window. This option can also be set by typing the following command on the MATLAB command prompt:
*runprocess('process','train','granularity','epoch','count',1,'fastrun',1);*

There are some limitations in executing fast mode of training.In this mode, network output logs can only be written at epoch level. Any smaller output frequency if specified, will be ignored.When running with fast code in gui mode, network viewer will only be updated at epoch level.Smaller granularity of display update is disabled to speed up processing.

There is a known issue with running this on linux OS.MATLAB processes running fast version appear to be consuming a lot of system memory. This memory management issue has not been observed on windows. We are currently investigating this and will be updating the documentation as soon as we find a fix.

### 8.3.2 Training and Lesioning with the rogers network

As provided the training pattern file used with the rogers network, (*features.pat*) contains 144 patterns. There are 48 different objects (eight each from six categories), with three training patterns for each. One provides external input to the name units, one to the verbal descriptor units (one large pool consisting of 'perceptual', 'functional' and 'encyclopedic' descriptors) and one provides input to the visual features units. In all three cases, targets are specified for all three visible pools. Each of the three visible pools is therefore an 'inout' pool. The network is set up to use Cross-Entropy error. If cross entropy error is not used, the network tends to fail to learn to activate all output units correctly. With cross-entropy error, this problem is avoided, and learning is quite robust.

If one wanted to simulate effects of damage to connections in the rogers network, the best approach would be to apply a mask to the weights in particular projections. For example, to lesion (i.e. zero out) weights in net.pool(4).proj(3) with $r$ receivers and $s$ senders, with a lesion probability of $x$:

1. First save your complete set of learned weights using the save weights command.

2. Type the following to find the number of receivers $r$ and senders $s$:

```
[r s] = size(net.pool(4).proj(3).weight);
```

3. Then create a mask of 0's and 1's to specify which weights to destroy (0) and which to keep (1):

   ```
   mask = ceil(rand(r,s) - repmat(x,r,s));
   ```

   This creates a mask matrix with each entry being zero with probability $x$ and 1 with probability $1 - x$.

4. Then use the elementwise matrix multiply to zero the unfortunate connections:

   ```
   net.pool(4).proj(3).weight = net.pool(4).proj(3).weight.*mask;
   ```

   This will zero out all of the weights associated with mask values of 0. You can apply further lesions if you wish or re-load and apply different lesions as desired.

One can add Gaussian noise with standard deviation s to the weights in a particular projection even more simply:

```
net.pool(4).proj(3).weight = net.pool(4).proj(3).weight + s*randn(r,s);
```

Lesioning units is a bit more complicated, and routines need to be implemented to accomplish this.

### 8.3.3 rbp pattern files.

The pattern file must begin with the string 'rbp' and this is followed by a ':' and two numbers, *nintervals* and *ticks per interval*. In the example provided (*features.pat*, for use with the rogers network) this line is thus:

```
rbp : 7 4
```

The pattern specifications follow the single header line. Here is an example of a pattern specification from the rogers network:

```
robin1
H 1 3 name 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
T 6 2 name 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
T 6 2 verbal_descriptors 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
T 6 2 visual_features 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
end
```

Each pattern specification begins with a *pname* followed by one or more input specifications and one or more target specifications followed by *end*. Input specifications begin with the letter $H$ or $S$ to indicate whether the pattern should be hard or soft clamped, followed by a start time and a duration. The first interval is interval 1 so to clamp from the beginning of a pattern presentation and leave the input clamped for three intervals, you would have 1 3. This information is followed by the name of the layer to which the pattern should be applied, followed by a string of numbers specifying input values for each unit in the pool. If the letter is $H$ the states of the units in the specified pool are hard clamped to the input values. If the letter is $S$ the value specified is treated as a component of the net input of the unit. In this case the input value and the unit's bias determine its initial net input for state 0, and its activation is set to the appropriate value for that net input.

For both hard and soft clamps, the input is applied to the state at the starting edge of the start time indicated, and remains in place for duration*ticksperinterval. In this case, the values 1 3 mean that the input is clamped on in states 0 through 11. This does not include the state at the starting edge of interval 4 (state 12).

Target specifications begin with the letter T then a start time and a duration. In this case the values are 6 and 2, specifying that the target is clamped for two intervals beginning with interval 6. The target applies to the state at the trailing edge of the first tick after the start time. So in this case the target applies to states 22 to 29. As with input patterns, the start time and duration are followed by a pool name and a sequence of values specifying targets for the units in the pool indicated.

### 8.3.4 Creating an rbp network

When creating an **rbp** network, it is necessary to specify the value of the variables *net.nintervals* and *net.ticksperinterval.*The values of these variables should be specified immediately after specifying the network type as an **rbp** network, before specifying pools and projections. Once created the .net file can be edited to change these parameters if desired, but the program must be closed and restarted for these changes to take effect. This makes it possible to train the network for some number of epochs with *ticksperinteral* at 1 and then save the weights, exit the program, change ticks per interval to 5, and then restart the network and reload the weights for fine tuning.

Specifying pools and projections in **rbp** is similar to other programs. Note however that in rbp, layers can project to themselves and there can be both a projection from layer $a$ to layer $b$ and from layer $b$ to layer $a$. All these projections are options, and each must be independently specified in the network specification file. Note also that layers can be *input*, *output*, *inout*, or *hidden* layers. An *inout* layer can have external inputs specified and also have targets specified.

# Chapter 9

# Temporal-Difference Learning

In this chapter, we introduce a reinforcement learning method called Temporal-Difference (TD) learning. Many of the preceding chapters concerning learning techniques have focused on supervised learning in which the target output of the network is explicitly specified by the modeler (with the exception of Chapter 6 Competitive Learning). TD learning is an unsupervised technique in which the learning agent learns to predict the expected value of a variable occurring at the end of a sequence of states. Reinforcement learning (RL) extends this technique by allowing the learned state-values to guide actions which subsequently change the environment state. Thus, RL is concerned with the more holistic problem of an agent learning effective interaction with its environment. We will first describe the TD method for prediction, and then extend this to a full conceptual picture of RL. We then explain our implementation of the algorithm to facilitate exploration and solidify an understanding of how the details of the implementation relate to the conceptual level formulation of RL. The chapter concludes with an excercise to help fortify understanding.

## 9.1 BACKGROUND

To borrow an apt example from Sutton (1988), imagine trying to predict Saturday's weather at the beginning of the week. One way to go about this is to observe the conditions on Monday (say) and pair this observation with the actual meteorological conditions on Saturday. One can do this for each day of the week leading up to Saturday, and in doing so, form a training set consisting of the weather conditions on each of the weekdays and the actual weather that obtains on Saturday. Recalling Chapter 5, this would be an adequate training set to train a network to predict the weather on Saturday. This general approach is called *supervised learning* because, for each input to the network, we have explicily (i.e. in a supervisory manner) specified a target value for the network. Note, however, that one must know the actual outcome value in order to train this network to predict it. That is, one must have observed the actual weather on Saturday before any learning can occur. This proves to be somewhat limiting in a variety of real-world scenarios, but most importantly, it seems a rather inefficient use of the information we have leading up to Saturday. Suppose, for instance, that it has rained persistently throughout Thursday and Friday, with little sign that the storm will let up soon. One would naturally expect there to be a higher chance of it raining on Saturday as well. The reason for this expectation is simply that the weather on Thursday and Friday are relevant to predicting the weather on Saturday, even without actually knowing the outcome of Saturday's weather. In other words, partial information relevant to our prediction for Saturday becomes available on each day leading up to it. In supervised learning as it was previously described, this information is effectively not employed in learning because Saturday's weather is our sole target for training.

*Unsupervised learning*, in contrast, operates instead by attempting to use intermediate information, in addition to the actual outcome on Saturday, to learn to predict Saturday's weather. While learning on observation-outcome pairs is effective for predictions problems with only a single step, pairwise learning, for the reason motivated above, is not well suited to prediction problems with multiple steps. The basic assumption that underlies this unsupervised approach is that predictions about some future value are ”not confirmed or disconfirmed all at once, but rather bit by bit” as new observations are made over many steps leading up to observation of the predicted value (Sutton, 1988). (Note that, to the extent that the outcome values are set by the modeler, TD learning is not unsupervised. We use the terms 'supervised' and 'unsupervised' here for convenience to distinguished the pairwise method from the TD method, as does Sutton. The reader should be aware that the classification of TD and RL learning as unsupervised is contested.)

While there are a variety of techniques for unsupervised learning in prediction problems, we will focus specifically on the method of Temporal-Difference (TD) learning (Sutton, 1988). In supervised learning generally, learning occurs by minimizing an error measure with respect to some set of values that parameterize the function making the prediction. In the connectionist applications that we are interested in here, the predicting function is realized in a neural network, the error measure is most often the difference between the output of the predicting function and some prespecificed target value, and the values that parameterize the function are connection weights between units in the network. For now, however, we will dissociate our discussion of the prediction function from the details of its connectionist implementation for the sake of simplicity and because the principles which we will explore can be implemented by methods other than neural networks. In the next section, we will turn our attention to understanding how neural networks can implement these prediction techniques and how the marriage of the two techniques (TD and connectionism) can provide added power.

The general supervised learning paradigm can be summarized in a familiar formula

$$\Delta w_t = \alpha \left( z - V(s_t) \right) \nabla_w V(s_t),$$

where $w_t$ is a vector of the parameters of our prediction function at time step $t$, $\alpha$ is a learning rate constant, $z$ is our target value, $V(s_t)$ is our prediction for input state $s_t$, and $\nabla_w V(s_t)$ is the vector of partial derivatives of the prediction with respect to the parameters $w$. We call the function, $V(\cdot)$, which we are trying to learn, the value function. This is the same computation that underlies back propagation, with the amendment that $w_t$ are weights in the network and the back propagation procedure is used to calculate the gradient $\nabla_w V(s_t)$. As we noted earlier, this update rule cannot be computed incrementally with each step in a multi-step prediction problem because the value of $z$ is unknown until the end of the sequence. Thus, we are required to observe a whole sequence before updating the weights, so the weight update for an entire sequence is just the sum of the weight changes for each time step,

$$w \leftarrow w + \sum_{t=1}^{m} \Delta w_t$$

where $m$ is the number of steps in the sequence.

The insight of TD learning is that the error, $z - V_t$, at any time can be represented as the sum of changes in predictions on adjacent time steps, namely

$$z - V_t = \sum_{k=t}^{m} (V_{k+1} - V_k)$$

where $V_t$ is shorthand for $V(s_t)$, $V_{m+1}$ is defined as $z$, and $s_{m+1}$ is the terminal state in the sequence. The validity of this formula can be verified by merely expanding the sum and observing that all terms cancel, leaving only $z - V_t$.

Thus, substituting for $\Delta w_t$ in the sequence update formula we have

$$w \leftarrow w + \sum_{t=1}^{m} \alpha (z - V_t) \nabla_w V_t,$$

and substituting the sum of differences for the error, $(z - V_t)$, we have,

$$w \leftarrow w + \sum_{t=1}^{m} \alpha \sum_{k=t}^{m} (V_{k+1} - V_k) \nabla_w V_t.$$

After some algebra, we obtain

$$w \leftarrow w + \sum_{t=1}^{m} \alpha (V_{k+1} - V_k) \sum_{k=1}^{t} \nabla_w V_t$$

from which we can extract the weight update rule for a single time step by noting that the term inside the first sum is equivalent to $\Delta w_t$ by comparison to the sequence update equation,

$$\Delta w_t = \alpha \left(V_{t+1} - V_t\right) \sum_{k=1}^{t} \nabla_w V_k.$$

It is worth pausing to reflect on how this update rule works mechanistically. We need two successive predictions to form the error term, so we must already be in state $s_{t+1}$ to obtain our prediction $V_{t+1}$, but learning is occurring for prediction $V_t$ (i.e. $w_t$ is being updated). Learning is occuring retroactively for already visited states. This is accomplished by maintaining the gradient $\nabla_w V_k$ as a running sum. When the error appears at time $t+1$, we obtain the appropriate weight changes for updating *all* of our predictions for previous input states by multiplying this error by the summed gradient. The update rule is, in effect, updating all preceding predictions to make each closer to prediction $V_{t+1}$ for the current state by using the difference in the successive prediction values, hence the name Temporal-Difference learning.

This update rule will produce the same weight changes over the sequence as the supervised version will (recall that we started with the supervised update equation for a sequence in the derivation), but this rule allows weight updates to occur incrementally - all that is necessary is a pair of successive predictions and the running sum of past output gradients. Sutton (1988) calls this the TD(1) procedure and introduces a generalization, called TD($\lambda$), which produces weight changes that supervised methods cannot produce. The generalized formula for TD($\lambda$) is

$$\Delta w_t = \alpha \left(V_{t+1} - V_t\right) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w V_k.$$

We have added a constant, $\lambda$, which exponentially discounts past gradients. That is, the gradient of a prediction occurring $n$ steps in the past is weighted by $\lambda^n$. Naturally, $\lambda$ lies in the interval [0,1]. We have seen that $\lambda = 1$ produces the same weight changes as a supervised method that uses observation-outcome pairs as its training set. Setting $\lambda = 0$ results in the formula

$$\Delta w_t = \alpha \left(V_{t+1} - V_t\right) \nabla_w V_t.$$

This equation is equivalent to the the supervised learning rule with $V_{t+1}$ substituted for $z$. Thus, TD(0) produces the same weight changes as would a supervised method whose training pairs are simply states and the predictions made on the immediately follow state, i.e. training pairs consist of $s_t$ as input and $V_{t+1}$ as target. Conceptually, intermediate values of $\lambda$ result in an update rule that falls somewhere between these two extremes. Specifically, $\lambda$ determines the extent to which the prediction values for previous observations are updated by errors occuring on the current step. In other words, it tracks to what extent previous predictions are eligible for updating based on current errors. Therefore, we call the sum,

$$e_t = \sum_{k=1}^{t} \lambda^{t-k} \nabla_w V_k,$$

the *eligibility trace* at time $t$. Eligibility traces are the primary mechanisms of temporal credit assignment in TD learning. That is, credit (or "blame") for the TD error occurring on a given step is assigned to the previous steps as determined by the eligibility trace. For high values of $\lambda$, predictions occurring earlier in a sequence are updated to a greater extent than with lower values of $\lambda$ for a given error signal on the current step.

To consider an example, suppose we are trying to learn state values for the states in some sequence. Let us refer to states in the sequence by letters of the alphabet. Our goal is to learn a function $V(\cdot)$ from states to state-values. State-values approximate the expected value of the variable occuring at the end of the sequence. Suppose we initialize our value function to zero, as is commonly done (it is also common to initialize them randomly, which is natural in the case of neural networks). We encounter a series of states such as $a$, $b$, $c$, $d$. Since predictions for these values is zero at the outset, no useful learning will occur until we encounter the outcome value at the end of the sequence. Nonetheless, we find the output gradient for each input state and maintain the eligibility trace as a discounted sum of these gradients. Suppose that this particular sequence ends with $z = 1$ at the terminal state. Upon encountering this, we have a useful error as our training signal, $z - V(d)$. After obtaining this error, we multiply it by our eligibility trace to obtain the weight changes. In applying these weight changes, we correct the predictions of all past states to be closer to the value at the end of the sequence, since the eligibility trace includes the gradient for past states. This is desirable since all of the past states in the sequence led to an outcome of 1. After a few such sequences, our prediction values will no longer be random. At this point, valuable learning can occur even before a sequence ends. Suppose that we encounter states $c$ and then $b$. If $b$ already has a high (or low) value, then the TD error, $V(b) - V(c)$ will produce an appropriate adjustment in $V(c)$ toward $V(b)$. In this way, TD learning is said to *bootstrap* in that it employs its own value predictions to correct other value predictions. One can think of this learning process as the propagation of the outcome value back through the steps of the sequence in the form of value predictions. Note that this propagation occurs even when $\lambda = 0$, although higher values of $\lambda$ will speed the process. As with supervised learning, the success of TD learning depends on passes through many sequences, although learning will be more efficient in general.

