Clark, A. (1993) *Associative Engines: Connectionism, Concepts, and Representational Change*. Cambridge, MA: MIT Press.
Cowan, J. D. and Sharp, D. H. (1988) Neural nets and artificial intelligence. *Daedalus*, 117, 85–121.
Cummins, R. and Cummins, D. D. (eds) (2000) *Minds, Brains, and Computers: The Foundations of Cognitive Science: An Anthology*. Oxford: Blackwell.
Franklin, S. (1995) *Artificial Minds*. Cambridge, MA: MIT Press.
Grossberg, S. (1982) *Studies of Mind and Brain: Neural Principles of Learning, Perception, Development, Cognition, and Motor Control*. Dordrecht: Reidel.
Rutgers Optimality Archive (ROA): electronic repository of papers on Optimality Theory at *http://ruccs.rutgers.edu/roa.html*.

2

CONNECTIONIST ARCHITECTURES

Connectionist networks are intricate systems of simple units which dynamically adapt to their environments. Some have thousands of units, but even those with only a few units can behave with surprising complexity and subtlety. This is because processing is occurring in parallel and interactively, in marked contrast with the serial processing to which we are accustomed. To appreciate the character of these networks it is necessary to observe them in operation. Thus, in the first section of this chapter we will describe a simple network that illustrates several features of connectionist processing. In the second section we will examine in some detail the various design principles that are employed in developing networks. In the final section we will discuss several appealing properties of networks that have rekindled interest in using them for cognitive modeling: their neural plausibility, satisfaction of "soft constraints," graceful degradation, content-addressable memory, and capacity to learn from experience. Connectionists maintain that the investment in a new architecture is amply rewarded by these gains but, as we will also note, they must overcome some serious challenges.

2.1 The Flavor of Connectionist Processing: A Simulation of Memory Retrieval

We will begin by describing a connectionist model which McClelland (1981) designed for the purpose of illustrating how a network can function as a content-addressable memory system. Its simple architecture conveys the flavor of connectionist processing in an intuitive manner. The information to be encoded concerns the members of two gangs, the Jets and the Sharks, and some of their demographic characteristics (figure 2.1). Figure 2.2 shows how this information is represented in a network, focusing on just five of the 27 gang members for readability. These figures are redrawn (including corrections) from McClelland and Rumelhart's *Handbook* (1988, pp. 39, 41), which uses the gang database for several exercises; there is related discussion by Rumelhart, Hinton, and McClelland (1986) in *PDP:2* (pp. 25–31). In this section we present the results of several different runs which we performed on the Jets and Sharks network using the **iac** (interactive activation and competition) program in chapter 2 of the *Handbook*. It will be seen that this simple network could retrieve names of individuals from properties, retrieve properties from names, generalize, and produce typicality effects.

The Jets and the Sharks

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

Figure 2.1 Information about members of two gangs, the Jets and Sharks. Reprinted by permission of author from J. L. McClelland (1981) Retrieving general and specific knowledge from stored knowledge of specifics, *Proceedings of the Third Annual Conference of the Cognitive Science Society*. Copyright 1981 by J. L. McClelland.

2.1.1 Components of the model

The most salient components of a connectionist architecture are: (a) simple elements called *units*; (b) equations that determine an *activation* value for each unit at each point in time; (c) weighted *connections* between units which permit the activity of one unit to influence the activity of other units; and (d) *learning rules* which change the network's behavior by changing the weights of its connections. The Jets and Sharks model exhibits components (a)–(c); we defer the important topic of learning until later.

(a) *The units* There are 68 units in the complete model: a unit for each gang member (27 units); a unit for each gang member's name (27 units); and a unit for each of the properties members can exhibit (14 units). The units are grouped into seven clusters (the "clouds" in figure 2.2); within each cluster the units are mutually exclusive.[1] In addition to two clusters for the members and their names, there are five clusters for properties that distinguish the members (age, occupation, marital

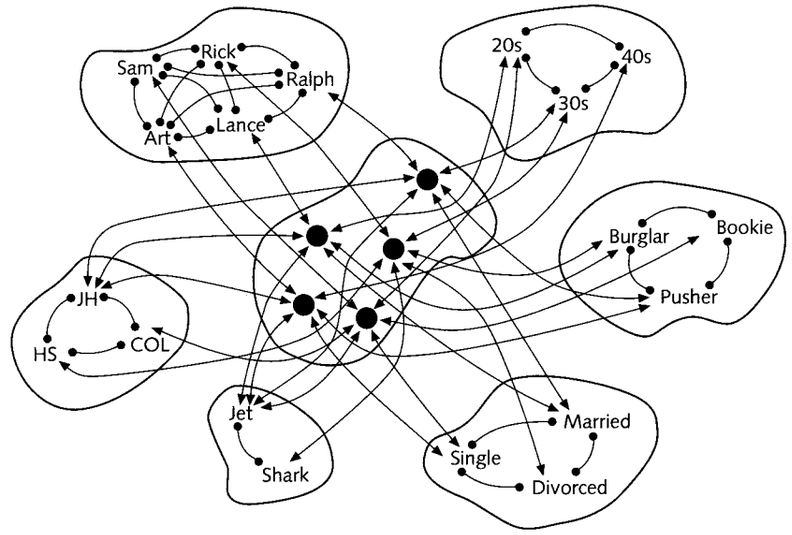

Figure 2.2 McClelland's (1981) Jets and Sharks network. Each gang member is represented by one person unit (center) that is connected to the appropriate name and property units. Only 5 of the 27 individuals from figure 2.1 are included in this illustration. Adapted (with corrections) from J. L. McClelland (1981) Retrieving general and specific knowledge from stored knowledge of specifics, *Proceedings of the Third Annual Conference of the Cognitive Science Society*. Copyright 1981 by J. L. McClelland.

status, educational level, and gang membership). Note that the names are regarded as a special kind of property; the name cluster is just one cluster among others around the periphery. Each individual gang member is represented, not by his name, but by a person unit in the center cluster that is connected to the appropriate name and property units. As a notational convention in the equations that follow, any of these units can be referenced by the variables u (the *unit* of interest) and i (a unit that provides *input* to u).

(b) *Activations* Associated with each unit is an activation value, $activation_u$. Initially each unit is set at a "resting activation" of -0.10. When a simulation is run, the activations vary dynamically between the values -0.20 and $+1.00$, reflecting the effects of external input, the propagation of activation from other units in the system, and decay over time. External input is the activation of certain units by the environment (in practice, the investigator, who wishes to observe the effects). It is only the property and name units, however, that can receive external input; for this reason they are referred to as the *visible* units. The person units cannot be directly accessed from outside the network, and are therefore referred to as *invisible* or *hidden* units. Their only source of change in activation, besides decay, is the propagation of activation from other units to which they are connected.

