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 & Rumelhart, 1981; Rumelhart & 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). (The interactive activation model of word perception is presented in Chapter 7.)

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).

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 (see 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

$$net_i = \sum_j w_{ij} output_j + extinput_i . \qquad (1)$$

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.

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 $max = 1$, $min \leqslant rest \leqslant 0$, and *decay* between 0 and 1. Note also that a_i is assumed to start, and to stay, within the interval [*min, max*].

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

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

$$a_i = \frac{(max)(net_i) + (rest)(decay)}{net_i + decay}. \tag{2}$$

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

$$a_i = \frac{net_i}{net_i + decay}. \tag{3}$$

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

$$a_i = \frac{net_i/decay}{(net_i/decay)+1}. \tag{4}$$

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:

$$a_i = \frac{(min)(net_i) - (decay)(rest)}{net_i - decay}. \tag{5}$$

Using $rest = 0$, this simplifies to

$$a_i = \frac{(min)(net_i)}{net_i - decay} \tag{6}$$

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

$$a_i = -\frac{(amin)(anet_i)}{anet_i + decay}. \tag{7}$$

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.

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 in competition, and imagine that both are receiving some excitatory input from outside but that the excitatory input to a (e_a) is stronger than the excitatory input to b (e_b). Let γ represent the strength of the inhibition each unit exerts on the other. Then the net input to a is

$$net_a = e_a - \gamma\,(output_b) \tag{8}$$

and the net input to b is

$$net_b = e_b - \gamma\,(output_a). \tag{9}$$

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

$$net_a = e_a - \gamma a_b \tag{10}$$

and

$$net_b = e_b - \gamma a_a. \tag{11}$$

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.

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 $2\times decay$. 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

$$\begin{aligned}\Delta a_a &= (1 - a_a)net_a - (decay)a_a \\ &= (1 - 0.5)(2)(decay)(0.5) - (decay)(0.5) \\ &= (0.5)(2)(decay)(0.5) - (decay)(0.5) \\ &= 0.\end{aligned}$$

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.

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 γ 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, Ozog, & Szoc, 1975) and play a prominent role in Grossberg's theory of learning in neural networks (see Grossberg, 1980).

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.

$$\Delta a = (max - a)e - (a - min)i - decay(a - rest). \tag{12}$$

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 tends always to excite the unit and net negative input tends always to 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

$$\Delta a = (1)(e) - (amin)(i) \tag{13}$$

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.

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.

Architecture

The IAC model consists of several units, divided into *pools*. In each pool, all the units are mutually inhibitory. Between pools, units may have excitatory connections. The model assumes that these connections are bidirectional, so that whenever there is an excitatory connection from unit *i* to unit *j*, there is also an excitatory connection from unit *j* back to unit *i*.

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.

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.

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.

IMPLEMENTATION

The IAC model is implemented by the **iac** program. This program, like all of our simulation programs, is written in C. The program consists of several parts: the command interpreter, the display package, the network configuration package, the patterns package, and the core routines of the model. In describing the implementation of this and other models, we will focus our attention on the core routines, but here we will briefly describe the rest of the package so that the reader has some pointers to understanding what is going on. More detailed implementation information is provided in Appendix F, which serves as a guide for readers who wish to actually explore and possibly alter the source code itself. Here follow brief descriptions of the various noncore parts of the program.

The Command Interpreter

The command interpreter is a set of subroutines that reads commands, either from a start-up file when the program is first called or from the keyboard while the program is running. There is also a facility that allows the user to direct the command interpreter to read and execute a sequence of commands found in a file. The command interpreter works by looking up commands it encounters in a large table of commands and executing the subroutine that is found in the table associated with the command. We will explain how to use the command interpreter in the section "Running the Program" later in this chapter.

The Display Package

The display package is a set of routines that manages the 24×80 character display screen. One set of routines is used to read a file called the *template* file when the program is first called. The information in this file is used to set up a set of *templates*, or display objects, and to indicate what each template contains and where on the screen it should be displayed. Another set of routines is used to do the actual displaying; these are commands that can be issued either by the user directly or from other parts of the program.

The Network Configuration Package

This package consists of a set of routines that is used in configuring the program for a particular application. The routines read commands from a file called the *network configuration* file, or the *network* file for short, and use these commands to set up arrays for the units and the weights, to specify initial values for the connections, and to indicate whether connections are modifiable or not.

The Patterns Package

This package consists of a set of routines that is used to read in a set of patterns for use as inputs to the model. These routines read a file called the *pattern* file. Some of the programs—among them, the **iac** program—can be run without reading in a file full of patterns, but the package is available for use if desired.

The Core Routines

Beyond the routines just mentioned is a set of *core* routines that implements the activation and competition processes described earlier. The routines are simple and make up a rather small part of the program. Here we explain the basic structure of the core routines used in the **iac** program.

getinput. This routine is used to specify which of the units in the network will receive external input. The routine prompts the user for names or numbers of units and for corresponding external input values, after first allowing the external inputs to be cleared to all zeros if desired. Note that this does not actually start the process of sending inputs to the units; it simply says which units should receive inputs and how strong they should be when the process actually starts.

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 other relevant variables are cleared, and the display is updated to show the initial state of 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, 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). At the end of *ncycles* of processing, the display is updated if it has not been updated on every cycle. The routine looks like this:

```
cycle() {

  for (cy = 0; cy < ncycles; cy++) {
    cycleno++;
    getnet();
    update();

/* what follows is concerned with
   pausing and updating the display */
    if (step_size == CYCLE) {
      update_display();
```

```
        if (single_step) {
          if (contin_test() == BREAK) break;
        }
      }
    }
    if (step_size > CYCLE) {
      update_display();
    }
}
```

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 to each unit. 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 unit, 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.