Now we are in a state to consider how it is exactly that TD errors can drive more efficient learning in prediction problems than its supervised counterpart. After all, with supervised learning, each input state is paired with the actual value that it is trying to predict - how can one hope to do better than training on veridical outcome information? Figure 9.1 illustrates a scenario in which we can do better. Suppose that we are trying to learn the value function for a game. We have learned thus far that the state labeled BAD leads to a loss 90% of the time and a win 10% of the time, and so we appropriately assign it a low value. We now encounter a NOVEL state which leads to BAD but then results in a win. In supervised learning, we would construct a NOVEL-win training pair and the NOVEL state would be initially assigned a high value. In TD learning,

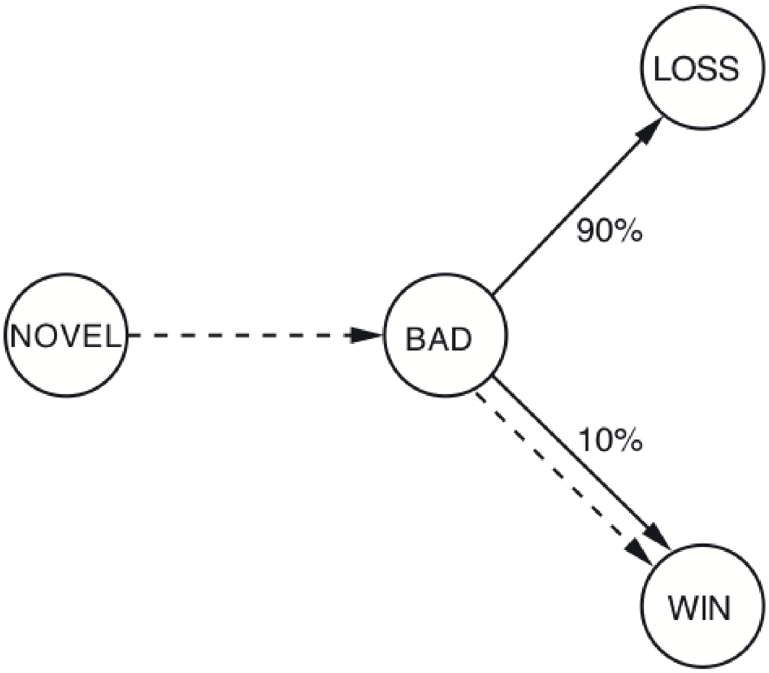


Figure 9.1: State space of a game, see text for explanation. (From Sutton (1988). Reprinted by permission.)

the NOVEL state is paired with the subsequent BAD state, and thus TD assigns the NOVEL state a low value. This is, in general, the correct conclusion about the NOVEL state. Supervised methods neglect to account for the fact that the NOVEL state ended in a win only by transitioning through a state that we already know to be bad. With enough training examples, supervised learning methods can acheive equivalent performance, but TD methods make more efficient use of training data in multi-step prediction problems by bootstrapping in this way and so can learn more efficiently with limited experience.

## 9.2 REINFORCEMENT LEARNING

So far we have seen how TD learning can learn to predict the value of some variable over multiple time steps - this is the *prediction problem.* Now we will address the *control problem*, i.e. the problem of getting an RL agent to learn how to control its environment. The control problem is often considered to be the complete reinforcement learning problem in that the agent must learn to both predict the values of environment states that it encounters and also use those predicted values to change the environment in order to maximize reward. In this section, we introduce the framework for understanding the full

RL problem. Later, we present a case study of combining TD learning with connectionist networks.

### 9.2.1 Discounted Returns

We now introduce a slight change in terminology with an accompanying addition to the concept of predicting outcomes that we explored in the previous section. We will hereafter refer to the outcome value of a sequence as the *reward.* Since we will consider cases in which the learning agent is trying to maximize the value of the outcome, this shift in terminology is natural. Rewards can be negative as well as positive - when they are negative they can be thought of as *punishment.* We can easily generalize the TD method to cases where there are intermediate reward signals, i.e. reward signals occuring at some state in the sequence other than the terminal state. We consider each state in a sequence to have an associated reward value, although many states will likely have a reward of 0. This transforms the prediction problem from one of estimating the expected value of a reward occurring at the end of a sequence to one of estimating the expected value of the sum of rewards encountered in a sequence. We call this sum of rewards the *return.* Using the sum of rewards as the return is fine for tasks which can be decomposed into distinct episodes. However, to generalize to tasks which are ongoing and do not decompose into episodes, we need a means by which to keep the return value bounded. Here we introduce the discounting parameter, gamma, denoted $\gamma$. Our total discounted return is given by

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}.$$

Discounting simply means that rewards arriving futher in the future are worth less. Thus, for lower values of $\gamma$, distal rewards are valued less in the value prediction for the current time. The addition of the $\gamma$ parameter not only generalizes TD to non-episodic tasks, but also provides a means by which to control how far the agent should look ahead in making predictions at the current time step.

We would like our prediction $V_t$ to approximate $R_t$, the expected return at time $t$. Let us refer to the correct prediction for time $t$ as $V_t^*$; so $V_t^* = R_t$. We can then derive the generalized TD error equation from the right portion of the return equation.

$$\begin{aligned}
V_t^* &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\
&= r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+k+2} \\
&= r_{t+1} + \sum_{k=0}^{\infty} \gamma^{k+1} r_{t+k+2} \\
&= r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \\
&= r_{t+1} + \gamma V_{t+1}^*
\end{aligned}$$

We note that the TD error at time $t$ will be $E_t = V_t^* - V_t$ and we use $V_{t+1}$ as an imperfect proxy for the true value $V_{t+1}^*$ (bootstrapping). We substitute to obtain the generalized error equation:

$$E_t = (r_{t+1} + \gamma V_{t+1}) - V_t$$

This error will be multiplied by a learning rate $\alpha$ and then used to update our weights. The general update rule is

$$V_t \leftarrow V_t + \alpha[r_{t+1} + \gamma V_{t+1} - V_t]$$

Recall, however, that when using the TD method with a function approximator like a neural network, we update $V_t$ by finding the output gradient with respect to the weights. This step is not captured in the previous equation. We will discuss these implementation specific details later.

It is worthwhile to note that the return value provides a replacement for planning. In attaining some distal goal, we often thing that an agent must plan many steps into the future to perform the correct sequence of actions. In TD learning, however, the value function $V_t$ is adjusted to reflect the total expected return after time $t$. Thus, in considering how to maximize total returns in making a choice between two or more actons, a TD agent need only choose the action with the highest state value. The state values themselves serve as proxies for the reward value occurring in the future. Thus, the problem of planning a sequence of steps is reduced to the problem of choosing a next state with a high state value. Next we explore the formal characterization of the full RL problem.

### 9.2.2 The Control Problem

In specifying the control problem, we divide the agent-environment interaction into conceptually distinct modules. These modules serve to ground a formal characterization of the control problem independent of the details of a particular environment, agent, or task. Figure 9.4.1 is a schematic of these modules as they

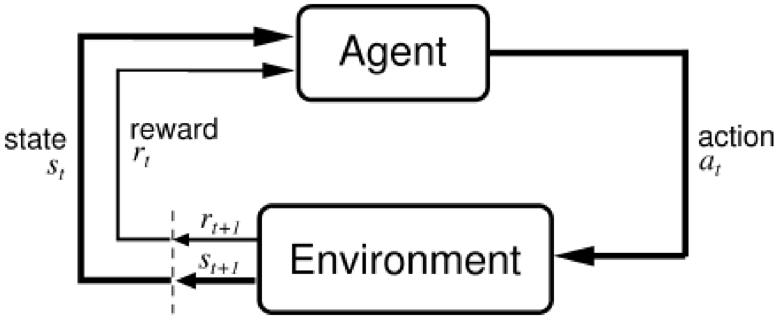


Figure 9.2: The architecture of a RL agent. On the left, the arrows labeled 'state' and 'reward' denote the two signals that the agent received from the environment. On the right, the arrow labeled 'action' denotes the only signal the environment receives from the agent. For each step, the agent receives state and reward signals and then produces an action signal that changes the environment. The dotted line denotes the time horizon of a single step with the new state and reward signals after action $a_t$ has been performed. (From Sutton and Barto (1998). Reprinted by permission.)

relate to one another. The environment provides two signals to the agent: the current environment state, $s_t$, which can be thought of as a vector specifying all the information about the environment which is available to the agent; and the reward signal, $r_t$, which is simply the reward associated with the co-occurring state. The reward signal is the only training signal from the environment. Another training signal is generated internally to the agent: the value of the successor state needed in forming the TD error. The diagram also illustrates the ability of the agent to take an action on the environment to change its state.

Note that the architecture of a RL agent is abstract: it need not align with actual physical boundaries. When thinking conceptually about an RL agent, it is important to keep in mind that the agent and environment are demarcated by the limits of the agent's control (Sutton and Barto, 1998). That is, anything that cannot be arbitrarily changed by the agent is considered part of the agent's environment. For instance, although in many biological contexts the reward signal is computed inside the physical bounds of the agent, we still consider them as part of the environment because they cannot be trivially modified by the agent. Likewise, if a robot is learning how to control its hand, we should consider the hand as part of the environment, although it is physically continuous with the robot.

We have not yet specified how action choice occurs. In the simplest case, the agent evaluates possible next states, computes a state value estimate for each one, and chooses the next state based on those estimates. Take, for instance, an agent learning to play the game Tic-Tac-Toe. There might be a vector of length nine to represent the board state. When it is time for the RL agent to choose an action, it must evaluate each possible next state and obtain a state value for each. In this case, the possible next states will be determined by the open spaces where an agent can place its mark. Once a choice is made, the environment is updated to reflect the new state.

Another way in which the agent can modify its environment is to learn state-action values instead of just state values. We define a function $Q(\cdot)$ from state-action pairs to values such that the value of $Q(s_t, a_t)$ is the expected return for taking action $a_t$ while in state $s_t$. In neural networks, the $Q$ function is realized by having a subset of the input units represent the current state and another subset represent the possible actions. For the sake of illustration, suppose our learning agent is a robot that is moving around a grid world to find pieces of garbage. We may have a portion of the input vector to represent sensory information that is available to the robot at its current location in the grid - this portion corresponds to the state in the $Q$ function. Another portion would be a vector for each possible action that the robot can take. Suppose that our robot can go forward, back, left, right, and pickup a piece of trash directly in front of it. Therefore, we would have a five place action vector, one place corresponding to each of the possible actions. When we must choose an action, we simply fix the current state on the input vector and turn on each of the five action units in turn, computing the value for each. The next action is chosen from these state-action values and the environment state is modified accordingly.

The update equation for learning state-action values is the same in form as that of learning state values which we have already seen:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)].$$

This update rule is called SARSA for State Action Reward State Action because it uses the state and action of time $t$ along with the reward, state, and action of time $t + 1$ to form the TD error. This update rule is called *associative* because the values that it learns are associated with particular states in the environment. We can write the same update rule without any $s$ terms. This update rule would learn the value of actions, but these values would not be associated with any environment state, thus it is *nonassociative.* For instance, if a learning agent were playing a slot machine with two handles, it would be learning only action values, i.e. the value of pulling lever one versus pulling lever two. The associative case of learning state-action values is commonly thought of as the full reinforcement learning problem, although the related state value and action value cases can be learned by the same TD method.

Action choice is specified by a *policy.* A policy is considered internal to the agent and consists of a rule for choosing the next state based on the value predictions for possible next states. More specifically, a policy maps state values to actions. Policy choice is very important due to the need to balance *exploration* and *exploitation.* The RL agent is trying to accomplish two related goals at the same time: it is trying to learn the values of states or actions and it is trying to control the environment. Thus, initially the agent must explore the state space to learn approximate value predictions. This often means taking suboptimal actions in order to learn more about the value of an action or state. Consider an agent who takes the highest-valued action at every step. This policy is called a *greedy* policy because the agent is only exploiting its current knowledge of state or action values to maximize reward and is not exploring to improve its estimates of other states. A greedy policy is likely undesirable for learning because there may be a state which is actually good but whose value cannot be discovered because the agent currently has a low value estimate for it, precluding its selection by a greedy policy. Furthermore, constant exploratory action is necessary in *nonstationary* tasks in which the reward values actually change over time.

A common way of balancing the need to explore states with the need to exploit current knowledge in maximizing rewards is to follow an $\varepsilon$-greedy policy which simply follows a greedy policy with probabiliy $1 - \varepsilon$ and takes a random action with probability $\varepsilon$. This method is quite effective for a large variety of tasks. Another common policy is softmax. You may recognize softmax as the stochastic activation function for Boltzmann machines introduced in Chapter 3 on Constraint Satisfaction - it is often called the Gibbs or Boltzmann distribution. With softmax, the probability of choosing action $a$ at time $t$ is

$$p(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}},$$

where the denominator sums over all the exponentials of all possible action values and $\tau$ is the temperature coefficient. A high temperature causes all actions to be equiprobable, while a low temperature skews the probability toward a greedy policy. The temperature coefficient can also be annealed over the course of training, resulting in greater exploration at the start of training and greater exploitation near the end of training. All of these common policies are implemented in the pdptool software, along with a few others which are documented in Section 9.5.

## 9.3 TD AND BACK PROPAGATION

As mentioned previously, TD methods have no inherent connection to neural network architectures. TD learning solves the problem of *temporal credit assignment*, i.e. the problem of assigning blame for error over the sequence of predictions made by the learning agent. The simplest implementation of TD learning employs a lookup table where the value of each state or state-action pair is simply stored in a table, and those values are adjusted with training. This method is effective for tasks which have an enumerable state space. However, in many tasks, and certainly in tasks with a continuous state space, we cannot hope to enumerate all possible states, or if we can, there are too many for a practical implementation. Back propagation, as we already know, solves the problem of *structural credit assignment*, i.e. the problem of how to adjust the weights in the network to minimize the error. One of the largest benefits of neural networks is, of course, their ability to generalize learning across similar states. Thus, combining both TD and back propagation results in an agent that can flexibly learn to maximize reward over mulitple time steps and also learn structural similarities in input patterns that allow it to generalize its predictions over novel states. This is a powerful coupling. In this section we sketch the mathematical machinery involved in coupling TD with back propagation. Then we explore a notable application of the combined TDBP algorithm to the playing of backgammon as a case study.