(c) *Weighted connections* In this particular network, all connections are bidirectional and are assigned a binary weight. Wherever there is a connection from unit i to unit u with $weight_{ui}$, there is a converse connection from unit u to unit i with $weight_{iu}$ of

the same value. (Conventionally, the order of subscripts is the reverse of the direction of propagation of activation.) Specifically, for each person unit there is a two-way excitatory connection (with weight + 1) with that person's name and with each of his properties (one property unit per cluster). Hence, a person unit propagates activation to all of its property units, and a property unit propagates activation to all of the units for persons who exhibit that property. Weights of – 1 are used to form inhibitory connections among units within a cluster; hence, activation of one property tends to suppress the activity of other properties in its cluster. (However, we deleted the inhibitory connections between name units to obtain certain generalizations across names.) For example, if the property **divorced** is activated, the immediate effects are that the property units **single** and **married** will become less active (due to their inhibitory connections with **divorced**) at the same time that the person units **JIM, GEORGE, RICK**, and **DAVE** will become more active (due to their excitatory connections with **divorced**). Note that person units like **RICK** (which are hidden units) are indicated by upper case, and name units like "**Rick**" (which are visible units) by lower case in quotes.

2.1.2 Dynamics of the model

The Jets and Sharks model exhibits a variety of interesting behaviors when it performs memory retrieval tasks. To understand the dynamics, it is important to work through the equations that govern the propagation of activation through the network. In this section we introduce the general task of memory retrieval and then describe the equations that are involved in carrying it out. In the final section, we illustrate the operation of the network by tracking its performance across several specific memory retrieval tasks.

2.1.2.1 Memory retrieval in the Jets and Sharks network To simulate a memory retrieval task, we supply an external input to one or more of the visible units and observe the effects. For example, to simulate using Art's name to retrieve his properties, we can increase the input into "**Art**" (Art's name unit). The excitatory connections in the network will propagate this activation first to the person unit **ART**, and from there to the units for Art's properties. This is only the beginning, however; the increased activation will continue to reverberate through the network across numerous cycles of processing, during which Art's property units will become increasingly active (in addition to other, less direct effects). At the same time, each active unit will send inhibitions to other units in its cluster. Every change in activation produces additional changes in other units, and the process of dynamically changing activation values can be repeated many times. For tractability, it is useful to set up discrete processing cycles; once per cycle, the fixed amount of external input is again supplied, and each unit sends and receives excitations and inhibitions and updates its own activation. After a number of cycles, the system will stabilize so that the input to each unit will be precisely that which enables it to retain its current activation. (In the language of dynamical systems, discussed in section 8.2.1, the network has settled into a *point attractor*.) At this time, only a subset of the units will have high activation values. In our "**Art**" example, the units that would stabilize at high activations include **ART**, **Jets**, **single**, **pusher**, **40s**, and **junior high**. Thus, by querying the network with a name, we recovered the person's other properties.

2.1.2.2 The equations To explain how these effects are produced, we will present some of the relevant equations. We have made every effort to make this material accessible even to those with some degree of math anxiety. To enhance readability, we use English-like labels for variables and constants; most are similar to those in McClelland and Rumelhart's (1988) *Handbook*. The subscripts that we use to index units are mnemonic (and therefore idiosyncratic). It is fairly straightforward to translate our equations into the *Handbook*'s relatively accessible notation. To aid with transfer to the somewhat less accessible notation in Rumelhart and McClelland's (1986) *PDP* volumes, we provide translations of important equations in Appendix A (p. 349). Notation varies widely in connectionist modeling, and we leave it as an exercise for the reader to carry out any additional translations when reading primary sources.

Most of the equations can be viewed as focusing on a particular unit, for example, a unit whose activation is being calculated. We refer to this unit as u. (Actually, in its usual use as a subscript, u is an index that ranges over all of the units to which the equation will be applied.) Often the equation refers as well to another unit (or units) that is feeding into u; we refer to such a unit as i. (This notation is not particularly mnemonic here, but it will be later when we discuss feedforward networks.) To propagate activation, each unit i sends an excitatory or inhibitory output to every unit u to which it is connected. In the simplest case, the output sent by a unit would be identical to its activation. In practice, a variety of output functions have been explored. For the Jets and Sharks model as implemented in the *Handbook*, the output is identical to the activation if it is above a threshold of zero, and is set at zero otherwise. That is:

$$output_i = activation_i \text{ if } activation_i > 0 \text{ and } output_i = 0 \text{ otherwise.} \quad (1)$$

When i's output value is multiplied by the *weight* of its connection with u, the resulting value serves as an *input* to u:

$$input_{ui} = weight_{ui}\, output_i \quad (2)$$

For the Jets and Sharks network, in which weights are either + 1 or – 1, the weight simply determines the sign of the input (whether it is excitatory or inhibitory). In most models, the weight varies within a continuous range, such as + 1 to – 1, and therefore affects the magnitude of the input as well.

Next, the concept of *net input* is needed. Unit u receives input from all of the units to which it is connected. Usually these inputs are simply added together, and the total multiplied by a strength parameter, to obtain the net input to u. (The strength parameter is simply a number that is selected to scale down the input to a desired degree; the lower its value, the more gradual are the changes in activation values.) However, if u is in contact with the environment (as are the property and name units in Jets and Sharks), it might also receive an external input. In this model any external input is supplied at a value determined by the modeler, which is then scaled by its own strength parameter. The two strength parameters allow for adjusting the relative influence of internal input versus external input; we have used the *Handbook*'s default values of 0.1 (internal) and 0.4 (external). (There is an option of setting different internal strength parameters for excitatory versus inhibitory inputs; for simplicity we omit that distinction here.) Therefore, for the options we have taken, the equation for calculating the net input is:

$$netinput_u = 0.1 \sum_i weight_{ui}\ output_i + 0.4\ extinput_u \quad (3)$$

The term in this equation that begins with a summation sign (Σ) with an index i tells us that the input to u from each unit i is calculated as in equation (2) above, and that the inputs from all of the i units are then added together for inclusion in the *netinput*. On the basis of the net input, the unit will now either increase or decrease its activation according to a fairly simple activation rule, as shown in equations (4) and (5) below. We will use a_u to represent the current $activation_u$, and Δa_u to represent the net change to be made to $activation_u$. There are two terms in the equation. The first calculates the change that is due to the net input (an increase for positive net input, a decrease for negative net input). The second term is a decay term that decreases activation, even in the absence of net input. (One effect of this is that external input has its greatest effect when it is first presented to a unit.) Because the first term depends on the sign of the input, there are two versions of the equation. If the net input is positive (greater than 0), then the change in the activation is given by:

$$\Delta a_u = (\max - a_u)\ (netinput_u) - (decayrate)\ (a_u - rest) \quad (4)$$

Here *max* represents the maximum activation value that a unit can take (1 in this case). Hence, the first term says that if we have a positive net input, we scale it by a multiplier that depends upon how far the current activation is from the maximum activation, and then increase the activation by that amount. Thus, the greater the net input and the lower the current activation, the more we increase the activation. The decay term, which is subtracted from that amount, is determined by the decay rate (which is set at 0.1) and the difference between the current activation and the unit's resting activation (which is set at – 0.1). Thus, the lower the current activation, the less we adjust for decay.