Whether a connection is excitatory or inhibitory is determined by its sign. Thus if w_{ij} is positive, $w_{ij}a_j$ is added into the excitation term of unit i. If w_{ij} is negative, $w_{ij}a_j$ is added into the inhibition term of unit i. These operations are only performed if the activation of the sending unit is greater than 0. The code that implements these calculations is as follows:

```
getnet() {

  for (i = 0; i < nunits; i++) {
    excitation[i] = inhibition[i] = 0;

    for (j = 0; j < nunits; j++) {
      if (activation[j] > 0) {
        if (w[i][j] > 0) {
          excitation[i] += weight[i][j]*activation[j];
        }
        else if (w[i][j] < 0) {
          inhibition[i] += weight[i][j]*activation[j];
        }
      }
    }
    netinput[i] = estr*extinput[i] + alpha*excitation[i]
                      + gamma*inhibition[i];
  }
}
```

Standard update. The *update* routine increments the activation of each unit, based on the net input and the existing activation value. Here is what it looks like:

```
update() {

  for (i = 0; i < nunits; i++) {
    if (netinput[i] > 0) {
      activation[i] += (max - activation[i])*netinput[i]
                        - decay*(activation[i] - rest);
    }
    else {
      activation[i] += (activation[i]-min)*netinput[i]
                        - decay*(activation[i] - rest);
    }
    if (activation[i] > max) activation[i] = max;
    if (activation[i] < min) activation[i] = min;
  }
}
```

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 to each unit 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:

```
excitation[i]*= alpha*excitation[i];
inhibition[i]*= gamma*inhibition[i];
if (extinput[i] > 0) excitation[i] += estr*extinput[i];
else if (extinput[i] < 0)
  inhibition[i] += estr*extinput[i];
```

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

```
update() {

  for (i = 0; i < nunits; i++) {
    activation[i] += (max - activation[i])*excitation[i]
                  + (activation[i] - min)*inhibition[i]
                    - decay*(activation[i] - rest);
```

```
        if (activation[i] > max) activation[i] = max;
        if (activation[i] < min) activation[i] = min;
    }
}
```

The reader may have noticed that the main computational loops of the program make no explicit mention of the IAC network architecture, in which the units are organized into competitive (inhibitory) pools 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. Actually, some complexity has been suppressed, but not much. What makes the programs rather complex is all of the auxiliary routines.

RUNNING THE PROGRAM

Starting Up

To run the **iac** program, it is first necessary to set up a working directory containing the relevant files. An explanation of how this is done is given in Appendix A. Here we assume that you have created a working directory for **iac** and that you have positioned yourself in that directory. To execute the program, you would enter the following:

iac <templatefile> <startupfile>

Note that any commands entered either inside or outside of our program must be terminated by pressing the *return* or *enter* key. We adopt the convention of giving variables that must be replaced by specific values inside of angle brackets. Thus *<templatefile>* must be replaced by the name of a specific template file, and *<startupfile>* must be replaced by the name of a specific start-up file. By convention, the names of template files end with the extension *.tem* and the names of start-up files end with the extension *.str*. Henceforth we will refer to the template file as the *.tem* file, and the start-up file as the *.str* file.

The program will run without a template file or a start-up file being given, but the template file is necessary to tell the program what to display and where to display it; without one, there will be no display on the screen. The first argument to the program is always interpreted as a template file name, so the program will misinterpret the *.str* file if the *.tem* file is left out. To prevent this, the program may be run with a single "–" in place of the template file name:

iac – <startupfile>

The *.str* file can be omitted without any ill effects. In general this file contains commands that initialize the network configuration and set the values of various parameters of the model. These can all be entered directly by the user once the program has started to run. The two things that are special about the *.str* file is that the commands in it are executed without printing anything to the screen and that errors encountered in the *.str* file cause the program to terminate immediately, with an error message printed to the screen. The *.str* file can contain any commands the user wishes to put in it, including commands to run the program, save output, and quit. This allows programs to be run in background mode on UNIX systems, using a script of commands from the *.str* file.

Assuming the *.str* file is processed without error and without encountering a *quit* command, the program will present a display containing an **iac**: prompt on line 0, a menu listing commands that may be entered on lines 1 through 4, and a display of the current state of the network. From this point on, the user may enter commands to the program via the command interface.

Entering Commands via the Command Interface

It is useful to think of commands as being entered one per line, with spaces separating the command from its various arguments. For example, the **iac** program provides a command that allows the user to display any of the various display chunks or *templates* that have been specified in the *.tem* file. This command is given by entering

iac: *disp <template>*

where *<template>* is the name of any template specified in the *.tem* file. Note that in this and subsequent examples, we display the prompt typed by the computer in **bold**, with the response from the user in *italic*. Also note that the user interface is case sensitive, and command names are in lowercase throughout. In the exercises we capitalize the first letter of some of the unit names; otherwise everything is lowercase.

A nice feature of the command interface is that it will generally prompt you with possible options should you wish to see them. At the top level, the program always provides a list of the commands that can be entered.

To see lists of options that are specifiable within a given command, enter the command name by itself. Thus if you enter

```
iac: disp
```

the program comes back with the list of possible continuations of the display command and a revised prompt,

```
iac: disp /
```

indicating that you may now enter the continuation you want. If you just want to look at the list of possible continuations, you can type *return* (press the return or enter key), and the program will return to the top level. Alternatively, you may enter one of the available continuations. If further input is required, you will be prompted for it; you may type *return* at almost any time, and the program will revert to the top-level prompt, awaiting a new command input.

One may think of the commands available in the program as consisting of a command name, followed by one or more *specifiers*, followed finally by one or more *arguments*. The specifiers indicate which particular one of several specific commands you wish to execute, and the arguments are the parameters of the command itself. Commands or specifiers that must be followed by further specifiers are terminated in the menus by a "/" character. The other commands or specifiers do not require further specifiers, though the user will generally be prompted for additional arguments. The *display* command is an example of the former type of command: It requires a further specifier indicating which template to display. The *log* command is an example of the latter type. It is used to control the storing of a log of the activity of the network in a file. This command requires a file name argument.