### 9.3.1 Back Propagating TD Error

As we saw in the beginning of this chapter, the gradient descent version of TD learning can be described by the general equation reproduced below:

$$\Delta w_t = \alpha \left( V_{t+1} - V_t \right) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w V_k.$$

The key difference between regular back propagation and TD back propagation is that we must adjust the weights for the input at time $t$ at time $t+1$. This requires that we compute the output gradient with respect to weights for input at time $t$ and save this gradient until we have the TD error at time $t+1$. Just as in regular back propagation, we use the logistic activation function, $f(net_i) = \frac{1}{1+exp(-net_i)}$, where $net_i$ is the net input coming into output unit $i$ from a unit $j$ that projects to it. Recall that the derivative of the activation function with respect to net input is simply $\frac{\partial a_i}{\partial net_i} = f'(net_i) = a_i(1-a_i)$, where $a_i$ is the activation at output unit $i$, and the derivative of the net input $net_i$ with respect to the weights is $\frac{\partial net_i}{\partial w_{ij}} = a_j$. Thus, the output gradient with respect to weights for the last layer of weights is

$$\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial a_i}{\partial net_i} \frac{\partial net_i}{\partial w_{ij}} = a_i(1-a_i)a_j.$$

Similarly, the output gradient for a weight from unit $k$ to unit $j$ is

$$\frac{\partial a_i}{\partial w_{jk}} = \frac{\partial a_i}{\partial net_i} \frac{\partial net_i}{\partial a_j} \frac{\partial a_j}{\partial net_j} \frac{\partial net_j}{\partial w_{jk}}$$

We have seen all of this before in Chapter 5. But now, we use $\delta_{ik}$ to denote $\frac{\partial a_i}{\partial net_k}$, where $k$ is a hidden layer unit and $i$ is an output unit. By using the chain rule, we can calculate a $\delta$ term for each hidden layer. $\delta_{ii}$ for the output layer is simply $a_i(1-a_i)$. The $\delta$ term for any hidden unit $k$ is given by

$$\delta_{ik} = \delta_{ij} w_{jk} f'(net_k) = \frac{\partial a_i}{\partial net_j} \frac{\partial net_j}{\partial a_k} \frac{\partial a_k}{\partial net_k} = \frac{\partial a_i}{\partial net_k},$$

where $j$ is a unit in the next layer forward and $\delta_{ij}$ has already been computed for that layer. This defines the back propagation process that we must use. Note the difference between this calculation and the similar calculation of $\delta$ terms introduced in Chapter 5: in regular back propagation, the $\delta_i$ at the output is $(t_i - a_i)f'(net_i)$. Thus, in regular back propagation, the error term is already multiplied into the $\delta$ term for each layer. This is not the case in TDBP because we do not yet have the error. This means that, in general, the $\delta$ term for any hidden layer will be a two dimensional matrix of size $i$ by $k$ where $i$ is the number of output units and $k$ is the number of units at the hidden layer. In the case of the output layer, the $\delta$ term is just a vector of length $i$.

We use the $\delta$ terms at each layer to calculate the output gradient with respect to the weights by simply multiplying it by the activation of the projecting layer:

$$\frac{\partial a_i}{\partial w_{jk}} = \frac{\partial a_i}{\partial net_j}\frac{\partial net_j}{\partial w_{jk}} = \delta_{ij} a_k.$$

In general, this will give us a three dimensional matrix for each weight projection $w_{jk}$ of size $i$ by $j$ by $k$ where element $(i, j, k)$ is the derivative of output unit $i$ with respect to the weight from unit $k$ to unit $j$. This gradient term is scaled by $\lambda$ and added to the accumulating eligibility trace for that projection. Thus, eligibility traces are three dimensional matrices denoted by $e_{ijk}$. Upon observing the TD error at the next time step, we compute the appropriate weight changes by

$$\Delta w_{jk} = \sum_i E_i e_{ijk}$$

where $E_i$ is the TD error at output unit $i$.

In regular back propagation, for cases where a hidden layer projects to more than one layer, we can calculate $\delta$ terms back from each output layer and simply sum them to obtain the correct $\delta$ for the hidden layer. This is because the error is already multiplied into the $\delta$ terms, and they will simply be vectors of the same size as the number of units in the hidden layer. In our implementation of TDBP, however, the $\delta$ term at a hidden layer that projects to more than one layer is the concatenation of the two $\delta$ matrices along the $i$ dimension (naturally, this is true of the eligibility trace at that layer as well). The TD error for that layer is similarly a vector of error terms for all output units that the layer projects to. Thus, we obtain the correct weight changes for networks with multiple paths to the output units.

### 9.3.2 Case Study: TD-Gammon

Backgammon, briefly, is a board game which consists of two players alternately rolling a pair of dice and moving their checkers opposite directions around the playing board. Each player has fifteen checkers distributed on the board in a standard starting position illustrated in Figure 9.3.2. On each turn, a player rolls and moves one checker the number shown on one die and then moves a second checker (which can be the same as the first) the number shown on the other die. More than one of a player's checkers can occupy the same place on the board, called a "point." If this is the case, the other player cannot land on

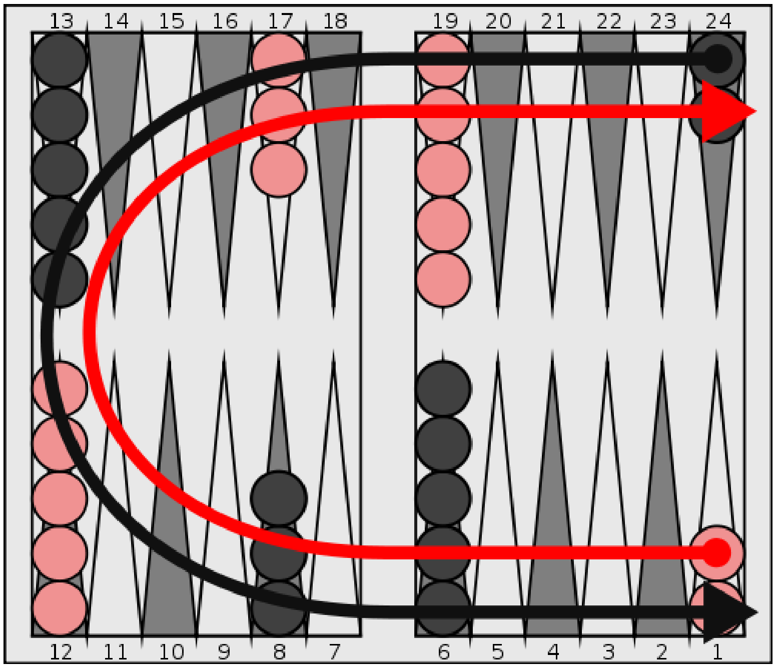


Figure 9.3: The starting position and movement of checkers in backgammon. Players move in opposite directions around the board in a horseshoe shape. (Adapted from the Wikimedia Commons file "File:Bg-movement.svg" http://commons.wikimedia.org/wiki/File:Bg-movement.svg)

the point. However, if there is only one checker on a point, the other player can land on the point and take the occupying checker off the board. The taken pieces are placed on the "bar", where they must be reentered into the game. The purpose of the game is to move each of one's checkers all the way around and off the board. The first player to do this wins. If a player manages to remove all her pieces from the board before the other player has removed any of his pieces, she is said to have won a "gammon," which is worth twice a normal win. If a player manages to remove all her checkers and the other player has removed none of his and has checkers on the bar, then she is said to have won a "backgammon," which is worth three times a normal win. The game is often played in matches and can be accompanied by gambling.

The game, like chess, has been studied intently by computer scientists. It has a very large branching rate: the number of moves available on the next turn is very high due to the high number of possible dice rolls and the many options for disposing of each roll. This limits tree search methods from being very effective in programming a computer to play backgammon. The large number of possible board positions also precludes effective use of lookup tables. Gerald Tesauro at

IBM in the late 80's was the first to successfully apply TD learning with back propagation to learning state values for backgammon (Tesauro, 1992).

Tesauro's TD-Gammon network had an input layer with the board representation and a single hidden layer. The output layer of the network consisted of four logistic units which estimate the probability of white or black both achieving a regular win or a gammon. Thus, the network was actually estimating four outcome values at the same time. The input representation in the first version of TD-Gammon was 198 units. The number of checkers on each of the 24 points on the board were represented by 8 units. Four of the units were devoted to white and black respectively. If one white checker was on a given point, then the first unit of the four was on. If two white checkers were on a given point, then both the first and second unit of the group of four were on. The same is true for three checkers. The fourth unit took on a graded value to represent the number of checkers above three: $(n/2)$, where $n$ is the number of checkers above three. In addition to these units representing the points on the board, an additional two units encoded how many black and white checkers were on the bar (off the board), each taking on a value $(n/2)$, where $n$ is the number of black/white checkers on the bar. Two more units encoded how many of each player's checkers had already been removed from the board, taking on values of $(n/15)$, where $n$ is the number of checkers already removed. The final two units encoded whether it was black or white's turn to play. This input projected to a hidden layer of 40 units.

Tesauro's previous backgammon network, Neurogammon, was trained in a supervised manner on a corpus of expert level games. TD-Gammon, however, trained by completely by self-play. Moves were generated by the network itself in the following fashion: the computer simulated a dice roll and generated all board positions that were possible from the current position and the number of the die. The network was fed each of these board positions and the one that the network ranked highest was chosen as the next state. The network then did the same for the next player and so on. Initially, the network chose moves randomly because its weights were initialized randomly. After a sufficient number of games, the network played with more direction, allowing it to explore good strategies in greater depth. The stochastic nature of backgammon allowed the network to thoroughly explore the state space. This self-play learning regime proved effective against Neurogammon's supervised technique.

The next generation of TD-Gammon employed a different input representation that included a set of conceptual features that are relevent to experts. For instance, units were added to encode the probability of a checker being hit and the relative strength of blockades (Tesauro, 2002). With this augmentation of the raw board position, TD-Gammon acheived expert-level play and is still widely regarded as the best computerized player. It is commonly used to analyze games and evaluate the quality of decisions by expert players. Thus, the input representation to TDBP networks is an important consideration when building a network.

There are other factors contributing to the success that Tesauro achieved with TD-Gammon (Tesauro, 1992). Notably, backgammon is a non-deterministic game. Therefore, it has a relatively smooth and continuous state space. This means simply that similar board positions have similar values. In deterministic games, such as chess, a small difference in board position can have large consequences for the state value. Thus, the sort of state value generalization in TD-Gammon would not be as effective and the discontinuities in the chess state space would be harder to learn. Similarly, a danger with learning by self-play is that the network will learn a self-consistent strategy in which it performs well in self-play, but performs poorly against other opponents. This was remedied largely by the stochastic nature of backgammon which allowed good coverage of the state space. Parameter tuning was largely done heuristically with $\lambda = 0.7$ and $\alpha = 0.1$ for much of the training. Decreases in $\lambda$ can help focus learning after the network has become fairly proficient at the task, but these parameter settings are largely the decision of the modeler and should be based on specific considerations for the task at hand.

## 9.4 IMPLEMENTATION

The **tdbp** program implements the TD back propagation algorithm. The structure of the program is very similar to that of the **bp** program. Just as in the **bp** program, pool(1) contains the single bias unit, which is always on. Subsequent pools must be declared in the order of the feedforward structure of the network. Each pool has a specified type: input, hidden, output, and context. There can be one or more input pools and all input pools must be specified before any other pools. There can be zero or more hidden pools and all hidden pools must be specified before output pools but after input pools. There can be one or more output pools, and they are specified last. There are two options for output activation function: logistic and linear. The logistic activation function, as we have seen in earlier chapters, ranges between 0 and 1 and has a natural interpretation as a probability for binary outcome tasks. The linear activation function is equivalent to the net input to the output units. The option of a linear activation function provides a means to learn to predict rewards that do not fall in the range of 0 to 1. This does not exclude the possibility of using rewards outside that range with the logisitic activation function - the TD error will still be generated correctly, although the output activation can never take on values outside of [0, 1]. Lastly, context pools are a special type of hidden pool that operate in the same way as in the **srn** program.

The *gamma* parameter can be set individually for each output pool, and if this value it unspecified, the network-wide *gamma* value is used. Output units also have an *actfunction* parameter which takes values of either 'logistic' or 'linear'. Each pool also has a *delta* variable and an *error* variable. As mentioned previously, the *delta* value will, in general, be a two dimensional matrix. The *error* variable simply holds the error terms for each output unit that lies forward from the pool in the network. Thus, it will not necessarily be of the same size as the number of units in the pool, but it will be the same size as the first dimension of the *delta* matrix. Neither the *error* nor the *delta* parameter is available for display in the network window although both can be printed to the console since they are fields of the pool data structure.

Projections can be between any pool and a higher numbered pool. The bias pool can project to any pool, although projections to input pools will have no effect because the units of input pools are clamped to the input pattern. Context pools can receive a special type of copyback projection from another hidden pool, just as in the **srn** program. Projections from a layer to itself are not allowed. Every projection has an associated eligibility trace, which is not available for display in the network display since, in general, it is a three dimensional matrix. It can, however, be printed to the console and is a field of the projection data structures in the *net* global variable called *eligtrace*. Additionally, each projection has a *lambda* and *lrate* parameter which specify those values for that projection. If either of these parameters is unspecified, the network-wide *lambda* and *lrate* values are used.

There are two modes of network operation - “beforestate” or “afterstate” - which are specified in the .net file of the network when the network is created. In “beforestate” mode, the **tdbp** program (1) presents the current environment state as the network input, (2) performs weight changes, (3) obtains and evaluates possible next states, and (4) sets the the next state. In contrast, the “afterstate” mode instructs the **tdbp** program to (1) obtain and evaluate possible next states, (2) set the selected state as input to the network, (3) perform weight changes, and (4) set the next state. In both cases, learning only occurs after the second state (since we need two states to compute the error). For simple prediction problems, in which the network does not change the state of the environment, the “beforestate” mode is usually used. For any task in which the network modifies the environment, “afterstate” is likely the desired mode. Any SARSA learning should use the “afterstate” mode. Consider briefly why this should be the case. If the agent is learning to make actions, then in “afterstate” mode it will first choose which action to take in the current state. Its choice will then be incorporated into the eligibility trace for that time step. Then the environment will be changed according to the network choice. This new state will have an associated reward which will drive learning on the next time step. Thus, this new reward will be the result of taking the selected action in the associated state, and learning will correctly adjust the value of the initial state-action pair by this new reward. In other words, the reward signal should follow the selection of an action for learning state-action pairs. If you are learning to passively predict, or learning a task in which states are selected directly, then the pre-selection of the next state is unnecessary.

### 9.4.1 Specifying the Environment

The largest difference between the **tdbp** program and the **bp** program is that a Matlab class file takes the place of pattern files. As an introductory note, a class is a concept in object-oriented programming. It specifies a set of properties and methods that a virtual object can have. Properties are simply variables belonging to the object and methods are functions which the object can perform. To provide a simple example, we might wish to define a class called 'dog.' The dog class might have a property specifying, e.g., the frequency of tail wags and a method called 'fetch.' This class is said to be instantiated when a dog object is created from it. An instantiated class is called an 'object' or 'instance' and it differs from a class in that it owns its own properties and methods. So we might instantiate many dog objects, each with different frequency of tail wagging. Thus, a class specifies a data structure and an object is a discrete instance of that data structure. For more information about object-orient programming in Matlab, please consult the Matlab Object-Oriented User Guide (`http://www.mathworks.com/help/techdoc/matlab_oop/ug_intropage.html`).

Matlab class files are Matlab .m files that specify a Matlab object which has both data fields, like a data structure, and *methods*, which provide functionality. We call this class file an "environment class." The environment class is specified by the modeler to behave in a functionally equivalent way to the task environment that the modeler wishes to model. The environment class provides all input patterns to the network, all reward signals, and all possible next states for the network to evaluate. As such, environment classes must adhere to a standard interface of methods and properties so that it can interact with the **tdbp** program. The *envir_template.m* file is a skeletal specification of this standard interface. The environment class file is specified to **pdptool** just as .pat files are in the **bp** program. You can select an environment file (with .m extension) in the pattern selection dialogue box by clicking the Load Pattern button in the main pdp window and changing the file type to .m. The **tdbp** program instantiates the class as an object named "envir," which is a global variable in the workspace, just like the "net" variable. We will briefly review the functionality of the environment class, but please refer to the *envir_template.m* file for a full explanation of the requisite functionality. It is recommended that a modeler wishing to create his own environment class use the template file as a start.