If the net input is less than or equal to 0, the change in activation is given by:

$$\Delta a_u = (a_u - min)\ (netinput_u) - (decayrate)\ (a_u - rest) \quad (5)$$

The decay term is the same as above. If the net input is 0, the unit will simply decay by that amount. When the net input is negative, on the other hand, we will determine how much further to decrease the activation by multiplying the net input by the difference between the current activation and the minimum activation (which is set here at – 0.2). Hence, the greater the current activation, the greater is the effect of negative input in decreasing that activation.

2.1.3 Illustrations of the dynamics of the model

With the basic machinery in place, we now can work through what happens in the network when it performs memory retrieval tasks. By varying the queries that we present to the Jets and Sharks network, we can observe it perform several different tasks: retrieving properties from a name, retrieving a name from properties, categorization, prototype formation, and utilizing regularities.

2.1.3.1 Retrieving properties from a name This is the task that we briefly described above. The investigator activates the "**Art**" unit (by supplying it with external input), and consequently Art's properties become activated. On cycle 1, every unit's

current activation is equal to the resting activation of – 0.10. Equation (1) specifies that any unit with an activation below the threshold of zero (0.0) produces an output of 0.0; since this is the case for all units, their net inputs to other units are also 0.0. The name unit "**Art**" is supplied with an external input of 1.00, with the result that it is the only unit with a non-zero net input. By equation (3):

$$netinput_{\text{"Art"}} = (0.10)\ (0.0) + (0.40)\ (1.00) = 0.40$$

This strong net input causes the activation of "**Art**" to increase. By equation (4):

$$\begin{aligned}\Delta a_{\text{"Art"}} &= (1.00 - (-0.10))\ (0.40) - (0.10)\ (-0.10 - (-0.10)) \\ &= (1.00 + 0.10)\ (0.40) - (0.10)\ (0.00) = 0.44 - 0.0 = 0.44\end{aligned}$$

While all other units, including the hidden (person) units, remain unchanged at the resting activation value:

$$\begin{aligned}\Delta a &= (-0.10 - (-0.10))\ (0.0) - (0.10)\ (-0.10 - (-0.10)) \\ &= 0.0 - 0.0 = 0.0\end{aligned}$$

Because the current activation of "**Art**" for cycle 1 is – 0.10, adding 0.44 yields a new activation of 0.34. For all other units, adding 0.0 to – 0.10 yields a new activation that is the same as the current activation, – 0.10. These new activations are used as the current activations for cycle 2.

Beginning on cycle 2 the activation of "**Art**," which now is positive, sends excitatory (positive) input to **ART**, the person unit for Art. By cycle 4, **ART** has climbed to a positive activation. At the same time "**Art**" continues to grow in activation. This is partly due to the continued presentation of external input on each cycle, and partly (beginning in cycle 4) from the input it begins to receive from **ART**. After **ART** becomes positively activated, it begins to send excitatory inputs to the units for Art's properties. Thus, on cycle 5 the units **Jets, 40s**, etc., start to become less negative and eventually become positive (on cycle 12). Once Art's properties become positive, the competing properties in their clusters, such as **Sharks, 20s**, and **30s**, become slightly more negative. The reason is that the units for **Jets** and **40s** send inhibitory inputs to their competitors, thus driving them below the resting activation. These changes in activation are illustrated for the age property cluster in figure 2.3.

Hence, the person unit **ART** becomes active during the early cycles of processing, and by propagating activity to the property units to which it is connected, enables the retrieval of Art's properties. Beginning with cycle 18, though, some other activities begin to appear in the network. The person units **CLYDE** (and also **RALPH**, not shown), and subsequently **MIKE** (and also **FRED** and **GENE**, not shown) become less negative, and on cycle 25 **CLYDE** becomes positive. The reason for this is that the units for Art's properties begin to send positive activations to the units for persons who share properties with Art. Clyde, in fact, shares all of Art's properties except for occupation (he is a bookie whereas Art is a pusher). Mike shares three of five properties with Art, and so his person unit also begins to rise in value, but not sufficiently to achieve a positive activation.[2] Eventually the activation on **CLYDE** becomes high enough that it sends a positive input to the name unit "**Clyde**," and it too becomes active. The result is that by accessing the system through "**Art**," we not only get back Art's properties, but also the names of people similar to Art. One way to interpret this process intuitively is to note that thinking about a person's properties may tend to remind us of people who are very similar to that person.

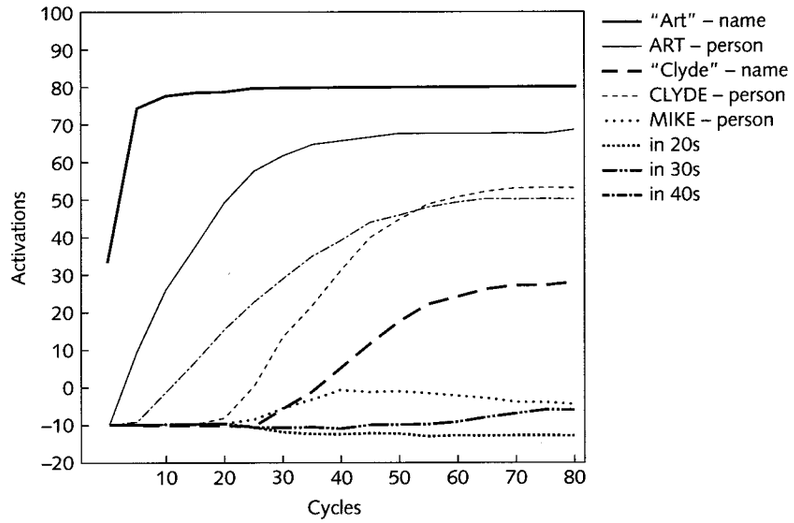

Figure 2.3 The activation values across cycles of some of the units in the Jets and Sharks network after the name unit "**Art**" is activated by an external input.