In all cases, the entire command, including the command name, the specifiers, and the arguments, may be entered as a single line. Alternatively, the user may enter any part of a command and be prompted for the possible continuations of it.

A further feature of the command interface is that commands and command specifiers do not have to be entered in their entirety; instead, it is only necessary to type enough of the beginning of the command or specifier to uniquely distinguish it from all other available alternatives. Thus,

```
iac: lo foo.log
```

is sufficient to open a log file called *foo.log*, given that there are no other commands accessible at the top level that begin with the characters *lo*.

The command interpreter prints an error message if the command string entered is not consistent with any of the available commands. The message is available for a short period (a few seconds), then the command interpreter returns to the top level, ignoring the remainder of a command line on which an error is encountered.

After start-up, it remains possible to ask the command interpreter to process a preset list of commands from a file. This is done using the *do* command. This command requires a file-name argument; once it opens the file, the command interpreter simply reads commands as if they had been typed in by the user. When it encounters an error, it prints the corresponding error message and presents the following *interrupt* prompt:

iac: p to push /b to break /<cr> to continue:

If you enter *p*, the command interpreter will be called recursively, and you can enter any commands you wish (see below). If you enter *b*, the command interpreter will quit reading commands from the command file and return to the top level for further input. The notation *<cr>* stands for carriage return; if you just type a *return*, the command interpreter will continue trying to read commands from the command file, with possibly anomalous results. In general, it is best to break at this point and to quit the program and fix the *do* file.

Interrupting Processing

Sometimes when the program is running, you may see that it is doing something you had not intended or that it is continuing longer than you wanted. In this case, you can type *control-C* (hold down the key marked *control* and the *C* key at the same time). This causes the program to set a flag that is checked at the end of the current processing cycle. When the flag is found to be set, the program prints the interrupt prompt, giving the user the option to push, break, or continue. Again, you can continue processing by typing *return*; you can break and return to command level by entering *b*; or you can call the command interpreter recursively by entering *p*.

When you interrupt processing, the interrupt character (^C) will appear at a random place on the screen. If this is annoying, you may enter *display state* to the command interpreter to clear the screen and redisplay the state of the network.

Single Stepping

In order to examine the course of processing as it unfolds, the program offers the user the option of using the *single-step* mode. When this mode is on, the program interrupts itself after each processing step, displaying the interrupt prompt, just as if the user had typed an interrupt. Generally, the user will type *return* when ready to continue, but it remains possible to either break or push to a recursive command level.

The Recursive Command Level

When the command interpreter is called recursively, it displays the following prompt:

[N] iac:

Here [*N*] indicates the depth of the recursion (it is possible to embed recursive calls indefinitely, or at least until your computer runs out of stack space). In this mode you can enter commands as you normally would. To terminate this recursive call to the command interpreter, simply type *return*; this will return you to the interrupt prompt, from which the push, break, and continue options remain available.

Running Commands Outside the Program

Sometimes, while a simulation is in progress, it is useful to execute a command outside the simulation program. For example, you may wish to determine whether a particular file exists. To do this you can use the *run* command. Simply enter *run*, followed by the command, followed by any arguments to the command, followed by *end*. For example, on a PC, to see if the file *foo.log* exists, you would enter:

run dir foo.log end

The program will then pass the command

dir foo.log

to the MS-DOS command interpreter.

As another example, if you enter

run command end

the MS-DOS command interpreter will be invoked, and you can enter MS-DOS commands. To return to the simulation program at this point, just enter *exit*.

Quitting the Program

To quit the program, you enter *quit* to the command interpreter. Since you may have done various things that you would like to save before you quit, the program asks you to confirm your intention of quitting. If you enter *y* at this point, the program will terminate, closing all files that were open and clearing the screen. If you do not want to quit, type *return* or anything else.

Display Conventions

Displays are organized to pack lots of information into a small space. As a result, values of floating-point variables are usually given in a compact format in which only two or three character positions are used for each variable. The true value of the variable is first multiplied by a scale factor (specified in the template file) and then printed in the available space. For activations and other variables with absolute values less than 1.0, a scale factor of 100 is generally used, so that an activation of 0.01 displays as " 1" and 0.99 displays as " 99." Values that are negative are displayed, by default, in *standout* mode, which on most display devices is reverse video (lighted background). If you are running the programs on a display device that has no standout mode, you should turn off the use of standout by using the *display options standout* command described in the command description section. When standout mode is off (and when the screen image is saved in a file), negative numbers are displayed using minus signs where possible. Thus, if three character positions are available, −0.99 displays as "−99."

Special provisions have been made to deal with the problem of displaying variables that cannot fit the space available to them. Values that exceed the available space are printed as one or more "*" characters. When standout mode is off and the minus sign will not fit, the number is printed in an alphabetic code, where *a* to *j* stand for the digits 1 to 9 and *o* stands for 0; values that are still too large to fit are displayed as one or more *X*'s.

Making Graphs

Utility programs are provided for making simple graphs of output that has been stored in log files. The use of these utility programs is described in Appendix D. You may also wish to use other graphics software of your own, since the plotting program we provide has very coarse spatial resolution.

Command Descriptions

Here we describe all of the commands available in the **iac** program. We suggest you take a brief look at these descriptions of the commands and at the following discussion of the variables used in the program, to get a sense of how things are organized, and then proceed to the first exercise, where you will have a chance to learn about the commands by using them. (A list of all of the commands available in all programs is given in Appendix B.)

Note that we define a command as a sequence consisting of a command verb followed by one or more command specifiers. First the simple top-level commands are described: These are the commands that consist only of a command verb followed by arguments. Then the more general command verbs and the specific commands available using each verb are described in a nested fashion. We first describe the command verb and then give the specifiers for that verb; if there are subspecifiers, they are further nested under the specifiers.