The interface for the environment class specifies one property and ten methods, two of which are optional. The required property is a learning flag, *lflag*. Note that the **tdbp** program has its own *lflag* which can be set in the training options dialogue. If either of these variables is equal to 0, the network will skip training (and computing eligibilities) for the current time step. This allows you to control which steps the network takes as training data from within your environment class, which is particularly useful for game playing scenarios in which the network makes a move and then the game state is inverted so that the network can play for the other side. We only want to train the network on a sequence of states which only one of the players would have to play. Otherwise, we would only be learning to predict the sequence of game states of both players, not to move for a single player. This can be accomplished by toggling the *lflag* variable within the environment class for alternating steps.

The ten environment class methods are broadly divided into two groups: control methods and formal methods. This division is not important for the implementation of these methods, but it is important for conceptual understanding of the relationship between the **tdbp** program and the environment class. Control methods are those which are necessary to control the interaction between the environment class and the **tdbp** program - they have no definition in the conceptual description of RL. Formal methods, however, do have conceptual meaning in the formal definition of RL. We briefly describe the function of each of these methods here. It is suggested that you refer back to Figure 9.4.1 to understand why each method is classified control vs. formal.

*Note about Matlab classes.* Matlab classes operate a little differently from classes in other programming languages with which you might be familiar. If a method in your class changes any of the property variables of your object, then that method must accept your object as one of its input arguments and return the modified object to the caller. For example, suppose you have an object named `environment` who has a method named `changeState` which takes an argument `new_state` as its input and sets the object's state to the new state, then you must define this method with two input arguments like the following:

```
function obj = changeState(obj, new_state)
obj.state = new_state;
end
```

You should call this method in the following way:

```
environment = environment.changeState(new_state);
```

When you call this method in this way, `environment` gets send to the method as the first argument, and other arguments are sent in the order that they appear in the call. In our example method, the variable `obj` holds the object `environment`. Since you want to modify one of `environment`'s properties, namely the `state` property, you access it within your method as `obj.state`. After you make the desired changes to `obj` in your method, `obj` is returned to the caller. This is indicated by the first `obj` in the method declaration `obj = changeState(obj, new_state)`. In the calling code, you see that this modifed object gets assigned to `environment`, and you have successfully modified your object property. Many of the following methods operate this way.
*Control Methods:*

- `environment_template()` - This is the constructor method. It it called when the environment object is created and returns the new environment instance. You should rename this function as the name of your environment - it must have the same name as the name of the class at the top of the environment class file and the filename itself. In general, this function only calls the `construct()` function and returns the returned value. As you might guess, the `construct()` method does the real work of setting up the environment. We have separated these functions for two reasons. First, the `reset()` function can just call `construct()` to reset the environment. Second, you can intitialize variables in the `environment_template()` function which will persist over the course of training. You can think of there being two types of variables within your environment class: persisting and non-persisting. Non-persisting variables are initialized in the `construct()` method. Thus, they are reset each time `reset()` is called, which is at the end of every training episode. Persisting variables can be initialized in `environment_template()` and survive over the course of a training or testing run. You can use them to collect useful data about each episode over the course of a run by saving the relevant non-persisting variables in the persisting variables within the `reset()` function before it calls `construct()`. This data can later be manipulated and displayed with the `runStats()` function, described below.

- `construct(obj) -` As described above, this function does the work of initializing the environment by intializing any relevant variables. These variables will be reset at the end of every training episode. Also, be sure to set the *lflag* variable here to turn on learning with `obj.lflag = 1;`.

- `reset(obj) -` This function resets the state of the evironment object to prepare for another training episode by calling `construct(obj)`. If there is any data from the last episode that you wish to save in persisting variables, you must do it here before calling `construct(obj)`.

- `add_to_list = runStats(obj) -` This function provides a means by which to manipulate your persisting variables to extract relevant statistics and then return these statistics to be displayed in the pattern list in the network window. This function is called at the end of a training run, but it is optional and it can be turned on and off with the Runstats checkbox in the training options window. The return value, `add_to_list`, must be a cell array of strings. Each string in the cell array will be displayed on its own line in the pattern list. Refer to Matlab documentation for more information about cell arrays.

- `next_state = doPolicy(obj, next_states, state_values) -` This is an optional function which allows you to implement your own policy for choosing the next action or state based on the network's outputs. As described below, all of the built-in policies use only the activation of the first unit of the first output pool in selecting actions. If you desire to use multiple output units in determining the next state, then you need to implement this function. For example, perhaps your network is learning to play a game and you have two output units, one estimating the likelihood of winning (i.e. it receives a reward when you win) and the other estimating the likelihood of tying (i.e. it receives a reward when you tie). There may be circumstances in which your likelihood of winning is very low, but you are very likely to pull out with only a tie instead of a loss. In this case, you might implement a `doPolicy` function to consider the relative activation in these two units in choosing your play. This function takes two input arguments from the **tdbp** program. `next_states` is a matrix of the possible next states (which you have already given the network with the `getNextStates()` function, see below) where each row contains the input to the network. `state_values` contains a matrix of the network output for each of the states in `next_states`, where each row of `state_values` is the network output for the corresponding row of `next_states`. Your function should use these values to select a state in `next_states`, then return the selected state to the **tdbp** program by assigning the variable `next_state`.

- `endFlag = isTerminal(obj)` - This required function returns a logical value (0 or 1) indicating whether or not the environment is currently in its terminal state. Thus, this function must test for some terminal condition and return the result of the test. For example, if your environment only runs for exactly 10 steps, then you should keep track of how many steps have elapsed in a property within your object and simply test if 10 states have passed. Likewise, if an episode ends when some terminal state is reached, then `isTerminal` must test whether the current state of your environment is identical to this terminal state.

*Formal Methods:*

- `reward = getCurrentReward(obj)` - This function returns a vector of reward values to the **tdbp** program which are used to compute the TD error for the current time step. `reward` must be the same length as the total number of output units in the network, although for many applications you will only use one unit. The way that you determine the reward for a given state is largely dependent on the dynamics of the particular environment you are implementing.

- `state = getCurrentState(obj)` - This function simply returns an input vector to the network which represents the current state of the environment. Naturally, the vector returned in `state` must be the same length as the total number of input units in the network.

- `states = getNextStates(obj)` - This function returns a matrix of the next states or actions which are possible given the current state. `states` is a matrix each of whose rows is a possible next state; the same constraint on the length of these state vectors applies. The network will choose one of these states based on its output for each of the states and the policy you are using. If you have chosen to use the user-defined `doPolicy` method described above, then the network will simply take the `states` matrix returned by `getNextStates` and pass it to your `doPolicy` function - you need do nothing special.

- `obj = setNextState(obj, next_state)` - This function is the means by which the network changes the environment state. The **tdbp** program will pass the chosen next state to this function in the `next_state` argument. If you are learning action-values, then you need to locate the segment of `next_state` which represents actions and change the environment appropriately for the action unit that is on. The details of how to change the state of the environment based on the `next_state` argument will depend on the details of your application. Note that if you are using the *defer* policy, the next state is not determined by the network output. In this case, `next_state` will be a vector of zeros and your `setNextState` implementation should ignore it and simply set the environment state based on other criteria.

## 9.5 RUNNING THE PROGRAM

The **tdbp** is used in much the same way as the **bp** program. Here we list important differences in the way the program is used.

*Pattern list.* The pattern list serves the same role as it does in the **bp** program, except that patterns are displayed as they are generated by the environment class over the course of learning. There are two display granularities - *epoch* and *step* - which are specified in the drop down lists in both the Train and Test panels. When *epoch* is selected, the pattern list box will display the number of time steps for which an episode lasted and the terminal reward value of the episode, e.g. "steps: 13 terminal r: 2". When *step* is selected, the pattern list box will indicate the start of an episode and subsequent lines will display the input state of the network on each time step and the associated reward received in that state, e.g. "1 0 .5 0 1 r: 1." When an episode ends, a summary line containing the number of steps and the terminal reward will be displayed. Additionally, in the training and testing options window, there is a checkbox labelled "Show Values," which turns on the show value mode. In the show value mode, all states that the network evaluated in selecting the next state or action are listed under the current state, along with the associated network output for each. This is only available in *step* mode and allows a finer grained view of the state values the network is choosing among at each step. Just as in the **bp** program, you can specify the number of steps or epochs to wait before updating the pattern list and network viewer. The Reset and Newstart commands work just as in the **bp** program.

*Training options.* The training options window allows you to set the number of epochs to train the network; the global *lambda*, *gamma*, and *lrate* parameters; the learning granularity, the *wrange* and *wdecay* parameters; the *mu* and *clearval* parameters for context layers; and the state selection policy. There are six policy options in the Policy drop down menu. The *greedy* policy always selects the state with the highest value. The *egreedy* policy selects the highest value with probability 1 - *epsilon.* The *epsilon* value for this policy is right under the policy drop down menu. The *softmax* policy selects the next action by the Boltzmann distribution as already discussed. The *temp* value is specified by clicking the "annealsched" button. This will open a new window in which you can specify an annealing schedule for the *temp* value, and which works the same way as the annealing schedule in the **cs** program. If you only wish you set a constant *temp* value, then enter it in the "Initial temperature" text box and click Done. The *linearwt* policy chooses the state $s$ with probability $p(s) = V(s)/\sum_x V(x)$, where the denomenator is the sum of all next state values. This policy is only available for networks with the logistic function at the output layers. The *defer* policy merely calls the `setNextState()` method of the environment class with a vector of zeros, which should be ignored in the implementation of that function. This allows the environment to change its own state, and is used in passive prediction problems where the network does not select next states. Lastly, the *userdef* policy passes the same matrix of next states returned by the `getNextStates()` method of the environment class along with a corresponding matrix of the network output for each of the states to the `doPolicy()` method in the environment class. This allows the user to write a custom policy function, as described above. Note that all of the above policies will only use the value estimate on the first output unit of the lowest numbered output pool. If the task requires that multiple values be predicted by more than one ouput unit, and state selection should take multiple value estimates into account, then the *userdef* policy is required. Recall that TD-Gammon employed four output units to predict different possible outcomes of the game. One might wish to take the values of all these output units into account in choosing an action.

*Testing options.* The testing options window is much the same as the training options window, but only values relevant to testing are displayed. It is possible, especially early in training, for the network to cycle among a set of states when testing in with the *greedy* policy, resulting in a infinite loop. To prevent this, the *stepcutoff* parameter defines the maximum number of steps in an episode. When the network reaches the cutoff, the episode will be terminated and it will be reported in the pattern list.

## 9.6 EXERCISES

### Ex9.1. Trash Robot

In this exercise we will explore the effect of the environment reward structure and parameter values on the behavior of a RL agent. To begin, open the file *trashgrid.m* by locating it in the *tdbp* folder, right clicking it, and clicking “Open as Text.” The file will open in a new window. This is the environment class for the excercise and you will want to keep it open because you will need to edit it later. To begin the exercise, type *start3by3grid* to the Matlab command prompt. This will open the *3by3grid.net* network and set up the training settings. You can view the current training settings by opening the training options window. The network is a SARSA network and is set up to use the softmax policy for action selection with a constant temperature of 1. In the network window, you will see a 3 by 3 grid and another set of 4 units labeled “action.” These four units represent the four possible actions that can be taken in this grid world: moving up, down, left, or right. Thus, there are a total of 13 input units in this network. You also see that there are 6 hidden units and a single output unit whose value is reported in decimal form.

The job of the agent in the following exercises will be to make the necessary moves to enter the terminal square on the grid, where it will receive a reward based on how long it took. There is also a single piece of trash which the agent, who can be thought of as a simple trash-collecting robot, can pick up by entering the square where the trash is. Activate the test panel and set the Update After drop down to Step. Then click the Step button a few times to observe an episode. You will see that the unit in the upper left corner of the grid turns on - this is unit (1,1). The grid is numbered 1 to 3 from the top down and 1 to 3 from the left to right. This unit marks the current location of the agent in the grid. At each time step, the agent will move to a neighboring square in one of the four cardinal directions. The unit in the bottom left corner of the grid will have a value of .5. This indicates that there is a piece of trash in that square. The terminal square is unmarked, but it is position (2,3) (two down, three right). The episode ends when the agent enters this square. If you continue to step through the episode, you will see that the robot moves randomly. Try clicking Newstart and observe how the path of the robot changes. The robot may cycle between squares or stumble into the terminal square.

Q.9.1.1.

> Find the `getCurrentReward()` function in the *trashgrid.m* file. Try to figure out the reward structure of this task by understanding this function. If the robot took the fastest route to the terminal square, what reward would it receive? Explain the rewards the robot can receive and how.

Set the Update After text box to 10 epochs and train the network for 350 epochs. You will see a summary for each epoch appear in the patterns list. You will notice that initially the episodes are long and usually end with 0 reward. By the end of training, you should see improvement.

Q.9.1.2.

> After training the network, switch to the test panel and step through a few episodes. Since the test policy is *greedy*, every episode will be the same. What path does the robot take? Was your prediction about the reward correct? You can try clicking New Start and training the network again from scratch to see if the behavior changes. Do this a few times and report on what you see. Does the robot use the same path each time it is trained? If not, what do the different paths have in common?

After observing the network behavior with the *greedy* policy, switch the test policy to *softmax* and make sure the temperature is set to 1.

Q.9.1.3.

> Step through a few episodes. What changes do you notice? Since we are now using softmax to choose actions, the behavior of the robot across episodes will vary. It may help to run more than one episode

> at once. Open the test options window and change *nepochs* to 50 and click the Runstats checkbox, then switch the update mode to 'epoch' and run the simulation. You will notice that after the entire test run, a line will appear at the bottom of the pattern list reporting the average terminal reward over the runs and another line reporting the proportion of episodes in which the robot picked up the trash. You might try running a few more runs, each time observing the average reward over the episodes. Explain what has changed in the robot's behavior, its average reward, and how it is related to the action choice policy for the test runs.

To further investigate these changes, open the test options window, set *nepochs* to 1, check the 'Show values' check box, and click OK. This results in each possible next state that the network is choosing between, along with the network's output for that state, to be printed below the current state in the pattern list.

Q.9.1.4.

> Step through a few episodes and observe the action values that are displayed in the pattern list. It will help to remember that the actions are encoded at the end of the input vector, with the last four positions representing up, down, left, and right respectively. What is implicitly encoded in the relative value estimates at each step? How does this lead to the bistable behavior that the network is exhibiting?

Now switch to the training panel, reset the newtork, and inspect the reward function in the *trashgrid.m* environment class file.

Q.9.1.5.

> First, be sure to remember the current `getCurrentReward` function, as you will have to restore it later - copying and pasting it elsewhere is a good idea. Change one of the assignments to `reward` in order to change which of the two paths the robot prefers to follow. Then train the network to 350 epochs from scratch and observe its behavior under the greedy policy. Did the change you made result in your expected change in the robot's behavior? Why?

Now revert the changes you just made to the reward function and Reset the network again.

Q.9.1.6.

> Keeping in mind the function of the *gamma* parameter in calculating returns, change its value in order to induce the same pattern of behavior that you obtained my manipulating the reward function.

> Train the network after any change you make, always remembering to Reset between training runs. It may take a few tries to get it right. Test the effect of each change you make by stepping through an episode with the greedy policy. If you find that the network cycles between squares indefintiely during testing, then it likely needs more epochs of training. What change did you make and why did this change in behavior occur? Use your knowledge of the return calculation, the role of *gamma*, the reward function, and the layout of the grid to explain this change in behavior.