2.1.3.2 Retrieving a name from other properties The network is even more versatile than this, however. Suppose we access it by supplying external inputs simultaneously to several units, namely, the units for Art's demographic properties (**40s, junior high, single, pusher**). These will activate **ART**, which will activate "**Art**," and in this way Art's name will pop out. This is the clearest illustration of what is meant by a *content-addressable memory*: the name is retrieved by supplying contents (see the discussion of content-addressable memory in section 2.3.4). Generalization comes free along with this capability; that is, names of persons with similar properties will pop out also at a lower degree of activation.

2.1.3.3 Categorization and prototype formation The same memory retrieval processes can produce less obvious phenomena, which have been observed in human categorization performance. First, the network can recover category instances. If we supply external input to **Sharks**, for example, that unit will activate the person units for the individual Sharks. Second, as processing continues these individuals become graded according to how well they exemplify the category (analogous to the human ability to judge the relative typicality of various category members; see Rosch, 1975). Figure 2.4 shows the activation across cycles for three of the person units and three of the name units after we activated **Sharks**. Some names clearly acquire more activation than others. For example, after 70 cycles of processing, "**Phil**" is most active, "**Don**" is less active, and "**Dave**," after being initially activated, has dropped almost to its resting level. What causes this emergence of grading by typicality? The third capability, the extraction of prototypes, provides a key part of the mechanism (and is exhibited by humans as well; see Posner and Keele, 1968). Activating **Sharks** results in the activation of the person units for the Sharks, which results in the activation of the property units to which they are connected. The most widely shared properties become the most active. Thus, in figure 2.5 we see that the **30s**

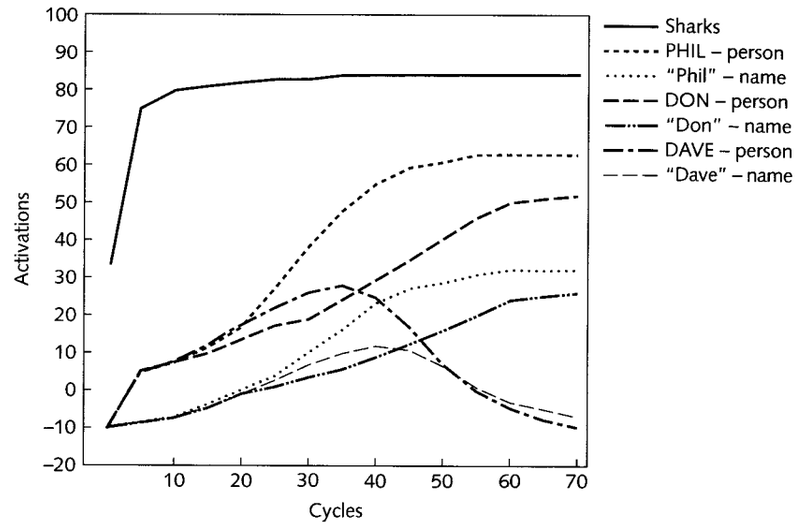

Figure 2.4 The activation values across cycles for name and person units of various members of the Sharks after the property unit **Shark** is activated by an external input. The name units become less active than the person units, because the name units receive activation only via the person units.

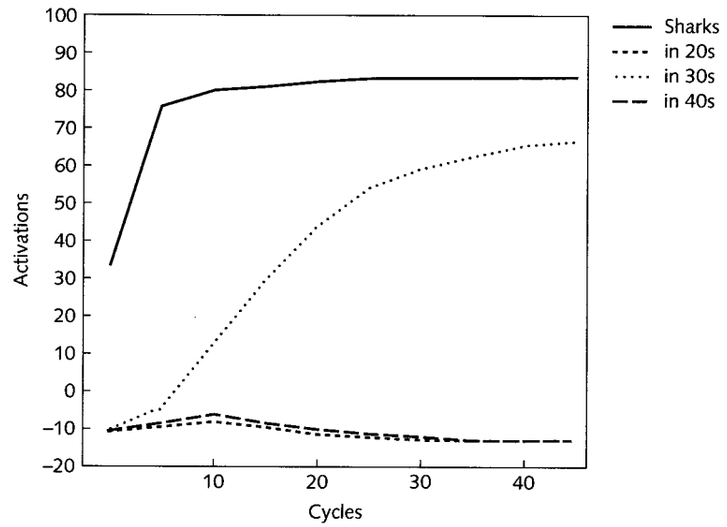

Figure 2.5 The activation values across cycles for the three age units after the property unit **Shark** is activated by an external input.

unit becomes quite active, while the **20s** and **40s** units never rise much above their resting level. This is due to the fact that nine of the twelve Sharks are in their **30s**. These activations are then forwarded to those person units that exhibit the most frequent properties, thus creating a positive feedback loop which further sharpens

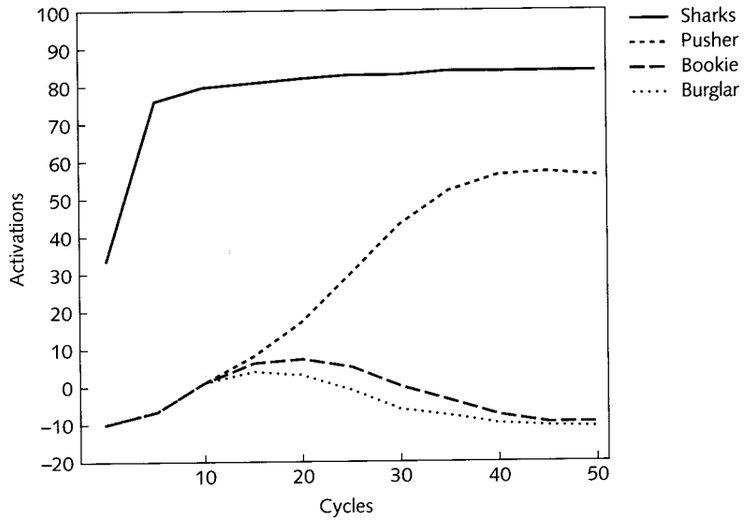

Figure 2.6 The activation values across cycles for the three occupation units when the property unit **Shark** is activated by an external input.

the prototype. The name units are too individual to regard them as part of the prototype, but they change by the same process as the other properties, and their activations come to reflect the extent to which each individual displays the prototypical properties (hence displaying the second capability mentioned just above).