The top-level commands in the program allow the user to run the simulation model and to carry out a variety of other actions. Most of these commands are common to all of the programs presented in this book.

clear

Clears the screen, leaving it blank.

cycle

Runs the program through *ncycles* processing cycles.

do

Prompts for a file name containing a list of commands to execute and then prompts for a count, which indicates the number of times to execute the entire list.

input

Prompts for unit names or numbers and then prompts for external input values. These may be any real number, though the usual values are $+1$ and -1. Entered values are placed in the external input vector for use during subsequent processing.

log

Prompts for a file name to store a log of the information displayed to the screen. If the file the user specifies already exists, the new log will be appended to the end of this file. Once a log file has been opened, each time the screen is updated, all the templates that are displayed are checked to see if they should be logged. All the information logged at one time is placed on a single line in the log file, separated by spaces. Items whose *dlevel* is less than or equal to the global *slevel* parameter are logged in the order they occur in the *.tem* file; all other items are skipped over. See Appendix D for a more detailed explanation. Logging can be turned off by calling *log* again and entering "–" instead of a file name; alternatively, a new file name will close the old log file and open a new one.

quit

Quits the program. Prompts for confirmation; to confirm, enter *y*.

reset

Resets the model to its initial state. The *cycleno* is set to 0, and the activations of all the units are set back to the resting level. The display is cleared and updated.

run

Passes a command out of the program for execution by the MS-DOS command interpreter. Prompts for a command, then for arguments. End the list of arguments by entering *end* or by simply typing an extra *return*.

test

Prompts for the name or number of a pattern to test the model with. If the pattern is successfully identified, the program uses it to set the values of the first *ninputs* elements of the external input vector; all additional units in the net are given an external input of 0. This command then resets the network and runs the program through *ncycles* of processing. The user may then run more cycles by using the *cycle* command.

This ends the simple commands. We now turn to the compound commands that require a general command followed by further specifiers. As in the program itself, we designate command words that require further specifiers with "/"; the user is not required to type these, however. We describe each general command in turn, giving each with a list of the further specifiers available.

disp/

Allows the user to display the current state of the network, to designate a specific template for display, or to set various display options, using the *opt* subcommand.

disp/ state

Clears the screen and displays the current state of the network, as specified by the global *dlevel* variable and by the attributes of the various templates found in the *.tem* file. All templates whose associated *dlevels* are less than or equal to the global *dlevel* are displayed.

disp/ <template>

Displays the designated template, regardless of its *dlevel*. All other template attributes are honored.

disp/ opt/

Allows the user to modify various display options using the subspecifiers that follow.

disp/ opt/ standout

Determines whether negative numbers are displayed to the screen in *standout* mode (reverse video, on most displays) or not. If standout mode is not available, you should set this option to 0;

otherwise, the program will attempt to use standout mode with negative numbers, and they will be indistinguishable from positive numbers.

disp/ opt/ <template>

Allows the user to change various options associated with the templates defined in the *.tem* file. After the user enters

disp opt <template>

the program prompts for an attribute of the template to change—either the display level, the number of digits of precision, or the scale factor associated with the template. The user then enters the name of the attribute (just the first character will do) followed by the new value to assign to this attribute.

exam/

Allows the user to examine and optionally set the value of one of the variables of the program. Use of this command is described under "Accessing Variables," later in this chapter. (This command is a synonym of *set/*.)

get/

Allows the user to get lists of things into the program, according to the option specified. The *get* commands either read a file or request a list of items terminated by *end* or a blank line.

get/ network

Allows the reading of a network specification from a file. The program prompts for the file name. The *get/ network* command is usually given in the *.str* file. A full description of the format of the network file is given in Appendix C.

get/ patterns

Allows the user to read in a set of input patterns from a file. Before it can be used, the variable *ninputs* must be defined. This variable tells the program how many input activation values to expect to find in each input pattern. The command prompts for a file name. Conventionally such files are given a *.pat* extension. A *.pat* file consists of a sequence of pattern specifications, typically one per line. The first entry in each pattern specification is a name for the pattern. Subsequent elements specify values to be used to specify inputs to each of the units in the network. Elements are separated by spaces. The first element determines the input to the first unit, the second element determines the input to the second unit, and so on. Elements may be floating-point numbers, or they may be a "+", "–", or "." character. A "+" indicates that the input to the corresponding unit is +1, a "–" indicates that the input is −1, and a "." indicates that the corresponding unit will receive a 0 input when the pattern is presented to the network. The number of patterns read is stored in the variable *npatterns*. The patterns are stored in an array called *ipattern*; each pattern is a separate row of this array. Thus, element 4 of pattern 2 is stored in *ipattern*[2][4].

get/ unames

Allows the user to enter a list of names for the units in the model. Prompts for one name at a time; these are assigned sequentially to the units, starting with unit 0. The end of the list is indicated by typing *return* (a blank line) or entering the string *end*.

get/ weights

Allows the reading of a file containing numerical weights, one for each connection in the network. Prompts for the name of a weight file. Weight files are expected to be in the same format as is used by the *save/ weights* command (see below).

save/

Allows the user to save various kinds of information in files. There are two specifiers available in **iac**: *screen* and *weights*.

save/ screen

Prompts for a file to store an image of the screen. If the file already exists, the screen image is appended to it. Standout mode is turned off since it cannot be "printed" in a file. The screen is displayed with standout mode off before the file name is requested; if you do not want to store what you see, simply type *return* instead of a file name, and no saving will take place.

save/ weights

Prompts for a file to store the current values of the weights. If the file already exists, you are warned and asked if you wish to overwrite it. A file so created can later be read in again using the *get/ weights* command. Entries in the file are ordered as follows: For each unit, all the weights to that unit are stored in increasing order. After all the weights come the biases for each unit, if there are any biases. (There are no biases in **iac**.)

set/

Allows the user to examine and optionally set the value of one of the variables of the program. Use of this command is described under "Accessing Variables," later in this chapter. (This command is a synonym of *exam/*.)