We have intentionally kept this grid world very small so that training will be fast. Your are encouraged to modify the *trashgrid.m* environment and make a new network to explore learning in a larger space. You may also wish to explore behavior with more than one piece of trash, or perhaps trash locations that change for each episode.

# Appendix A

# PDPTool Installation and Quick Start Guide

PDPTool is a neural network simulator for Matlab that implements the models described in Parallel distributed processing: Explorations in the microstructure of cognition (Rumelhart et al., 1986; McClelland et al., 1986). This program is a teaching aid for courses in Parallel Distributed Processing.

This document describes how to install and run PDPTool on your computer. For instructions on using the software, see the *PDPTool User's Guide*, Appendix C.

If you encounter difficulties with installation, send email to: pdplab-support@stanford.edu.

NOTE: The version of the software available here does not work on MATLAB versions R2014a or later (see immediately below for details).

## A.1 System requirements

l platforms / Matlab versions The current version works on Linux, Mac, and Windows platforms running versions of Matlab from about 2007 through 2013. Currently the software does not work with versions R2014a or R2014b

## A.2 Installation

Source files for PDPTool are located at http://web.stanford.edu/group/pdplab/pdptool/pdptool.zip.

Install PDPTool using the following steps.

1. Download the archive file "pdptool.zip" to your desktop.
2. Extract the archived files into a new folder called "pdptool".
3. Start Matlab.

4. In Matlab, set your path variable to point to PDPTool using the following steps.

   (a) From the File menu, select Set path. A dialog box opens.

   (b) Click the Add with subfolders button. A directory browser window opens.

   (c) Locate the folder called pdptool. Select it and click OK.

   (d) Click the Save button on the set path dialog box to save the path for future sessions.

   (e) Click the Close button.

5. Set your command history preferences using the following steps.

   (a) From the File menu, select Preferences. A dialog box opens.

   (b) Select Command History from the list of options on the left. This displays the current command history settings.

   (c) In the Saving section of the history settings, select Save after [n] commands, where [n] is a numerical field.

   (d) Change [n] to 1.

   (e) Click OK.

6. At the Matlab command prompt, type "pdp" to start the program.

## A.3 Using PDPTool at a Stanford Cluster Computer

If you use PDPTool at a Stanford cluster computer, install the program and save your work to your AFS filespace, which includes the desktop.

You will be unable to save your Matlab settings. At the beginning of each session, repeat steps 4 and 5 of Installation to set the correct path variable and command history preferences.

In the Set Path dialog box, do not click the Save button. When you click the Close button, a dialog box appears to ask if you would like to save the path. Click No.

## A.4 Using the software

To start working with the exercises, change your working directory to the subdirectory of the program you wish to run. For the first exercise, move to the **iac** subdirectory of your pdptool directory.

Instructions on using PDPTool for the first exercise 2.1 are available in the Interactive Activation and Competition chapter. Also consult the *PDPTool User's Guide*, Appendix C, for overall documentation.

## A.5 Notes when using Matlab 7.3 r2006b on OSX

The graphical interface of PDPTool for this version of Matlab is imperfect. To set the value of the external input to a unit, right-click just outside the units square graphic instead of inside it.

# Appendix B

# How to create your own network

In this appendix, we describe the steps you need to take to build your own network within one of the PDPtool simulation models. In the course of this, we will introduce you to the various files that are required, what their structure is like, and how these can be created through the PDPtool GUI. Since users often wish to create their own back propagation networks, we've chosen an example of such a network. By following the instructions here you'll learn exactly how to create an 8-3-8 auto-encoder network, where there are eight unary input patterns consisting of a single unit on and all the other units off. For instance, the network will learn to map the input pattern

```
1 0 0 0 0 0 0 0
```

to the identical pattern

```
1 0 0 0 0 0 0 0
```

as output, through a distributed hidden representation. Over the course of this tutorial, you will create a network that learns this mapping, with the finished network illustrated in Figure B.6.

Creating a network involves four main steps, each of which is explained in a section:

1. Creating the network itself (Appendix B.1)
2. Creating the display template (Appendix B.2)
3. Creating the example file (Appendix B.3)
4. Creating a script to initialize the network (Appendix B.4)

While this tutorial does a complete walkthrough of setting up an auto-encoder in PDPtool, in the interest of brevity, many of the commands and options of PDPtool are left unmentioned. You are encouraged to use the *PDPtool User's Guide* (Appendix C) as a more complete reference manual.

## B.1 Creating the network itself

The first thing you need to do is create your own network specification. A network specification is a file with a '.net' extention that specifies the architecture of your network. Such a file can be created by typing matlab commands, and can be edited using the Matlab editor. However, the easiest way to create such a file is to use the GUI.

It may be best to create a new directory for your example. So, at the command line interface type

```
mkdir encoder838
```

Then change to this directory

```
cd encoder838
```

and launch the pdp tool, by typing

```
pdp
```

In the main pdp window, select "Create..." from the Network pull-down menu. In the "Network Name" box, enter "encoder838" (or whatever you want to name your network). The "Network Type" is Feed-forward Back propagation.

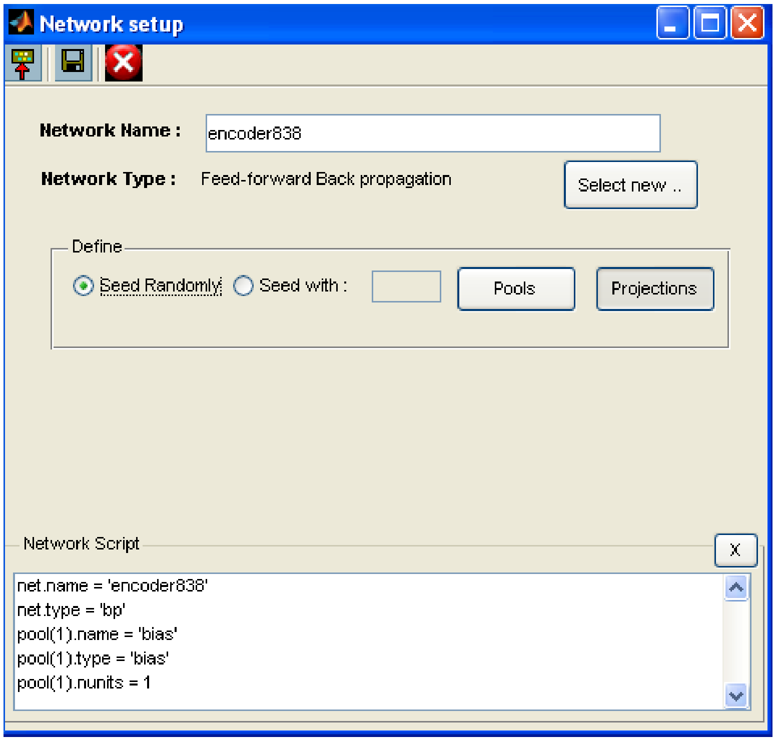
It might be useful to see the program create the ".net" file as we go along. Click the "View Script" button in the top-left corner of the window. Your "Network setup" box should look something like the one in Figure B.1. Note that one pool of units – pool(1), the bias pool, is already created for you. This pool contains a single unit that always has an activation of 1; connections from this pool to other pools implement bias weights in the network.

### B.1.1 Defining the Network Pools

It is now time to define the pools you want in your network. It is good practice to do this before defining the projections. Click the "Pools" button. We must define each of the 3 pools we want (input, hidden, and output) individually. Under "Select pool type" choose "input", then enter the name as "input" with 8 units, clicking "Add" when you are finished. You must do this for the hidden and output pool as well, using 3 and 8 units respectively. At any point during this process, to see the pools previously defined, follow the directions in this footnote.[1]

The network will now have four pools: bias (pool 1), input (pool 2), hidden (pool 3), and output (pool 4).

[1] At any point, to see the pools previously defined, the user can right click on the pool Name edit box. This will replace the edit box with a pop-up menu of all currently defined pools, selecting a pool name will show the number of units and the pool type that was set for it which can then be edited if desired. Right clicking on the pop-up menu will change it back into an edit box.



Network Name : encoder838

Network Type : Feed-forward Back propagation

● Seed Randomly ○ Seed with :

```
net.name = 'encoder838'
net.type = 'bp'
pool(1).name = 'bias'
pool(1).type = 'bias'
pool(1).nunits = 1
```

Figure B.1: The Network Setup window. This is the first step in setting up our feed-forward back propagation network.

### B.1.2 Defining the Projections

Projections are defined in a similar manner. For your back propagation network, you should define the standard feedforward projections from input to hidden to output, as well as projections from the bias units. Let's start with the bias projections. A projection is defined by selecting a pool from the sender list, one from the receiver list and then specifying how the weights will be initialized from the drop-down menu. Start by having the "Sender" be the bias and the "Receiver" be the hidden units. For a back propagation network, you should select your initial weights to be "Random" from the pull-down menu.[2] However, here are what all the options do:

**scalar** This option starts all the weights with the same initial scalar value, which you define in the given box.

**random** The weights are initialized as random draws from a uniform distribution from -0.5 to .5. This range can be set by the "Wrange" parameter in the training options once your network is up and running. The weights will be drawn from $-wrange/2$ to $wrange/2$.

**prandom** Weights are positive random numbers, initialized from 0 to wrange.

**nrandom** Weights are negative random numbers, initialized from -wrange to 0.

Once you set the options for the bias to hidden unit projections, click "OK." In the network script, this should add some code that looks like this:

```
pool(3).proj(1).frompool = 'bias'
pool(3).proj(1).constraint_type = 'Random'
pool(3).proj(1).lrate = NaN
```

In total, your network needs 4 projections, all defined with "Random" initial weights.[3]

1. bias to hidden
2. bias to output
3. input to hidden
4. hidden to output

[2] For network types with user-specified (as opposed to learned) connections, the user can have the network read the fixed connection values from a file containing a series of lines each containing a series of entries separated by white space. These can be real numbers, or any of these three shorthand entries: . (0), + (+1) or − (−1). The program also provides a facility for creating and saving such arrays by hand (the set by hand item in the Constraint popup menu). Once entered the user clicks 'Done', and saves the array to a file; the user must then select the 'read from file' option from the constraint popup and finally click 'Ok' to finalize the specification of the projection.

[3] Each projection can have its own learning rate; usually, however, none is specified, and in this case the network uses the network-wide learning rate parameter.

Once the projections are defined, you are done defining your network, and you are ready to continue with the other steps. Click the save button at the top of the window (the floppy disk icon), and save the file as 'encoder838.net'. Our encoder838.net file is in Figure B.2 so you can check if yours is the same.

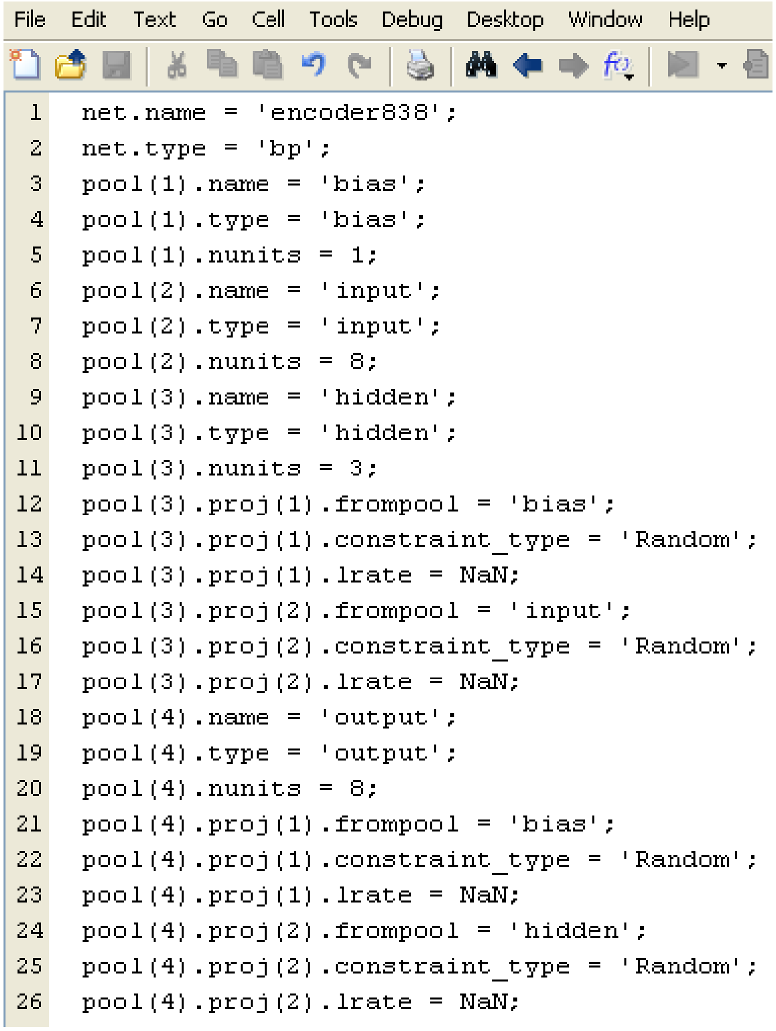
## B.2 Creating the display template

You are now ready to create the display template, which is the visualization of your network while it's running. In the pdp window, select "Select display items..." from the "Display" drop-down menu. The window is broken into the panels: the left panel is a tree-like structure of network objects that you can add to the the display template and the right panel is your current selection of such objections.

Start by clicking "+ net: struct" since the "+" indicates it can be expanded. This shows many of the network parts. You can add networks parts that you want displayed on the template. For each item you want displayed, you need to separately add a "Label" and "Value" to the Selected Items panel. The Value will be a vector, matrix, or scalar that displays the variable of interest. The Label is a floating text box that allows you to indicate which item is which on the display.

What items do you want in your template? For any network, you want the activations of the pools displayed (except for the bias pool). This allows you to see the pattern presented and the network's response. For many networks, if the pools are small enough (say less than 10 units each), you may want to display the weights to see how the network has solved the problem. Otherwise, you can ignore adding the weights, and you can always use the Matlab command window to view the weights if wanted during learning.

For our auto-encoder, we will want to display the pool activations, the target vector, the weights, and some summary statistics. Each of these items is also going to need a label. We'll walk you through adding the first item, and then the rest are listed so you can add them yourself. Let's start by adding the activation of the input layer, which is pool(2). Expand the pool(2) item on the left panel, and highlight the activation field. The click the "Add" button. You must now select whether you want to add a Label or Value. We will want to add both for each object. Thus, since Label is already selected, type the desired label, which should be short (we use "input act"). Click "Ok". The label you added should appear in the right panel. All we did was add a text object that says "input act," now we want to add the actual activation vector to the display. Thus, click add again on the pool(2).activation field, select Value, and set the Orientation to Vertical. The orientation determines whether the vector is a row or column vector on the template. This orientation can be important to making an intuitive display, and you may want to change it for each activation vector. Finally, set the vcslope parameter to be .5. Vcslope is used to map values (such as activations or weights) to the color map, controlling the sensitivity of the color around zero. We use .5 for activations and .1 for weights in this network.



```
net.name = 'encoder838';
net.type = 'bp';
pool(1).name = 'bias';
pool(1).type = 'bias';
pool(1).nunits = 1;
pool(2).name = 'input';
pool(2).type = 'input';
pool(2).nunits = 8;
pool(3).name = 'hidden';
pool(3).type = 'hidden';
pool(3).nunits = 3;
pool(3).proj(1).frompool = 'bias';
pool(3).proj(1).constraint_type = 'Random';
pool(3).proj(1).lrate = NaN;
pool(3).proj(2).frompool = 'input';
pool(3).proj(2).constraint_type = 'Random';
pool(3).proj(2).lrate = NaN;
pool(4).name = 'output';
pool(4).type = 'output';
pool(4).nunits = 8;
pool(4).proj(1).frompool = 'bias';
pool(4).proj(1).constraint_type = 'Random';
pool(4).proj(1).lrate = NaN;
pool(4).proj(2).frompool = 'hidden';
pool(4).proj(2).constraint_type = 'Random';
pool(4).proj(2).lrate = NaN;
```

Figure B.2: The ‘encoder838.net’ file created through the Network Setup window. Double-check to see if yours is the same.