An interesting twist can be observed for the occupation properties in figure 2.6. Initially the units for all three occupations rise in activation. This is because the gang members are equally distributed in their chosen careers. Nevertheless, subsequently the unit for **pusher** continues to grow in activation, while those for **burglar** and **bookie** drop back below zero. This is due to the fact that those individuals who provide the best match on those properties that are not equally distributed grow in their activations, and so provide increased activation to pusher, and it, in turn, inhibits both of the other occupation units.

2.1.3.4 Utilizing regularities As a final example of the variety of ways in which the memory can be queried, we can activate two properties, such as **20s** and **pusher**, and discover which individuals are most likely to fit that scenario. The network will initially produce higher activations in the person units for all individuals who possess any one of these properties, with those sharing both properties (**GREG, FRED**, and **GENE**) getting the highest activations. As person units become active, they not only activate name units, but also other property units. The units for the most widely shared properties (**Jet, single**, and **high school**) also became more active than other units in their cluster. This leads to Pete's person and name units receiving significant activation even though Pete did not fit the original description, since he is a bookie, not a pusher. Thus, the network not only identified which individuals shared the initial pair of properties, but what their other properties were likely to be, and who amongst those not possessing the initial pair show the best fit with those who did satisfy the initial pair of properties. Making inferences from known properties to other properties is a kind of behavior that is familiar to social psychologists working in attribution theory.

2.2 The Design Features of a Connectionist Architecture

In the Jets and Sharks simulation we have presented one particular network architecture that has some very nice characteristics for modeling recall of information from memory and for illustrating some of the capabilities of connectionist networks. However, this design is not suitable for most purposes, and work has proceeded using a variety of other designs. In fact, connectionism as a research paradigm is still in its infancy, and investigators are still in the process of exploring different kinds of connectionist systems. Many of the design features are rather complex, and require considerable mathematics to characterize. In order to provide a general overview of the various types of systems, we will bypass material that is foundational but complex. (For example, we make no direct use of vector notation[3] or matrix algebra.) Also, in this chapter and the next, we limit ourselves to those architectures that are emphasized in Rumelhart and McClelland's (1986) *PDP* volumes, which can be consulted for more depth. For other technical treatments of these or other architectures see, in particular, J. A. Anderson (1995), Ballard (1997), Grossberg (1982, 1988), Kohonen (1988), Levine (1991), and Wasserman (1989). We can characterize the distinctions between different connectionist architectures by considering four issues: (a) how the units are connected to one another; (b) how the activations of individual units are determined; (c) the nature of the learning procedures which change the connections between units; and (d) the ways in which such systems are interpreted semantically. We use a mnemonic notation in these sections; see Appendix A for a translation into two different notations used by Rumelhart and McClelland in the *PDP* volumes.

2.2.1 Patterns of connectivity

The first decision in setting up a connectionist network is to determine which units are connected to one another, that is, the pattern of connectivity. There are two major classes of patterns: (a) *feedforward networks* have unidirectional connections (inputs are fed into the bottom layer, and outputs are generated at the top layer as a result of the forward propagation of activation); (b) *interactive networks* have bidirectional connections. The Jets and Sharks exercises illustrate how interactive networks change state gradually over a large number of processing cycles, as dynamically changing activations are passed back and forth over the two-way connections. We will discuss each of these classes in turn.

2.2.1.1 Feedforward networks In feedforward networks, units are organized into separate layers, with units in one layer feeding their activations forward to the units in the next layer until the final layer is reached. The simplest such configuration consists of only two layers of units: *input units* and *output units*. There is a weighted connection from each input unit to each output unit. When the weights (connection strengths) are properly set, this type of network can respond to each of a variety of

input patterns with its own distinctive output pattern; therefore, it is sometimes referred to as a *pattern associator*.

For example, consider a network with four input units ($i_1 - i_4$), each of which is connected unidirectionally to each of four output units ($u_1 - u_4$), with output activations allowed to range over a continuous domain (figure 2.7). Several input patterns are constructed, each of which consists of a series of four binary values (+ 1 and − 1). When a pattern is presented to the input layer, each of its binary values is the external input to one input unit, which takes that value as its activation. In presenting the input pattern + 1 − 1 − 1 + 1, for example, an external input of − 1 is supplied to the second input unit (i_2), so its activation becomes − 1. The activations of the input units are then propagated to the output units by an activation rule that can supply a different weighted sum of the various input activations to each output unit. Therefore, each output unit achieves an activation value that reflects the activity of some input units more than others (see the following section for details). The activation patterns across the input and output units are, in mathematical terms, *vectors*. We will refer to them using the more familiar term *patterns*.

It is informative to compare figure 2.7 with figure 1.1 in the previous chapter. These figures illustrate the two approaches that are taken to diagramming two-layer networks. In figure 1.1, the sensory layer is drawn vertically and the motor layer horizontally. As a result, each connection must be drawn with a change of direction from horizontal to vertical, and its weight can be placed at the junction. The advantage is that the layout of the nodes for the weights is the same as in a weight matrix (cf. the matrix for Case A or B below). In figure 2.7, the layers are parallel and the connections are indicated by straight lines; each connection has a weight associated with it. The weights are not included in either illustration. This latter format has the advantage, though, that it can be adapted to more complex networks (see below).

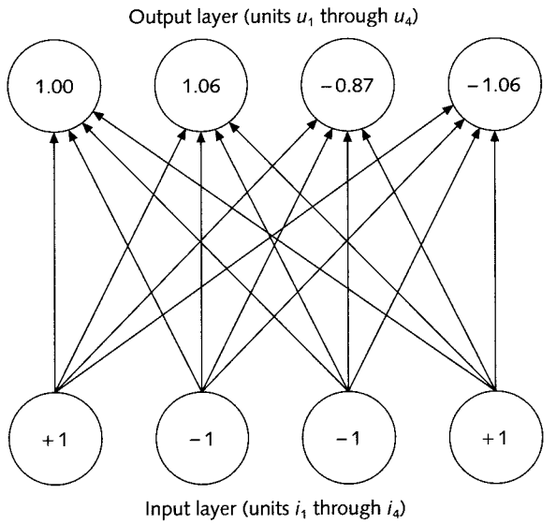

Figure 2.7 Pattern recognition network with two layers of units. Activations for the input layer are all set at + 1 or − 1; they are multiplied by the weight on each connection and summed to obtain the activations for the output layer. All connections are shown; each has a numerical weight that is not shown.

The paradigmatic task for a pattern associator is paired-associate learning in which the input–output pairings are arbitrary (e.g., supplying names for objects). However, with appropriate weights it could instead be used to reproduce each input pattern on the output layer, in which case it would be a type of *auto-associator*. Or, the output layer could be used to represent a small number of categories into which a larger number of input patterns would be sorted or classified. We will often use the term *pattern associator* generically for these varieties of two-layer networks.