Variable Types

Variables are of several types: Some are strings of characters, some are integers, and some are floating-point numbers. Within each of these types, some are single-valued variables; others are vectors, or lists of variables with an index; and others are matrices, or two-dimensional arrays of variables with two indexes. In accessing single-valued variables, once the variable has been entered, the program will display the current value, and at this point the user may enter a new value or simply type *return* to return to the command level. With vector variables, an index must be given before a value is displayed for possible alteration; with array variables, two indexes are required.

Vector indexes may be given as numbers (with the first element being element 0) or, if the elements of a vector have names, these may be used as indexes. Thus, if unit 0 has been named *Fred*, the index 0 may be specified by entering *Fred*. Note that both units and patterns may have names. The program is set up so that unit names and pattern names are consulted; the program can get confused if the same names are used for both units and patterns.

Arrays are generally used for weights or other variables that are associated with the connections to a particular unit from another unit. The indexes are specified in receiver-sender order. Note that if the units have names, the names can be used here as well. Arrays are also used for patterns, with the first index specifying the pattern number and the second specifying the number of the element within the pattern.

All of the programs have a fairly large number of variables. These variables are organized into several different functional groups:

- *Mode*. These are variables whose values determine switchable characteristics of the model being implemented by the program. Generally, the programs can be thought of as implementing a "base" model and several variants. When the *mode* variables are all set to 0 (off), the base model is implemented. When one or more of these variables is set to 1 (on), other variants are in force.

- *Configuration*. These variables determine basic configurational properties of the network that is being used in a particular simulation run, and they are generally set in the *.net* file. Configuration variables should generally not be changed during a run but can be examined. Also included as a configuration variable is the list of names that have been assigned to the units in the network.

- *Environment*. These are variables associated with the test environment in which the program is run, that is, with the set of patterns that may be presented to the model for testing.

- *Parameter*. These variables are the parameters of the model, the ones that determine such things as the relative strength of excitation vs. inhibition, the decay rate, and so on.

- *State*. These are variables that are associated with the current state of the processing network, such as the activation values of the units.

- *Top-level variables.* These variables are the ones you need to change to control the activity of the simulation model itself. Also included at the top level are the weights associated with the connections and the bias terms (if any) associated with the units in the network.

Accessing Variables

Variables are accessed using the *set* and *exam* commands. The top-level variables are accessible directly, because they tend to be accessed most often in using the programs. These variables are accessed by typing

set <variable>

or

exam <variable>

Other variables are accessed through specifiers that correspond to the different variable types; the specifiers are *config, mode, env, param*, and *state*. They are accessed by typing

set <specifier> <variable>

or

exam <specifier> <variable>

Variable List

Here follows a list of all of the variables available in **iac**. First listed are all of variables directly accessible via the *set* and *exam* commands. These are followed by the variables that require specifiers, as indicated.

dlevel
: An integer variable that determines which templates will be displayed when the display is updated. All templates with *dlevel*s as specified in the *.tem* file that are less than or equal to this global *dlevel* parameter are updated.

ncycles
: An integer variable that specifies how many processing cycles are executed when the *cycle* command is entered.

seed
: The current value of the seed used by the random number generator. May be set by the user to equal any integer. Not used in **iac**; see Chapter 3 for a full discussion.

single
: A switch variable, normally set to 0, that makes the program run in single-step mode when set to 1.

slevel
: An integer variable that determines which templates will be logged in the log file when the screen is updated. Note that a template is logged only if (a) a log file has been opened with the *log* command, (b) the template's *dlevel* is less than or equal to the global *dlevel*, and (c) the template's *dlevel* is also less than the global *slevel*.

stepsize
: A string variable that controls the size of the processing steps taken by the program between screen updates. Allowed values are *cycle* and *ncycles*.

weight
: A floating-point matrix variable. The matrix contains the weights to each unit in the network from each unit. The user can examine the value of a particular weight by entering the command *set/ weight <to_index> <from_index>*, where *<to_index>* is the name or number of the receiving unit and *<from_index>* is the name or number of the sending unit.

config/ ninputs
: An integer variable specifying the number of input units for which external inputs will be specified in each *ipattern* read by the *get/ patterns* command.

config/ nunits
: An integer variable specifying the number of units in the network. This is generally declared in the *.net* file and should not be reset but may be examined.

config/ uname
: A vector of character strings read in by the *get/ unames* command. These strings are taken to be the names associated with the units in the network.

env/ ipattern
: A floating-point array variable containing the patterns read in by the last *get/ patterns* command. Note that the first index specifies the *pattern*, and the second index specifies the *element* within the pattern. Thus the command *exam/ ipattern 3 2* requests the program to print the value of element 2 in pattern 3. Note, the term *ipattern* is used rather than just *pattern* because in some programs there are two types of patterns, input patterns and target patterns. The *ipattern* variable is used to refer to the former type.

env/ maxpatterns
: The maximum number of patterns that can be read into the network. This variable is automatically increased by the program if more patterns are encountered in the *.pat* file, but when large numbers of patterns are used, network initialization occurs more quickly if *maxpatterns* is set to a value a bit greater than the number of patterns that will be read in.

env/ npatterns
: The number of patterns that have been read into the program by the last *get/ patterns* command.

env/ pname
: A vector or list of character string variables specifying the names of the patterns read in by the *get/ patterns* command.

mode/ gb
: This mode variable, when set to 1, causes the program to use Grossberg's updating function rather than the standard function taken from McClelland and Rumelhart (1981).

param/ alpha
: A floating-point variable that scales the strength of the excitatory influences on units from other units.

param/ decay
: A floating-point variable specifying the decay rate of unit's activations.

param/ estr
: A floating-point variable that scales the strength of the external input to units in the network.

param/ gamma
: A floating-point variable that scales the strength of the inhibitory influences on units from other units.

param/ max
: A floating-point variable specifying the maximum activation of each unit.

param/ min
: A floating-point variable specifying the minimum activation of each unit.

param/ rest
: A floating-point variable specifying the resting activation of each unit.

state/ activation
: A vector of floating-point variables specifying the activation values of the units in the network.

state/ cpname
: A string variable specifying the name of the current pattern (if any) that was specified for testing via the *test* command.

state/ cycleno
: The number of processing cycles elapsed since the last reset.

state/ excitation

A vector of floating-point values specifying the excitatory input to each unit, as computed during the most recent processing cycle.

state/ extinput

A vector of floating-point values specifying the external input to each unit, as determined by the last *input* command or by the last pattern specified for testing via the *test* command.

state/ inhibition

A vector of floating-point values specifying the inhibitory input to each unit, as computed during the most recent processing cycle.

state/ netinput

A vector of floating-point values specifying the net input to each unit, as computed during the most recent processing cycle.

state/ patno

The index of the current pattern (if any) that was specified for testing via the *test* command.