Details of this parameter are in the *PDPtool User's Guide* (Appendix C).

For the auto-encoder network, follow the orientations we have selected. If you make a mistake when adding an item Value or Label, you can highlight it in the right panel and press "Remove".

Now it's time to add the rest of the items in the network. For each item, follow all the steps above. Thus, for each item, you need to add a Label and then the Value with the specified orientation. We list each item below, where the first one is the input activation that we just took care of. Your screen should look like Figure B.3 when you are done adding the items (however, this screen does not indicate whether or not you have the orientations or transposes the same way that I do, but this will make a difference in a second).

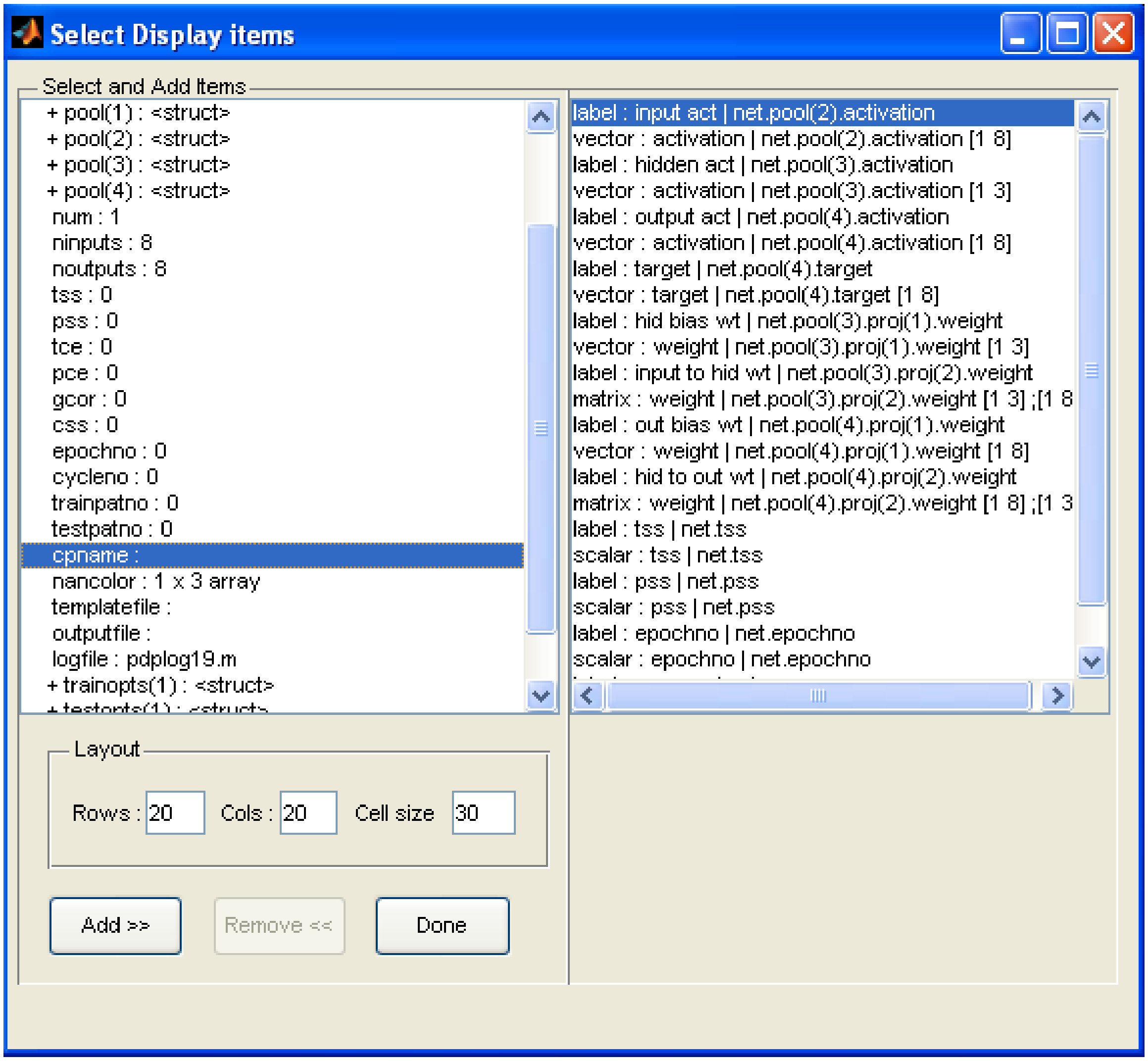
```
pool(2) activation (Label = input act; Orientation = Vertical; vcslope = .5)
pool(3) activation (Label = hidden act; Orientation = Horiz; vcslope = .5)
pool(4) activation (Label = output act; Orientation = Vertical; vcslope = .5)
pool(4) target (Label = target; Orientation =  Vertical; vcslope = .5)
pool(3).proj(1) weight (Label = hid bias wt; Orientation = Horiz; vcslope = .1)
pool(3).proj(2) weight (Label = input to hid wt; Transpose box checked; vcslope = .1)
pool(4).proj(1) weight (Label = out bias wt; Orientation = Vertical; vcslope = .1)
pool(4).proj(2) weight (Label = hid to out wt; Transpose box Un-checked; vcslope = .1)
tss (Label = tss)
pss (Label = pss)
epochno (Label = epochno)
cpname (Label = cpname)
```

After adding these items, click "Done" if your screen looks like Figure B.3. The "Set display Positions" screen should then pop-up, where you get to place the items on the template. An intuitive way to visualize this encoder network is shown in Figure B.4. To place an item on the template, select it on the left panel. Then, right click on the grid to place the item about there, and you can then drag to the desired position. If you want to return the item to the left panel, click "Reset" with the item highlighted.

## B.3 Creating the example file

An example file is the file that defines the stimuli for the network. In the case of a feed-forward back propagation network like the one we are creating, this consists of defining a set of input-teacher pattern pairs. The file should be a series of lines formated as follows: "name input teacher", where the name is the name of each pattern. The set of 8 unary input patterns, and their teachers (which is the same as the input pattern) are shown below. All you need to do is simply copy this segment of text into a text file, and save it as "unary.pat". To create a new text file within Matlab, click the "New M-File" button at the top left corner of the Matlab window.

```
p1 1 0 0 0 0 0 0 0   1 0 0 0 0 0 0 0
```



Select Display items

Select and Add Items

```
+ pool(1) : <struct>
+ pool(2) : <struct>
+ pool(3) : <struct>
+ pool(4) : <struct>
  num : 1
  ninputs : 8
  noutputs : 8
  tss : 0
  pss : 0
  tce : 0
  pce : 0
  gcor : 0
  css : 0
  epochno : 0
  cycleno : 0
  trainpatno : 0
  testpatno : 0
  cpname :
  nancolor : 1 x 3 array
  templatefile :
  outputfile :
  logfile : pdplog19.m
+ trainopts(1) : <struct>
+ testopts(1) : <struct>
```

```
label : input act | net.pool(2).activation
vector : activation | net.pool(2).activation [1 8]
label : hidden act | net.pool(3).activation
vector : activation | net.pool(3).activation [1 3]
label : output act | net.pool(4).activation
vector : activation | net.pool(4).activation [1 8]
label : target | net.pool(4).target
vector : target | net.pool(4).target [1 8]
label : hid bias wt | net.pool(3).proj(1).weight
vector : weight | net.pool(3).proj(1).weight [1 3]
label : input to hid wt | net.pool(3).proj(2).weight
matrix : weight | net.pool(3).proj(2).weight [1 3] ;[1 8
label : out bias wt | net.pool(4).proj(1).weight
vector : weight | net.pool(4).proj(1).weight [1 8]
label : hid to out wt | net.pool(4).proj(2).weight
matrix : weight | net.pool(4).proj(2).weight [1 8] ;[1 3
label : tss | net.tss
scalar : tss | net.tss
label : pss | net.pss
scalar : pss | net.pss
label : epochno | net.epochno
scalar : epochno | net.epochno
```

Layout

Rows : 20 Cols : 20 Cell size 30

Figure B.3: The Select Display Items window. When creating your template, this is the screen where you add network items to the display. For the autoencoder we are creating here, the list of added items should look like this (the cpname label and scalar are there but not visible).

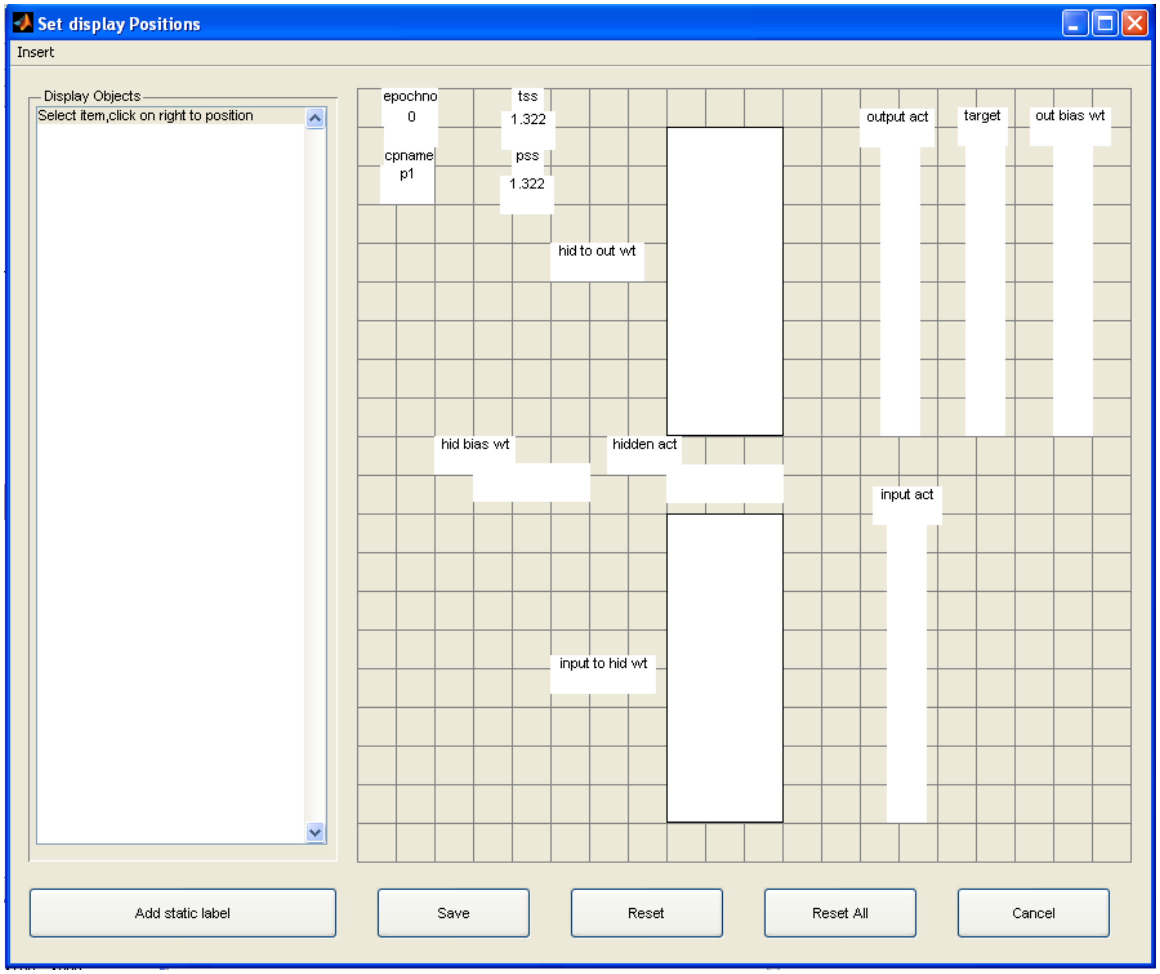


Figure B.4: The Set Display Positions window. Here, you place the items you selected in Figure B.3 on the display, which is the panel you see when your network is running. A recommended layout for the encoder network is displayed here.

```
p2 0 1 0 0 0 0 0 0    0 1 0 0 0 0 0 0
p3 0 0 1 0 0 0 0 0    0 0 1 0 0 0 0 0
p4 0 0 0 1 0 0 0 0    0 0 0 1 0 0 0 0
p5 0 0 0 0 1 0 0 0    0 0 0 0 1 0 0 0
p6 0 0 0 0 0 1 0 0    0 0 0 0 0 1 0 0
p7 0 0 0 0 0 0 1 0    0 0 0 0 0 0 1 0
p8 0 0 0 0 0 0 0 1    0 0 0 0 0 0 0 1
```

## B.4 Creating a script to initialize the network

Now, all of the heavy lifting is done! It is possible to run your network using the main pdp window, loading the script, the patterns, the template, and setting the train options. However, it is much easier if you have a script that does all of this for you. Then, your network can be setup with all the right parameters with a single command.

There are two ways to do this. (1) By setting up and launching your network in the main pdp window, PDPTool will log the underlying commands in a .m file that you can run later to repeat the steps. (2) You can simply type the appropriate setup commands by hand in a .m file.

Here, we'll describe describe the first method and provide the generated .m file (Figure B.5). Thus, you can also opt to simply type the content of Figure B.5 by hand if you prefer the second method.

Close the pdp program entirely. When you open the pdp program, your next series of actions will be recorded in the log file. First, type

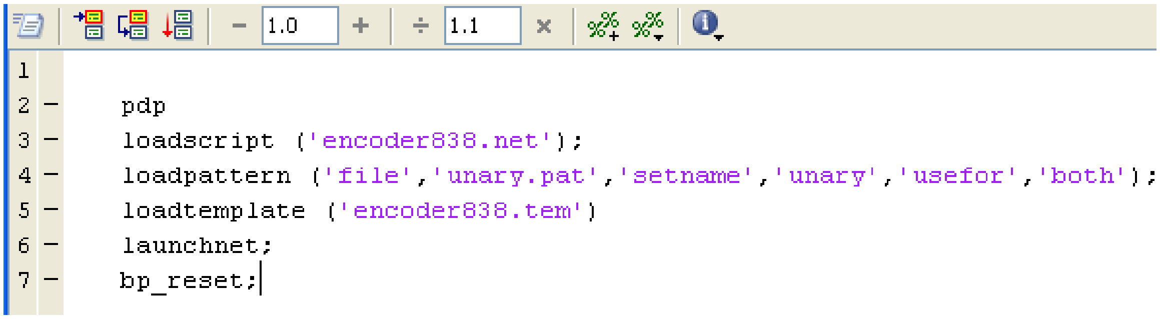
```
pdp
```

which starts the pdp program. First, click "Load script", and select the encoder838.net file that contains your network setup (Section B.1). Second, click "Load pattern" to load the unary.pat file you created (Section B.3) as both the testing and training patterns. Third, click the Display menu and select "Load Template," selecting the encoder838.tem you created in Section B.2.