Pattern associators have many useful applications, and we discuss several simulations that make use of them in subsequent chapters. There are, however, problems for which a two-layer network is inadequate, as we will discuss in section 3.2.1. A well-known example is the Boolean function of *exclusive or* (XOR), which is a special case of parity detection. To overcome the limitations of two-layer networks, it is necessary to add *hidden units* to the system. These are units which serve neither as input nor output units, but facilitate the processing of information through the system. We have already encountered hidden units in the Jets and Sharks network (in that network, however, there was no distinction between input and output units; all non-hidden units could serve both functions). In section 3.2.2 we illustrate how the XOR problem can be solved by a network with two input units, two hidden units, and one output unit. Most tasks for which multi-layered networks are used, however, require considerably more units in each layer. For examples, see the discussion of NETtalk in section 3.2.2 and the logic network in section 4.3.2.

As a point of terminology, note that investigators frequently refer to a network that has three layers of units as a *two-layer network*; in that case, it is the number of layers of *connections* that is being referenced. In this book we reference the number of layers of *units*, but reserve the term *multi-layered network* for networks with hidden units (i.e., three or more layers of units).

A number of variations can be made on two-layer and multi-layered architectures. One variation is to allow units in the same layer to send inhibitions and excitations to each other as well as to units in the next layer. A more interesting variation is the *recurrent network*, which can receive input sequentially and alter its response appropriately depending upon what information was received at previous steps in the sequence. It does this by feeding a pattern achieved on a higher layer back into a lower layer, where it functions as a type of input (Elman, 1990; see also the sequential networks of Jordan, 1986). We will discuss such networks in section 6.4. Finally, Rumelhart, Hinton, and McClelland (1986, *PDP:2*) discuss the interesting idea that multi-layered feedforward networks could be used for top-down rather than bottom-up processing, by reversing the direction of the connections without changing the units and patterns at each level. All of these variations soften the design constraint that activations be propagated exclusively in one forward pass. In the next section, we discuss a type of network that departs more dramatically from the basic feed-forward design.

2.2.1.2 Interactive networks For interactive networks, at least some connections are bidirectional and the processing of any single input occurs dynamically across a large number of cycles. Such networks may or may not be organized into layers; when they are, processing occurs backwards as well as forwards. A major exemplar of an interactive network is the *Hopfield net*, developed by the physicist John Hopfield (1982) by analogy with a physical system known as a *spin glass*. In their review, Cowan and Sharp (1988) characterized a spin glass as consisting of a matrix of atoms

which may be spinning either pointing up or pointing down. Each atom, moreover, exerts a force on its neighbor, leading it to spin in the same or in the opposite direction. A spin glass is actually an instantiation of a matrix or lattice system which is capable of storing a variety of different spin patterns. In the analogous network that Hopfield proposed, the atoms are represented by units and the spin is represented by binary activation values that units might exhibit (0 or 1). The influence of units on their neighbors is represented by means of bidirectional connections; any unit can be (but need not be) connected to any other unit (except itself). As with any interactive network, activations are updated across multiple cycles of processing in accord with an activation rule (see section 2.2.2).

Hopfield (1984) has also experimented with networks taking continuous activation values. Other examples of interactive networks include *Boltzmann machines* (Hinton and Sejnowski, 1986, in *PDP:7*) and *harmony theory* (Smolensky, 1986, in *PDP:6*). As in the original Hopfield nets, the units take binary activation values. We will not discuss harmony theory further, but it uses a probabilistic activation rule and a *simulated annealing* technique very similar to those of the Boltzmann machine (see section 3.2.3).

2.2.2 Activation rules for units

Networks differ not only in their pattern of connectivity, but also in the activation rules that determine the activation values of their units after processing. We have already encountered the major classes of possible activation values. (a) Discrete activations are typically binary, taking values of 0 and 1 (as in Boltzmann machines and harmony theory) or -1 and $+1$. (b) Continuous activations can be unbounded or bounded. As examples of bounded ranges, -0.2 to $+1.0$ was stipulated for the Jets and Sharks network, and a range of -1 to $+1$ is a common choice. In figure 2.7, we used binary input activations and continuous unbounded output activations.

Even greater variation is found in activation rules, which specify how to calculate the level of activation for each unit at a given time. In the following sections we present some of these rules, first for feedforward networks and then for interactive networks. Often the rules for the two types of networks are quite similar, and we will find a rule used in the interactive Jets and Sharks network useful as a framework for introducing our first feedforward activation rule.

First, though, a note of clarification. You may find it odd that the very investigators who reject rules and representations talk about *activation rules* and *learning rules*. These are not rules in the sense intended by symbolic theorists; rather, they refer to mathematical manipulations and often are referred to using the alternative terms *activation functions* and *learning functions*.

2.2.2.1 Feedforward networks Recall that for the Jets and Sharks network, the activation rule for unit u made use of the net input to that unit:

$$netinput_u = 0.1 \sum_i weight_{ui}\, output_i + 0.4\, extinput_u \tag{3}$$

Note that the net input has two components: the effects of activity in other units to which u is connected, and the effects of external input.[4] In a pattern associator (a two-layer feedforward network), the functions served by these two components

are divided between specialized sets of units. As shown in figure 2.7 above, units in the input layer (i) are specialized to receive external input, and take the values of the input patterns as their activations. The input unit's activation depends only on the external input, and therefore does not need to be determined by an activation rule. Based on that activation, the input unit then sends an output along each of its connections. In the simplest case, $output_i = activation_i$, but other functions are possible.

Units in the output layer (u) are specialized to receive activation from other units in the network (rather than receiving external input). The terminology now gets a bit confusing, because "input" and "output" are used to refer to the transmission of values as well as to types of units. Each input unit (i) sends output towards each output unit (u). To convert the input unit's output into the output unit's input, $output_i$ is multiplied by the weight of the connection: $input_{ui} = weight_{ui}\, output_i$. Adding these together for every unit i in the input layer yields the net input to u ($netinput_u$). The equation describing this is one component of equation (3), which was used for the Jets and Sharks network:

$$netinput_u = \sum_i weight_{ui}\, output_i \tag{6}$$

Optionally, a term $bias_u$ can be added to equation (6) in order to adjust the responsiveness of each output unit individually; it can be thought of as a fixed input supplied by a special unit that is not affected by what is happening in the rest of the system. If the value of the bias is low or negative, the output unit will respond conservatively to activation sent from the input units; if it is high, the output unit will behave "impulsively." We consider this version of equation (6) in section 3.2.2.