OVERVIEW OF 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. Answers to the questions in Exs. 2.1-2.3 are given in Appendix E.

Ex. 2.1. 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.

- Retrieval with noisy cues.
- 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 1. You are to use the **iac** program in conjunction with this data base to run illustrative simulations of these basic properties of memory. In so doing, you will observe behaviors of the network that you will have to explain using the analysis of IAC networks presented earlier in the "Background" section.

The Jets and The Sharks

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

FIGURE 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 J. L. McClelland, 1981, *Proceedings of the Third Annual Conference of the Cognitive Science Society*. Copyright 1981 by J. L. McClelland. Reprinted by permission.)

Starting up. Before running this exercise, you must first set up an *iac* directory as described in Appendix A. Change your working directory to this directory, and enter

```
iac jets.tem jets.str
```

This causes the program to begin running using the template information stored in the *jets.tem* file, with the start-up commands contained in the *jets.str* file. The latter file contains the command

```
get net jets.net
```

This command causes the program to set up a network containing 68 units. 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. The pool of *instance* units 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 (Figure 11 from *PDP:1*). Note that the figure is incomplete, in that only some of the name and instance units are shown. The *jets.str* file provides names for each of the units, using the *get/ names* command. These are given only for the convenience of the user, of course; all actual computation in the network occurs only by way of the connections.

The *jets.str* file also sets the values of the parameters of the model. These values are

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

These are all set using the command form:

```
set param <name> <value>
```

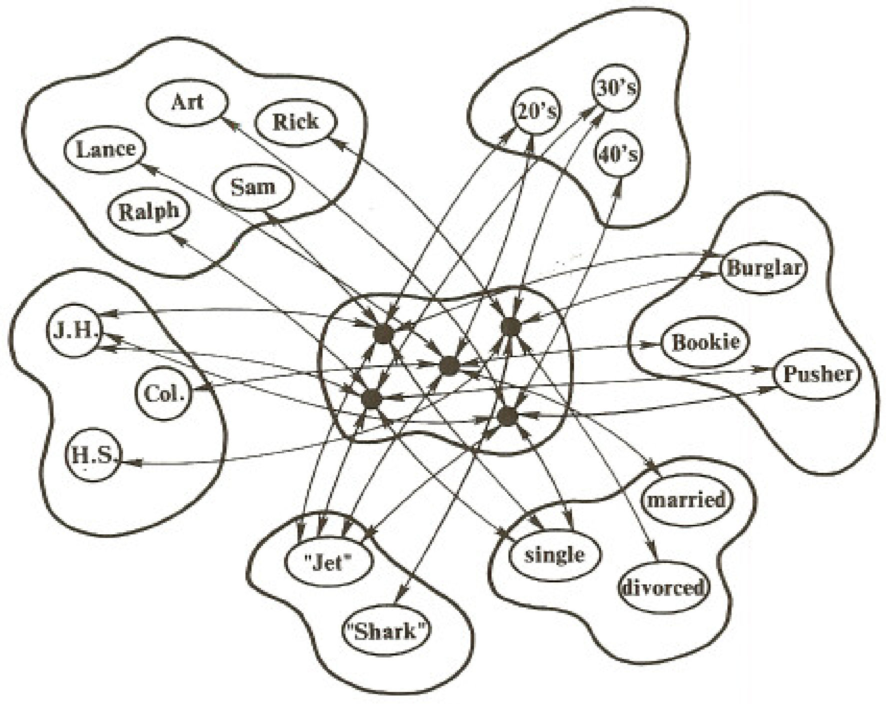
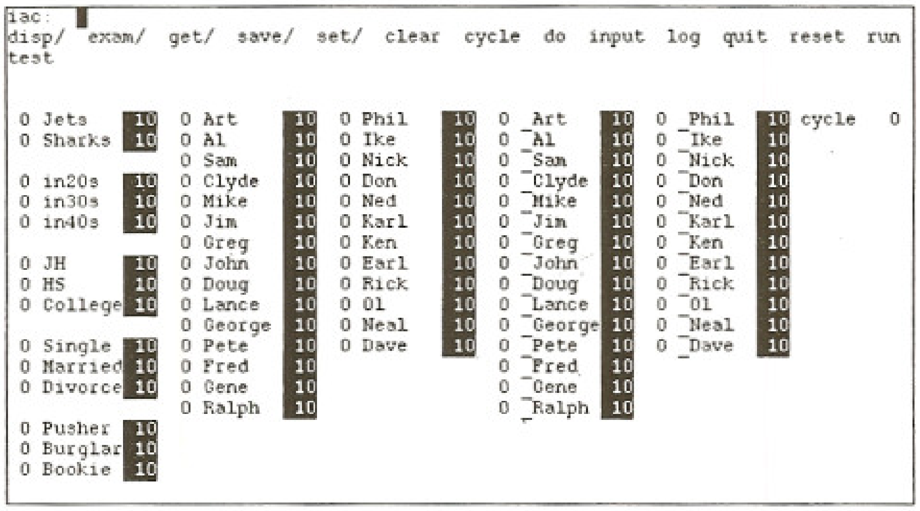


FIGURE 2. The units and connections for some of the individuals in Figure 1. (Two slight errors in the connections depicted in the original of this figure have been corrected in this version.) (From "Retrieving General and Specific Knowledge From Stored Knowledge of Specifics" by J. L. McClelland, 1981, *Proceedings of the Third Annual Conference of the Cognitive Science Society*. Copyright 1981 by J. L. McClelland. Reprinted by permission.)