Also, you will want to select Training options that are reasonable for this network, which will become the parameters used when the network starts up with this script. Select the "Network" menu then "Training options." The defaults are fine for this network, but they may have to be changed when setting up a custom network. For advice on setting some crucial parameters, such as the *lrate* and *wrange*, see the hints for setting up your own network in the back propagation chapter (Ex 5.4 *Hints*). When done adjusting training options, click "Apply" and then "OK."

We are ready to launch the network, which is done by selecting "Launch network window" from the "Network" menu. The Network Viewer should pop up, resembling Figure B.6 (except that the network in this figure has already been trained).

Finally, click "Reset," which is necessary for certain parameters such as "wrange" to take effect. Now, all the actions you have just completed should



```

    pdp
    loadscript ('encoder838.net');
    loadpattern ('file','unary.pat','setname','unary','usefor','both');
    loadtemplate ('encoder838.tem')
    launchnet;
    bp_reset;
```

Figure B.5: This is a sample 'bpencoder.m' file that we created in Section B.4. This script will load the network parameters, the 'unary.pat' training and testing patterns, and the template created in Section B.2. Training options can be specified in the settrainopts function.

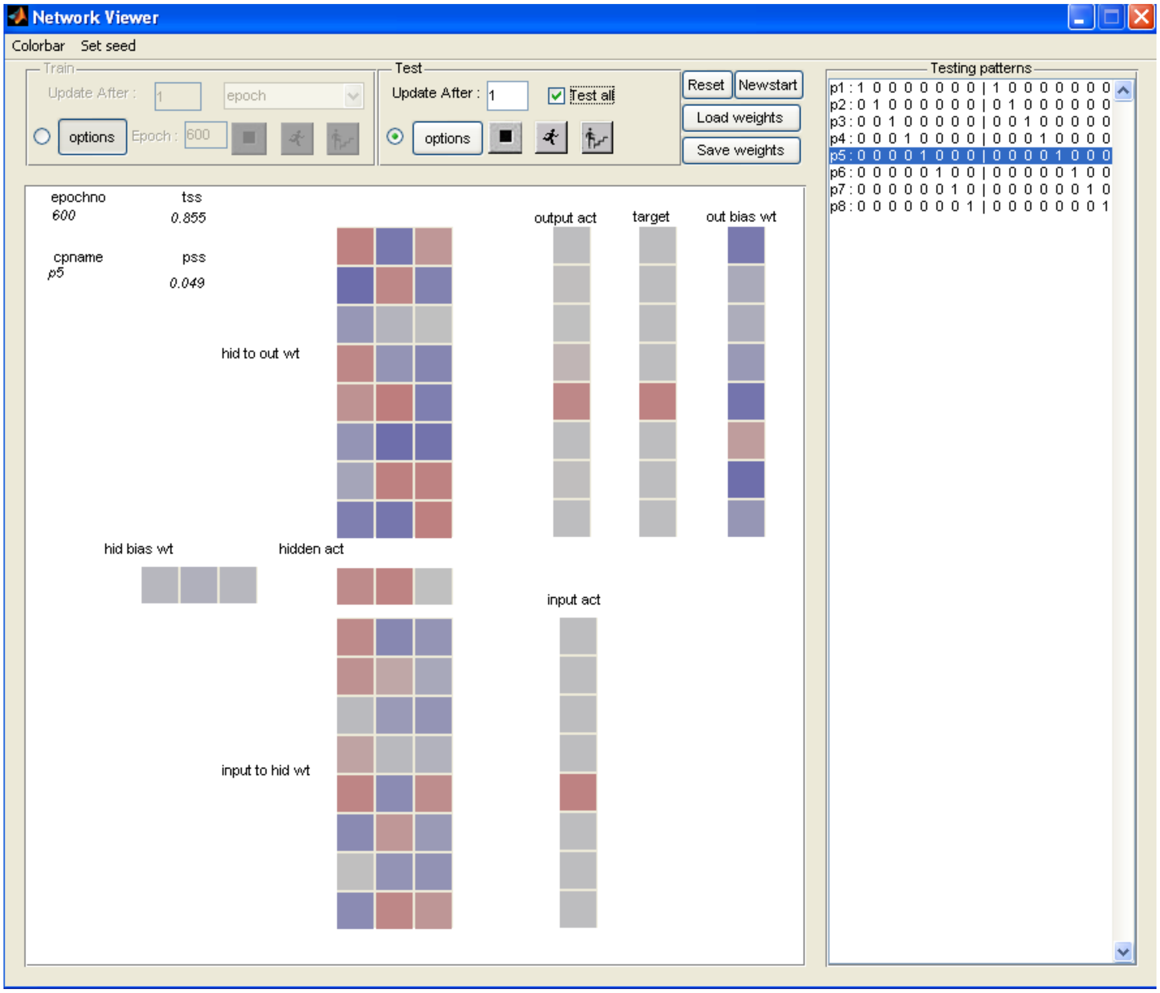
be logged, in order, in a .m file in your current directory. The file will be called "pdplog#.m", where the # will be the largest number in the directory (indicating the most recent log file created). Your file should look like Figure B.5. If your file does not look exactly like the figure, you can modify it to be exact (it's possible you have an extra command inserted on top of the file, for instance).

You should rename the file to be "bpencoder.m". Then you can initialize your network at any time by simply typing

```
bpencoder
```

in the Matlab command window, and the Network Viewer should pop-up.

That's it; the network is finished. Train the network and see how it uses the hidden layer to represent the eight possible input patterns.



Train: Update After : 1 epoch ○ Epoch : 600

Test: Update After : 1 ☑ Test all ●

epochno 600

tss 0.855

cpname p5

pss 0.049

Testing patterns

```
p1 : 1 0 0 0 0 0 0 0 | 1 0 0 0 0 0 0 0
p2 : 0 1 0 0 0 0 0 0 | 0 1 0 0 0 0 0 0
p3 : 0 0 1 0 0 0 0 0 | 0 0 1 0 0 0 0 0
p4 : 0 0 0 1 0 0 0 0 | 0 0 0 1 0 0 0 0
p5 : 0 0 0 0 1 0 0 0 | 0 0 0 0 1 0 0 0
p6 : 0 0 0 0 0 1 0 0 | 0 0 0 0 0 1 0 0
p7 : 0 0 0 0 0 0 1 0 | 0 0 0 0 0 0 1 0
p8 : 0 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 1
```

Figure B.6: This is the completed network, up and running. It has been trained for 600 epochs, and the network is being tested on pattern 5, where just the 5th input and output unit should be active. As you can see, the network output response is very good.

# Appendix C

# PDPTool User's Guide

This document is a brief introduction to PDPTool, a matlab application for providing hands-on experience with running connectionist or PDP models. The first section gives a quick overview of the software and its functionality. The second section provides a description of the applications menus, dialog boxes and display windows. This is a work in progress, so we will be updating the software and the documentation frequently as we make changes and upgrades.

Please download the document here: PDPTool.doc
Send software issues and bug reports to: pdplab-support@stanford.edu

# Appendix D

# PDPTool Standalone Executable

This section explains how to install your standalone copy of pdptool.This is a compiled binary of the application that can be run if MATLAB is not installed on your system.The procedures differ between Windows and Unix/Mac OSX.

If you encounter problems in installation, send an email to: pdplab-support@stanford.edu.

## D.1 Installing under Linux

Install a standalone copy of pdptool on Linux using the following steps.

1. Make a pdptool directory for yourself. Lets call it 'PDPStandAlone'.
   mkdir PDPStandAlone
   cd PDPStandAlone
2. Download the latest distribution for unix into your new directory.
3. Unzip the package with -
   unzip -x pdptool_linuxpkg.zip
4. rm pdptool_linuxpkg.zip
5. You will see the following files -
   (a) readme.txt - This describes essentially the same set of steps as this section. Additionally , it might contain instruction on updating the MCRInstaller on your computer.
   (b) pdptool - This is the pdptool executable
6. First read readme.txt. If this is your first time installation, you MUST download and run MCRInstaller.bin first. If you have done the installation previously, you are not required to run MCRInstaller again (i.e ignore step 7 and 8 ) UNLESS it is specified in the readme.txt file.

7. If the MCR is not installed or if it is an older version, download the MCRInsaller.bin file

   (a) After downloading ,type ./MCRInstaller.bin at the shell prompt to start the installation process.

   (b) This will bring up a dialog box and start InstallShield wizard for MATLAB Compiler Runtime.

   (c) Click on 'Next' button. You will be prompted for a directory name to install it in. You can create a new directory under your home directory or if you have root access, you can specfiy any other directory on the system.Let us call this directory MCRRootDir for future reference. Click the 'Next' button.

   (d) If the directory name is accepted, the install shield wizard will present a summary information. You can now click the 'Install' button to initiate installation. It may take several minutes to complete this process.

   (e) Click on 'Finish' to close the wizard.

8. Set up the environment :

   (a) If you are using bash shell - ( If you are not sure about your shell type echo $SHELL on the command prompt). Place the following commands in your .bash_profile file after replacing MCRRootDir with the name of the directory in which you installed MCR Compiler Runtime( see step 7(c)).

```
.bash_profile file will be directly under your home directory.
 Replace <version> with the MCR version intalled in step 7(c).
 The files in MCRRootDir are installed under version specific directory name e

 MCRROOT=MCRRootDir/<version>
 MCRJRE=${MCRROOT}/sys/java/jre/glnxa64/jre/lib/amd64
 LD_LIBRARY_PATH=${MCRROOT}/runtime/glnxa64:
 ${MCRROOT}/bin/glnxa64:
 ${MCRROOT/sys/os/glnxa64:
 ${MCRJRE/native_threads:
 ${MCRJRE}/server:
 ${MCRJRE}/client:
 ${MCRJRE}
 export XAPPLRESDIR ${MCRROOT}/X11/app-defaults
 export LD_LIBRARY_PATH ${LD_LIBRARY_PATH}
```

   (b) If you are using C shell, place the following commands in your .cshrc file after replacing MCRRootDir with the name of the directory in which you installed MCR Compiler Runtime ( see step 7(c)).

```
 .cshrc file will be directly under your home directory.
 Replace <version> with the MCR version intalled in step 7(c).
 The files in MCRRootDir are installed under version specific directory name e.g v714

 set MCRROOT=MCRRootDir/<version>
 set MCRJRE = ${MCRROOT}/sys/java/jre/glnxa64/jre/lib/amd64
 set LD_LIBRARY_PATH = ${MCRROOT}/runtime/glnxa64:
      ${MCRROOT}/bin/glnxa64:
      ${MCRROOT}/sys/os/glnxa64:
      ${MCRJRE}/native_threads:
      ${MCRJRE}/server:
      ${MCRJRE}/client:
      ${MCRJRE}
 setenv XAPPLRESDIR ${MCRROOT}/X11/app-defaults
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}
```

9. Add pdptool executable to your PATH environment variable.

   (a) If you are using bash shell - ( If you are not sure about your shell type echo $SHELL on the command prompt).Place the following command in your .bash_profile file. Replace pdptool_path with the full directory name created in Step 1.

   ```
    export PATH=${PATH}:pdptool_path

    The above change to .bash_profile file will be applied when you source the file using t

    source .bash_profile
   ```

   (b) If you are using C shell, place the following command in your .cshrc file

   ```
   setenv PATH=${PATH}:pdptool_path

   The above change to .cshrc file will be applied when you source the file using the comma

   source .cshrc
   ```

10. You are now ready to start using pdptool. For the most part, things will work as when running pdptool within MATLAB, so you can carry out exercises and create and run your own networks, as described in the main body of the Handbook. Information specific to running the standalone version under Linux is given in the next section.

## D.2 Running under Linux

1. If you have not already done so, Download the pdptool.zip file.

2. Extract the archived files into a new directory called 'pdptool' in your home directory or any other location on your linux machine using any archiving tool like unzip.

3. Open a shell prompt and cd to your pdptool directory (e.g. cd /home/Yourusername/pdptool).

4. To bring up an existing exercise, change to the subdirectory containing that exercise and then type pdptool exname.m at the shell prompt. For example, enter 'cd pa', then enter 'pdptool lin.m' to start the lin exercise in the pa directory. The exercise will load, and a subshell will be started with the comand prompt '>>pdp '. Alternatively, you can just type 'pdptool' at the shell prompt. The main pdp window will open and you can then proceed to create a network or load an existing network if you are in the directory containing the files relevant to this network.

5. The following features should facilitate your use of the software:

   (a) You can enter built-in MATLAB commands to the pdp command prompt. For example, entering '2/3' will return ans = .66667. Entering 'net' will return the top-level constituents of the loaded network, if any.

   (b) The up and down arrow keys allow you to move up and down through previous commands issued to the pdp prompt. Right and left arrows allow movement within a line so that you can edit a provious command.

   (c) You can use Ctrl+C and Ctrl+P keys to copy and paste text between the pdp prompt and outside the shell.

   (d) You can copy text from the pdp command Prompt window to the Clipboard by clicking and dragging the mouse to select the text to be copied; then press enter or right-click to execute the copy. You can then paste this information into an editor window to save to a file or edit. Such a file could become the basis of a set of commands that you could save in a script file (e.g., myscript.m) for execution when needed, using the command runscript myscript.m.

## D.3 Installing under Windows

Install a standalone copy of pdptool on windows using the following steps.

1. Create a new folder for yourself. Lets call it 'PDPStandAlone'.

2. Download the latest distribution for windows to your desktop.

3. Unzip the package with any windows archiving utility, extracting all files into your new folder. This will extract the following files -
   (a) readme.txt - This describes essentially the same set of steps as this section. Additionally , it might contain instruction on updating the MCR on your computer.
   (b) pdptool.exe - This is the pdptool executable.
4. Delete pdptool_winpkg.zip
5. Verify that MCR is installed on your system and is the version specified in the readme file. If it is, continue to next step. The usual location for MCR is C:\Program Files\MATLAB\MATLAB Compiler Runtime\<version >, where <version >indicates the version number, e.g v714. If the MCR is not installed or if it is an older version, download the MCRInsaller.exe file
   (a) After downloading double click on MCRInstaller.exe to start the installation process.
   (b) This will bring up a dialog box and start InstallShield wizard for MATLAB Compiler Runtime.
   (c) Follow the instructions, click on 'Next' button to initiate the installation process.
   (d) When the installation is complete, click on 'Finish' button to close the wizard.
6. Set up the environment : Note the full path to the PDPStandAlone folder where you extracted the executable to. (for example, the path might be 'C:\Users\YourUserName\Desktop\PDPStandAlone'). Add this to the 'Path' environment variable. You can do this easily by doing a right-click on 'My computer' , select properties, then select 'Advanced' tab. It will have an 'Environment Variables' button. Click on it to get a dialog box for setting envornment variables. Here, select 'Path' in the 'System Variables' , then select edit, it will open up a small editbox with the current path, enter a ';' (semi-colon), then add the path to the folder containing pdptool.exe. Then Click on 'Ok' and dismiss the properties window.
7. You are now ready to run pdptool. For the most part, things will work as when running pdptool within MATLAB, so you can carry out exercises and create and run your own networks, as described in the main body of the Handbook. Information specific to running the standalone version under Windows is given in the next section.

## D.4 Running under Windows

1. If you have not already done so, Download the pdptool.zip file.

2. Extract the archived files into a new folder called 'pdptool' to your Desktop or any other location on computer. (Some methods of extraction may create a directory called pdptool in the location you specify – others extract all files to that location; this depends on the extractor you use and the method of extraction).