Finally, an activation rule is applied which makes use of the net input to determine the activation of each unit u. We will refer to $activation_u$ simply as a_u. In the simplest case, the *linear activation rule*, $a_u = netinput_u$ (producing a straight-line, or linear, function). This rule is very useful when two-layer networks are provided with patterns that meet certain constraints (see section 3.2.1). The additional power needed to violate those constraints can be obtained by adding one or more layers of hidden units, but only if the activation rule is also changed to a nonlinear function. Typically the function chosen is a continuous, monotonically increasing (or at least nondecreasing) function of the input for which a derivative exists. In particular, the *logistic function* has been widely used (in two-layer as well as multi-layered networks):

$$a_u = \frac{1}{1 + e^{-(netinput_u - \theta_u)/T}} \tag{7}$$

This function is sigmoidal in form (as figure 2.8 illustrates for the stochastic version of this function in the section on Boltzmann machines below). Within the exponent, θ_u is a threshold that is subtracted from the net input; it has the same effect on the activation value as adding a bias to the net input equation (6) if $\theta_u = -bias_u$. (In their 1988 *Handbook* McClelland and Rumelhart uniformly used a bias term, whereas in their 1986 *PDP* volumes they usually subtracted a threshold from net input instead.) T is a parameter which determines how flat the curve is across the range of net input values. (When the number of input units that feed into each output unit is large, the range of net input values also tends to be large. A higher value of T stretches the function so that it will cover this range.)

Each of these activation rules can be adapted to obtain discrete rather than continuous activation values, typically for use in networks in which both input and output units are binary (on or off). For the linear activation rule, the adaptation is to compare the net input to a threshold value. If net input exceeds the threshold, the output unit's activation is set to 1 (on); otherwise it is set to 0 (off). With a zero threshold, for example, positive net input turns the output unit on and negative net input turns it off. A unit that uses a threshold in this way is called a *linear threshold unit*. A network with an output layer of linear threshold units and an input layer of binary units is an *elementary perceptron* (Rosenblatt, 1959). Linear threshold units can also be used in the hidden and output layers of multi-layered feedforward networks and in interactive networks.

For the logistic activation rule, discrete activations can be achieved by using a stochastic version of equation (7); this is presented as equation (9) in the discussion of Boltzmann machines in the next section. When equation (9) is used in a feedforward network of binary units, presenting the same input pattern on different trials will not always have the same effect on a given output unit; that is, the relation between its net input and its activation becomes probabilistic. The equation determines the relative frequency with which the unit will turn on versus turn off. An example of a feedforward network with a stochastic activation function is discussed in section 5.2.2 (Rumelhart and McClelland's 1986 past tense model).

2.2.2.2 Interactive networks: Hopfield networks and Boltzmann machines The equations that govern the propagation of activation in feedforward networks can be used in interactive networks as well. A parameter t for time (or in some notations, n for cycle number) must be included, however, because activations are updated many times on the same unit as the system works towards settling into a solution to a particular input. (For readability, we show t in our equations only when it is necessary to distinguish it from $t + 1$.) Interactive networks may use a *synchronous* update procedure, in which every unit's activation is updated once per timing cycle, or an *asynchronous* update procedure, in which there is no common sequence of cycles, but rather a random determination of the times at which each unit separately is updated. Each update requires a separate application of the activation rule. (In contrast, in a feedforward network there is just one forward sweep of activation changes; the activation rule is applied just once to each unit.)

In the original Hopfield nets (Hopfield, 1982), each unit is a linear threshold unit. That is, the activation rule is the same as that of Rosenblatt's perceptron (but it is applied many times to each unit). On each update, if a unit receives net input that is above its threshold, it acquires an activation of 1. Otherwise, its activation is 0. (Alternatively, values of + 1 and – 1 can be used if the threshold is adjusted appropriately.) Hopfield employed an *asynchronous* update procedure in which each unit at its own randomly determined times would update its activation depending upon the net input it received at that time. (This helps to prevent the network from falling into unstable oscillations.) Processing is initiated in a Hopfield net by providing an initial input pattern to a subset of units (i.e., each unit receives an activation value of 1 or 0). Then, all units will randomly update their activations until a state is achieved in which no unit will receive a net input that would lead it to change its activation. If that occurs, the network is then said to have *stabilized* or reached a state of *equilibrium*. The particular stable configuration into which the network settles constitutes the system's identification of the initial input. (Some networks, however, never stabilize; rather: they behave as chaotic systems that oscillate between different configurations.)

Hopfield's analogy between this sort of network and a physical system paid an important dividend when he showed that one could calculate a very useful measure of the overall state of the network (*energy*, or E)[5] that was equivalent to the measure of energy in a physical system (Hopfield, 1982). A Hopfield net tends to move towards a state of equilibrium that is mathematically equivalent to a state of lowest energy in a thermodynamic system. Using the update rule described in the previous paragraph, each change in activation of any unit will result in an overall lower (or same) energy state for the system. In our notation, the global energy measure E is given by:

$$E = -\sum_{u<1} weight_{ui}\, a_u\, a_i + \sum_u \theta_u\, a_u \qquad (8)$$

To see how the update rule lowers the value of E, consider one example that focuses on just two units in a network of binary units taking activations of 0 and 1. If at the outset $a_u = 0$, $a_i = 1$, $weight_{ui} = 1$ and $\theta_u = 0.5$, then the contribution to E of these two units is: $-(1)(0)(1) + (0.5)(0) = 0$. Now assume that unit u has been randomly selected to have its activation updated. The input to unit u from unit i equals $weight_{ui}$ $a_i = (1)(1) = 1$. Since this exceeds the threshold of 0.5, unit u changes its activation from 0 to 1. We now evaluate E for this part of the network as: $-(1)(1)(1) + (0.5)(1) = -0.5$. Therefore, the network has moved to a state with a lower value for E. (Note that in actual practice, we must apply the update rule by considering all inputs to a_u not just that from a single a_i. This insures that we would change a_u only if it would contribute to an overall lower value for E.) Rumelhart, Smolensky, McClelland, and Hinton (1986, *PDP:14*) emphasized that E indicates how well the network satisfies the constraints that are implicit in the pattern of weights and the input to the network. They therefore adapted Hopfield's energy measure to obtain G, a measure of the *goodness of fit* of the state achieved by the network to the constraints. G is the negative of E and may also include a separate term for input if the input is continuously supplied during processing (a procedure that is referred to as *clamping* the relevant subset of units to a constant activation value).