After the program has read the information contained in the *jets.tem* and *jets.str* files, it produces the display shown in Figure 3. The display shows the names of all of the units, each flanked with a number on the left and a number on the right. The number on the left indicates the external input to the unit; since no input has been specified, these numbers are now all 0. The number on the right indicates the unit's current activation value, which at this point is equal to its resting activation. These numbers are to be read as *hundredths*. Thus this initial display indicates that all of the units have a resting activation level of -0.10.

Units are organized into columns, with the property units in the first column, the name units for the Jets in the second column, the name units for the Sharks in the third column, and the instance units for the Jets and the Sharks in the fourth and fifth columns, respectively. Note that the names of the instance units begin with "_"; the name unit for Lance is called *Lance* but the instance unit for Lance is called *_Lance*. On the far right of the display is the current cycle number, which is initialized to 0.



```
iac:
disp/  exam/  get/  save/  set/  clear  cycle  do  input  log  quit  reset  run
test


 0 Jets     10  0 Art     10  0 Phil    10  0 _Art    10  0 _Phil   10 cycle    0
 0 Sharks   10  0 Al      10  0 Ike     10  0 _Al     10  0 _Ike    10
                0 Sam     10  0 Nick    10  0 _Sam    10  0 _Nick   10
 0 in20s    10  0 Clyde   10  0 Don     10  0 _Clyde  10  0 _Don    10
 0 in30s    10  0 Mike    10  0 Ned     10  0 _Mike   10  0 _Ned    10
 0 in40s    10  0 Jim     10  0 Karl    10  0 _Jim    10  0 _Karl   10
                0 Greg    10  0 Ken     10  0 _Greg   10  0 _Ken    10
 0 JH       10  0 John    10  0 Earl    10  0 _John   10  0 _Earl   10
 0 HS       10  0 Doug    10  0 Rick    10  0 _Doug   10  0 _Rick   10
 0 College  10  0 Lance   10  0 Ol      10  0 _Lance  10  0 _Ol     10
                0 George  10  0 Neal    10  0 _George 10  0 _Neal   10
 0 Single   10  0 Pete    10  0 Dave    10  0 _Pete   10  0 _Dave   10
 0 Married  10  0 Fred    10                0 _Fred   10
 0 Divorce  10  0 Gene    10                0 _Gene   10
                0 Ralph   10                0 _Ralph  10
 0 Pusher   10
 0 Burglar  10
 0 Bookie   10
```

FIGURE 3. The initial display produced by the **iac** program for Exercise 1.

At the top of the initial display is the program prompt, **iac:**, as well as a menu of all of the command words that can be entered by the user.

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

Retrieving an individual from his name. To illustrate retrieval of the properties of an individual from his name, we will use Ken as our example. To specify external input to the name unit for Ken, you use the *input* command. When you type this command to the **iac:** prompt, the program asks if you want to clear all previous inputs (there are none, so you can enter *y* or *n*). Then the command prompts for a unit name or number, followed by an external input value for that unit. In this case you simply enter *Ken* as the name and 1 as the input value.

The program will accept as many name-value pairs as you wish to enter; to stop entering pairs just enter *end* or type *return*. When you've done this, the screen will be updated to indicate the external inputs you have specified. Note that external input of 1.00 displays as "**"; so if you have provided external input to *Ken* with value 1, there will be two stars instead of a 0 next to his name. After this update, you will get back the **iac:** prompt, and you are ready to run your first simulation.

To start processing, enter the *cycle* command. This causes the program to run for 10 cycles. As it runs each cycle, it will update the screen, but this will generally occur rather quickly. If you'd like to watch the activation process unfold one cycle at a time, you can place the program in single-step mode by entering

set single 1

Having turned on single mode, you can continue to run more cycles if you wish. Alternatively, you may want to start over. To do that, enter *reset*, then enter *cycle* again. Now the screen will be updated after each cycle and processing will pause with the prompt:

p to push /b to break /<cr> to continue:

When you are ready to let the program run the next cycle just type *return*. In this way you can step through the cycles, one at a time, and examine at your leisure what happened on each cycle. Processing will again terminate at the end of 10 cycles. If you would like to allow processing to go on for longer, you can simply set *ncycles* to a larger number, say 100, by entering

set ncycles 100

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 about 100 cycles, you'll have about reached asymptotic activation values. (You may, of course, find it convenient to set single back to 0 at some point in this process.)

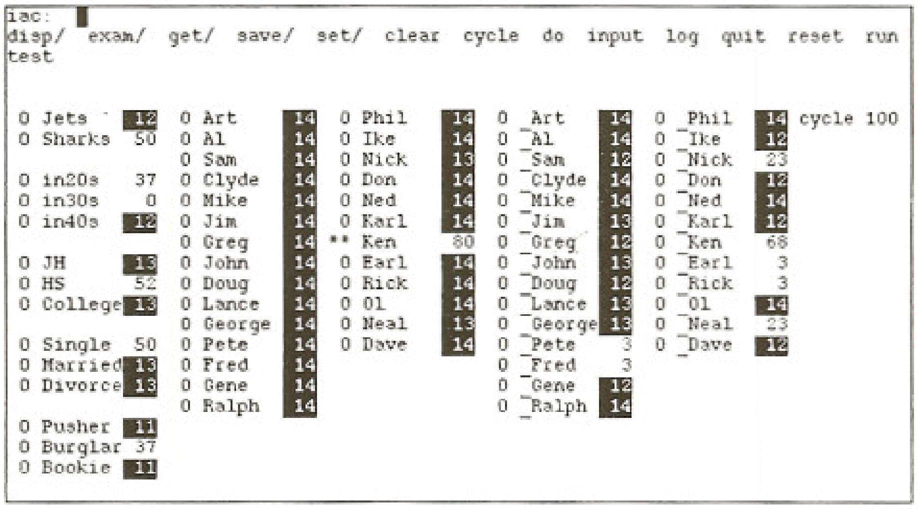
A picture of the screen after 100 cycles is shown in Figure 4. At this point, you can check to see that the model has indeed retrieved the pattern for Ken correctly. There are also several other things going on that are worth understanding. Try to answer all of the following questions (you'll have to refer to the properties of the individuals, as given in Figure 1).