3. Open a Windows MS/Dos Command Prompt window (Found under Start/All Programs/Applications/Command Prompt), and, within that window, cd to your pdptool directory (e.g., cd C:\YourUserName\Desktop\pdptool).

4. To bring up an existing exercise, change to the subdirectory containing that exercise and then type pdptool exname.m at the MS-DOS command prompt. For example, enter 'cd pa', then enter 'pdptool lin.m' to start the lin exercise in the pa directory. The exercise will load, and a subshell will be started within the MS-DOS Command Prompt window with the comand prompt '>>pdp'. Alternatively, you can just type 'pdptool' at the command prompt. The main pdp window will open and you can then proceed to create a network or load an existing network if you are in the directory containing the files relevant to this network.

5. The following features should facilitate your use of the software:

   (a) You can enter built-in matlab commands to the pdp command prompt. For example, entering '2/3' will return ans = .66667. Entering 'net' will return the top-level constituents of the loaded network, if any.

   (b) The up and down arrow keys allow you to move up and down through previous commands issued to the pdp command prompt or the pdp prompt (whichever is active). Right and left arrows allow movement within a line so that you can edit a provious command.

   (c) You can allow cut and paste operations into the command prompt window by activating a mode called 'QuickEdit Mode' at the command prompt. This mode can be set by right-clicking on the title bar of the command prompt window, then selecting 'Properties' then selecting 'QuickEdit Mode' under 'Options'. You can also select QuickEdit mode under 'Defaults' in the same way to have this option set automatically for future sessions.

   (d) You can now copy text from the pdp command Prompt window to the Clipboard by clicking and dragging the mouse to select the text to be copied; then press enter or right-click to execute the copy. You can then paste this information into an editor window to save to a file or edit. Such a file could become the basis of a set of commands that you could save in a script file (e.g., myscript.m) for execution when needed, using the command runscript myscript.m.

   (e) You can also paste text into the command window that has been saved to the clipboard (e.g., with Ctrl+C or the Copy menu command). Once the text you want is in the clipboard, simply Right

Click at the command prompt to enter the information. See what happens when you Copy the following text to the clipboard and then Right Click at the command prompt, and (if necessary) then hit enter:

```
2 +2; ans/3
```

## D.5 Installing under Mac OSX

Install a standalone copy of pdptool on a Mac OS X version 10.5 or above using the following steps.

1. Create a new folder for yourself. Lets call it 'PDPStandAlone'.

2. Download the latest distribution for mac osx to your desktop.

3. Unzip the package with any archiving utility, extracting all files into your new folder.

4. Delete pdptool_osxpkg.zip

5. You will see the following files -

   (a) pdptool - This is the pdptool executable

   (b) pdptool.app

   (c) run_pdptool.sh (shell script run to temporarily set environment variables and execute the application)

   (d) readme.txt - This describes essentially the same set of steps as this section. Additionally , it might contain instruction on updating the MCRInstaller on your computer.

6. First read readme.txt. If this is your first time installation, you MUST download and run MCRInstaller.dmg first. Verify that MCR is installed on your system and is the version specified in the readme file. If it is, continue to next step. The usual location for MCR is /Applications/MATLAB/MATLAB_Compiler_Runtime/<vers >, where <version >indicates the version number, e.g v714. If the MCR is not installed or if it is an older version, download the MCRInsaller.exe file

   (a) After downloading, double click on MCRInstaller.dmg or select Open menu item from the File menu on Finder window.

   (b) This will open a window with MCRInstaller.pkg file in it.

   (c) Double click on it to start the installation wizard.This will setup MCR on your system.

(d) Typically, it is installed in /Applications/MATLAB/MATLAB_Compiler_Runtime/<version >. Here, <version >refers to directory name that reflects the mcr version that is being installed. Currently it is v714. You can click on 'Choose Install Location' button to select a Destination folder of you choice for installation.

(e) Click the 'Install' button to begin the installation.

(f) It may take a few minutes to complete the process after which you will get a 'Successful installation' dialog box.

7. Set up the environment-

   (a) Open run_pdptool.sh file using any editor.

   (b) Edit line 15 to set MCRROOT variable to the destination folder you selected in Step 6(d). It is by default set to the destination folder /Applications/MATLAB/MATLAB_Compiler_Runtime/ <version >. Replace <version >with the MCR version intalled in step 7(c).The MCR files are installed under version specific directory name e.g v714.

   (c) Save and close run_pdptool.sh file

   (d) If you are using bash shell - ( If you are not sure about your shell type echo $SHELL on the command prompt). Place the following in your .bash_profile file. Replace pdptool_path with the full directory name created in Step 1 of this section.

   ```
   export PATH=${PATH}:pdptool_path
   ```

   (e) The above changes to .bash_profile file will be applied when you source the file using the command:

   ```
   source .bash_profile
   ```

   (f) If you are using C shell - ( If you are not sure about your shell type echo $SHELL on the command prompt). Place the following in your .cshrc/.tcshrc file. Replace pdptool_path with the full directory name created in Step 1 of this section.

   ```
   setenv PATH=${PATH}:pdptool_path
   ```

   (g) The above changes to .cshrc file will be applied when you source the file using the command:

   ```
   source .cshrc
   ```

8. You are now ready to start using pdptool. For the most part, things will work as when running pdptool within MATLAB, so you can carry out exercises and create and run your own networks, as described in the main body of the Handbook. Information specific to running the standalone version under OSX is given in the next section.

## D.6 Running under Mac OSX

1. If you have not already done so, Download the pdptool.zip file.
2. Extract the archived files into a new directory called 'pdptool' in your home directory or any other location on your linux machine using any archiving tool like unzip.
3. Open a terminal window and cd to your pdptool directory (e.g.cd /Users/Yourusername/pdptool).
4. To bring up an existing exercise, change to the subdirectory containing that exercise and then type pdptool exname.m at the terminal shell prompt. For example, enter 'cd pa', then enter 'pdptool lin.m' to start the lin exercise in the pa directory. The exercise will load, and a subshell will be started with the comand prompt '>>pdp '. Alternatively, you can just type 'pdptool' at the shell prompt. The main pdp window will open and you can then proceed to create a network or load an existing network if you are in the directory containing the files relevant to this network.
5. The following features should facilitate your use of the software:
   (a) You can enter built-in MATLAB commands to the pdp command prompt. For example, entering '2/3' will return ans = .66667. Entering 'net' will return the top-level constituents of the loaded network, if any.
   (b) The up and down arrow keys allow you to move up and down through previous commands issued to the pdp prompt. Right and left arrows allow movement within a line so that you can edit a provious command.
   (c) You can use Command+C and Command+P keys to copy and paste text between the pdp prompt and outside the shell.
   (d) You can copy text from the pdp command Prompt window to the Clipboard by clicking and dragging the mouse to select the text to be copied; then press enter or right-click to execute the copy. You can then paste this information into an editor window to save to a file or edit. Such a file could become the basis of a set of commands that you could save in a script file (e.g., myscript.m) for execution when needed, using the command runscript myscript.m.

# Bibliography

Anderson, J. A. (1977). Neural models with cognitive implications. In LaBerge, D. and Samuels, S. J., editors, *Basic processes in reading perception and comprehension*, pages 27–90. Erlbaum, Hillsdale, N.J.

Anderson, J. A. (1983). Cognitive and psychological computation with neural models. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:799–815.

Blake, A. (1983). The least disturbance principle and weak constraints. *Recognition Letters*, 1:393–399.

Cleeremans, A. and McClelland, J. L. (1991). Learning the structure of event sequences. *J Exp Psychol Gen*, 120:235–253.

Dilkina, K., McClelland, J. L., and Plaut, D. C. (2008). A single-system account of semantic and lexical deficits in five semantic dementia patients. *Cogn Neuropsychol*, 25:136–164.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Elman, J. L. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–224.

Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition*, 48:71–99.

Farah, M. J. and McClelland, J. L. (1991). A computational model of semantic memory impairment: modality specificity and emergent category specificity. *J Exp Psychol Gen*, 120:339–357.

Feldman, J. A. (1981). A connectionist model of visual memory. In Hinton, G. E. and Anderson, J. A., editors, *Parallel Models of Associative Memory*, chapter 2. Erlbaum, Hillsdale, NJ.

Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20:121–136.

Geman, S. and Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions of Pattern Analysis and Machine Intelligence*, PAMI-6:721–741.

Grossberg, S. (1976). Adaptive pattern classification and universal recoding: Part I: Parallel development and coding of neural feature detectors. *Biological Cybernetics*, 23:121–134.

Grossberg, S. (1978). A theory of visual coding, memory, and development. In Leeuwenberg, E. L. J. and Buffart, H. F. J. M., editors, *Formal Theories of Visual Perception*. John Wiley & Sons, New York.

Grossberg, S. (1980). How does the brain build a cognitive code? *Psychological Review*, 87:1–51.

Hebb, D. O. (1949). *The Organization of Behavior*. Wiley, New York.

Hertz, J. A., Palmer, R. G., and Krogh, A. (1991). *Introduction to the Theory of Neural Computation*. Westview Press.

Hinton, G. E. (1977). *Relaxation and Its Role in Vision*. PhD thesis, University of Edinburgh.

Hinton, G. E. and Anderson, J. A., editors (1981). *Parallel models of associative memory*. Erlbaum, Hillsdale, NJ.

Hinton, G. E. and Sejnowski, T. J. (1983). Optimal perceptual inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Washington, DC.

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

Hopfield, J. J. (1984). Neurons with graded response have collective computational properaties like those of two-state neurons. *Proceedings of the National Academy of Sciences, USA*, 81:3088–3092.

James, W. (1890/1950). *The Principles of Psychology*. Dover, New York.

Jenkins, W. M., Merzenich, M. M., Ochs, M. T., Allard, T., and Guíc-Robles, E. (1990). Functional reorganization of primary somatosensory cortex in adult owl monkeys after behaviorally controlled tactile stimulation. *Journal of Neurophysiology*, 63(1):82–104.

Kohonen, T. (1977). *Associative memory: A system theoretical approach*. Springer, New York.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69.

Lambon Ralph, M. A., McClelland, J. L., Patterson, K., Galton, C. J., and Hodges, J. R. (2001). No right to speak? The relationship between object naming and semantic impairment: neuropsychological evidence and a computational model. *J Cogn Neurosci*, 13:341–356.

Levin, J. A. (1976). Proteus: An activation framework for cognitive process models. Technical Report ISI/WP-2, University of Southern California, Information Sciences Institute, Marina del Rey, CA.

McClelland, J. L. (1981). Retrieving general and specific information from stored knowledge of specifics. In *Proceedings of the Third Annual Conference of the Cognitive Science Society*, pages 170–172, Berkeley, CA. [PDF].

McClelland, J. L. (1991). Stochastic interactive activation and the effect of context on perception. *Cognitive Psychology*, 23:1–44. [PDF].

McClelland, J. L. and Patterson, K. (2002). 'Words or Rules' cannot exploit the regularity in exceptions. *Trends in Cognitive Sciences*, 6:464–465. [PDF].

McClelland, J. L. and Rogers, T. T. (2003). The parallel distributed processing approach to semantic cognition. *Nature Reviews Neuroscience*, 4:310–322. [PDF].

McClelland, J. L. and Rumelhart, D. E. (1981). An interactive activation model of context effects in letter perception: Part 1. An account of basic findings. *Psychological Review*, 88:375–407. [PDF].

McClelland, J. L. and Rumelhart, D. E. (1988). *Explorations in parallel distributed processing: A handbook of models, programs, and exercises.* MIT Press, Boston, MA. [Archive].

McClelland, J. L., Rumelhart, D. E., and the PDP Research Group (1986). *Parallel distributed processing: Explorations in the microstructure of cognition. Volume 2: Psychological and biological models.* MIT Press, Cambridge, MA. [Book].

Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA.

Pinker, S. and Prince, A. (1988). On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition*, 28:73–193.

Pinker, S. and Ullman, M. T. (2002). The past and future of the past tense. *Trends in Cognitive Sciences*, 6:456–463. [PDF].

Plaut, D. C., McClelland, J. L., Seidenberg, M. S., and Patterson, K. (1996). Understanding normal and impaired word reading: computational principles in quasi-regular domains. *Psychol Rev*, 103:56–115.

Plaut, D. C. and Shallice, T. (1993). Deep dyslexia: A case study of connectionist neuropsychology. *Cognitive Neuropsychology*.

Rebur, A. S. (1976). Implicit learning of synthetic languages: The role of instuctional set. *Journal of Experimental Psychology: Human Learning and Memory*, 2:88–94.

Rogers, T. T., Lambon Ralph, M. A., Garrard, P., Bozeat, S., McClelland, J. L., Hodges, J. R., and Patterson, K. (2004). The structure and deterioration of semantic memory: A neuropsychological and computational investigation. *Psychological Review*, 111(205-235). [PDF].

Rogers, T. T. and McClelland, J. L. (2004). *Semantic Cognition: A Parallel Distributed Processing Approach*. MIT Press, Cambridge, MA.

Rohde, D. (1999). Lens: The light, efficient network simulator. Technical Report CMU-CS-99-164, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA.

Rohde, D. and Plaut, D. C. (1999). Language acquisition in the absence of explicit negative evidence: How important is starting small? *Cognition*, 72:67–109.

Rosenblatt, F. (1959). Two theorems of statistical separability in the perceptron. In *Mechanisation of Thought Processes: Proceedings of a Symposium Held at the National Physical Laboratory, November 1958, Volume 1*, pages 421–456, London. HM Stationery Office.

Rosenblatt, F. (1962). *Principles of neurodynamics*. Spartan, New York.

Rumelhart, D. E. and McClelland, J. L. (1982). An interactive activation model of context effects in letter perception: Part 2. The contextual enhancement effect and some tests and extensions of the model. *Psychological Review*, 89:60–94. [PDF].

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*. MIT Press, Cambridge, MA. [Book].

Rumelhart, D. E. and Todd, P. M. (1993). Learning and connectionist representations. In Meyer, D. E. and Kornblum, S., editors, *Attention and Performance XIV: Synergies in Experimental Psychology, Artificial Intelligence, and Cognitive Neuroscience*, pages 3–30. MIT Press, Cambridge, MA.

Rumelhart, D. E. and Zipser, D. (1985). Feature discovery by competitive learning. *Cognitive Science*, 9:75–112.

Servan-Schreiber, D., Cleeremans, A., and McClelland, J. L. (1991). Graded state machines: The representation of temporal contingencies in simple recurrent networks. *Machine Learning*, 7:161–193. [PDF].

Smolensky, P. (1983). Schema selection and stochastic inference in modular environments. In *Proceedings of the National Conference on Artificial Intelligence AAAI-83*, pages 109–113.

Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–34.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* MIT Press.

Tabor, W., Juliano, C., and Tanenhaus, M. K. (1997). Parsing in a dynamical system: An attractor-based account of the interaction of lexical and structural constraints in sentence processing. *Language and Cognitive Processes*, 12(2):211–271.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8(3-4):257–277.

Tesauro, G. (2002). Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199.

von der Malsburg, C. (1973). Self-organizing of orientation sensitive cells in the striate cortex. *Kybernetik*, 14:85–100.

Weisstein, N., Ozog, G., and Scoz, R. (1975). A comparison and elaboration of two models of metacontrast. *Psychological Review*, 82:325–343.

Widrow, G. and Hoff, M. E. (1960). Adaptive switching circuits. In *Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record, Part 4*, pages 96–104, New York. IRE.

Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Chauvin, Y. and Rumelhart, D. E., editors, *Back-propagation: Theory, Architectures and Applications.* Erlbaum.