Q.2.1.1. Why are some of Ken's properties more strongly activated than others?

Q.2.1.2. None of the other name units were activated, yet several other instance units are active (i.e., their activation is greater than 0). Explain this difference.

Q.2.1.3. Why are some of the active instance units more active than others?

Q.2.1.4. The *in30s* unit is receiving almost as much excitation as the *in20s* unit. Why is the latter so much more active than the former?



```
iac:
disp/  exam/  get/  save/  set/  clear  cycle  do  input  log  quit  reset  run
test


 0 Jets      12  0 Art      14  0 Phil     14  0 _Art     14  0 _Phil    14 cycle 100
 0 Sharks    50  0 Al       14  0 Ike      14  0 _Al      14  0 _Ike     12
                 0 Sam      14  0 Nick     13  0 _Sam     12  0 _Nick    23
 0 in20s     37  0 Clyde    14  0 Don      14  0 _Clyde   14  0 _Don     12
 0 in30s      0  0 Mike     14  0 Ned      14  0 _Mike    14  0 _Ned     14
 0 in40s     12  0 Jim      14  0 Karl     14  0 _Jim     13  0 _Karl    12
                 0 Greg     14 ** Ken      80  0 _Greg    12  0 _Ken     68
 0 JH        13  0 John     14  0 Earl     14  0 _John    13  0 _Earl     3
 0 HS        52  0 Doug     14  0 Rick     14  0 _Doug    12  0 _Rick     3
 0 College   13  0 Lance    14  0 Ol       14  0 _Lance   13  0 _Ol      14
                 0 George   14  0 Neal     13  0 _George  13  0 _Neal    23
 0 Single    50  0 Pete     14  0 Dave     14  0 _Pete     3  0 _Dave    12
 0 Married   13  0 Fred     14                0 _Fred     3
 0 Divorce   13  0 Gene     14                0 _Gene    12
                 0 Ralph    14                0 _Ralph   14
 0 Pusher    11
 0 Burglar   37
 0 Bookie    11
```

FIGURE 4. The display screen after 100 cycles with external input to the name unit for Ken.

If you can answer all of these questions correctly, you understand the interactive activation and competition process rather well.

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*. To do this, enter the *input* command, enter *y* to the question about clearing the previous inputs, and then specify external inputs of 1 for *Shark* and *in20s*. After getting back to the **iac:** prompt, enter *reset* and then enter *cycle*. Run a total of 100 cycles again, and take a look at the state of the network.

Q.2.1.5. Describe the differences between this state and the state at the end of the previous run. What are the main differences?

Q.2.1.6. Explain why the occupation units show partial activations of units other than Ken's occupation, which is Burglar. Take the explanation as far as you can, contrasting the current case with the previous case as much as possible.

In comparing this case with the previous one, it may be useful for you to make a copy of the state of the screen for future reference. To do so, just enter

save screen

and give a file name as requested. Note that negative numbers are saved with minus signs instead of in reverse video, so that they can easily be printed.

Graceful degradation. What is the effect of erroneous information in the probe supplied to an IAC network? Here we examine how errors influence the ability of the network to retrieve an individual's name from a description of his properties. To begin to explore this issue, run the model twice: once activating all of Ken's properties other than his name via the input command and once with the same external input, but activating *JH* instead of *HS*. Note that for the second case, you can retain the old input specification, and then just set the input to *HS* to 0 and the input to *JH* to 1 before retesting.

Q.2.1.7. How well does the model do with the "noisy" version of Ken compared to the correct version of Ken? Would it do this well with all noisy versions of individuals? Test with at least one other individual, and explain your results.

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. We illustrate by using the *set/ weights* command to set the weights between the instance unit for Lance and the property unit for Burglar to 0. First, run 100 cycles, providing external input of 1 to *Lance*, to see what happens before we delete the connections. Then, remove these connections as follows. To the **iac:** prompt type

```
set weight Burglar _Lance 0
set weight _Lance Burglar 0
```

Then reset the network, and run 100 cycles again.

Q.2.1.8. 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?" Be sure to reinstall the connections between *Burglar* and *_Lance* (set them back to 1). Once you've done that, you can ask the model to generalize about the Jets by providing external input to the Jets unit alone, then cycling 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.2.1.9. 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?

Ex. 2.2. 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*).

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.

Hint. For this exercise, you should first consider the asymptotic activations of units—what do you expect based on the discussion in the "Background" section? If you wish to follow the time course of activation, you can use the *log* command to store a record of processing in a file, then you can use the **colex** and **plot** utilities to make graphs of activations vs. cycle numbers for selected units. A discussion of how to do this is given in Appendix D.

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.

Ex. 2.3. Grossberg Variations

Explore the effects of using Grossberg's update rule rather than the default rule used in the IAC model. This requires you to use the command

```
set mode gb 1
```

Now redo one or two of the simulations from Exercise 1.

Q.2.3.1. What happens when you repeat some of the simulations suggested in Exercise 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.

Hints. 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.

Ex. 2.4. Construct Your Own IAC Network

Construct a task that seems appropriate for an IAC network, along with a knowledge base (in the form of a *.net* file), and explore how well the network does in performing your task. You may find it useful to set up *.str*, *.tem*, and *.loo* files, in addition to the *.net* file. (Detailed specifications for the *.net*, *.tem*, and *.loo* files are given in Appendix C.)

Q.2.4.1. Describe your task, 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.

Hints. 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.