Hopfield nets are useful for solving a variety of optimization problems. The connections literally constrain the possible stable configurations into which the network can settle. If we regard the initial pattern of activation supplied to such a network as specifying a problem and the stable state as a solution, then the connections will represent conceptual constraints on the solution and the stable state should be the state of the network that best satisfies these constraints. The traveling salesperson problem is one type of constraint satisfaction problem which has traditionally been used as a challenging case for developing optimization procedures. The salesperson needs to visit a number of cities and desires to travel the shortest distance. Hopfield and Tank (1985; see also Durbin and Willshaw, 1987) developed a modified Hopfield net which offers quite good solutions to this problem. Although it does not find the absolutely shortest route, its performance is comparable to that of a (nonnetwork) computational procedure for constrained optimization problems developed by Lin and Kernighan (1973).

One of the difficulties confronted by the Hopfield net is that it can settle into local minima (see section 3.2.1.3), in particular, situations in which there would not be sufficient net input to any given unit to get it to change its value, but in which the

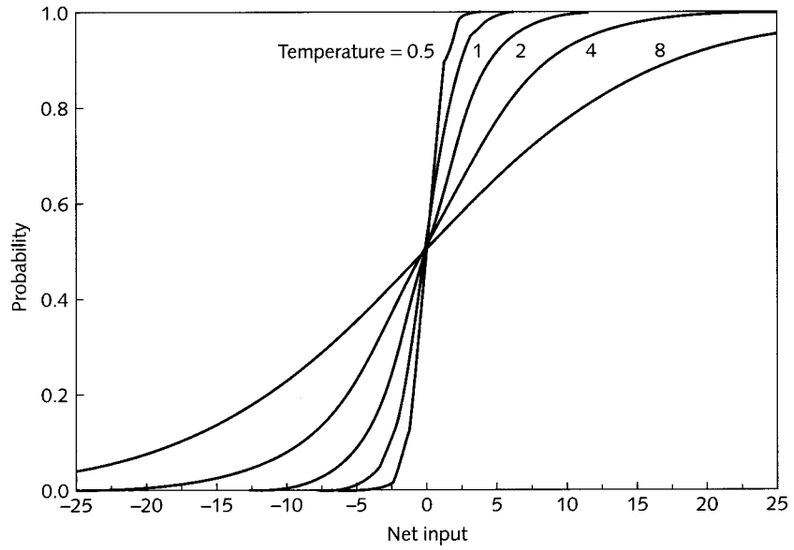

Figure 2.8 Probability that a unit takes an activation value of one as a function of net input at five different values of T (temperature). Note that θ (threshold) is zero. Reprinted with permission from D. E. Rumelhart, G. E., Hinton, and J. L. McClelland (1986) A general framework for parallel distributed processing. In D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol 1: *Foundations*. Cambridge, MA: MIT Press, p. 69.

system would still not have reached the optimal overall solution given the constraints imposed by the weights in the network. That is, the stable state is not the state that would yield the lowest possible value of E (the global energy minimum). This may result when different parts of the network have settled into incompatible solution patterns, each part of which is stable and unable to be altered by the other partial solutions. The Boltzmann machine is an adaptation of the Hopfield net which reduces this tendency.

The Boltzmann machine was proposed by Hinton and Sejnowski (1983, 1986; see also Ackley, Hinton, and Sejnowski, 1985). Like the Hopfield net, it updates its binary units by means of an asynchronous update procedure. However, it employs a *stochastic activation function* rather than a deterministic one. Specifically, it is a probabilistic version of the logistic function in equation (7). On each update of a particular unit u, the probability that it becomes active is a function of its net input:

$$probability\,(a_u = 1) = \frac{1}{1 + e^{-(netinput_u - \theta_u)/T}} \tag{9}$$

The effect of T is to alter the slope of the probability curve, as illustrated in figure 2.8 (with $\theta = 0$). When T is close to zero, the curve approaches a discontinuous step function that jumps from 0 to 1 when $netinput_u$ crosses the value 0.0 (i.e., it approximates a linear threshold unit). When T becomes very large, the curve flattens, so there is more variability in the unit's response to a given net input value across

updates. At high values of T the network will jump quickly into a solution to a new input (that is, it will require relatively few updates) but the solution is unlikely to be optimal.[6]

Equation (9) works best when a procedure called *simulated annealing* is used to vary the temperature parameter during the processing of a single input pattern. The procedure is based on an analogy from physics. Something comparable to local minima occurs in formation of crystals when incompatible sets of bonds begin to form in different parts of the crystal. If these bonds become fixed, the crystal will have a fault in it. The common way to avoid such faults is referred to as *annealing*. In this process, a material is heated, thereby weakening the bonds and allowing the atoms to reorient, and then cooled very slowly so that there is a maximal chance that as the bonds reform the atoms will orient appropriately with each other. If the cooling is carried out slowly enough around certain critical temperatures, the alignment emerging in one part of the structure has the greatest opportunity to affect that emerging elsewhere in the structure so as to develop one cohesive structure. The idea is carried over to networks by treating the patterns of activation in different parts of the network as comparable to the alignment of atoms. Raising the temperature value T has the effect of increasing the probability that activations of units in the network will shift. Reducing T very slowly at critical junctures allows time for patterns of activation developing in one part of the network to affect the patterns developing elsewhere so that one coherent pattern emerges as the network settles into a solution.

Note that the equations just presented make no reference to the prior state of activation of the unit. In contrast, the Jets and Sharks network computed a change in activation (Δa_u) which was added to u's current activation ($a_{u,t}$) to obtain u's new activation ($a_{u,t+1}$):

$$a_{u,t+1} = a_{u,t} + \Delta a_u \tag{10}$$

We left this equation implicit in the Jets and Sharks discussion, but did present two equations for Δa_u. In that case, the change in activation of a unit depended upon its current activation $a_{u,t}$ (relative to the discrepancy with its maximum or minimum activation), its net input, and the rate of decay with time. If the net input was sufficient to overcome the decay, the activation increased; otherwise it decreased. A simpler rule for capturing the same idea is:

$$\Delta a_u = k\ netinput_u - decay \tag{11}$$

where k is a constant determined by the network designer, $netinput_u$ represents the net input to the unit of interest, and *decay* represents a function that specifies an amount by which the activation of each unit will be decremented on each processing cycle. Note that this simple linear rule will not keep activations within a bounded range unless special care is taken in crafting the decay function.

2.2.2.3 Spreading activation vs. interactive connectionist models J. R. Anderson's ACT and ACT* versions of spreading activation in (nondistributed) semantic networks (Anderson, 1976, 1983; see also Anderson and Lebière, 1998) utilized nonlinear activation functions that, as in the PDP tradition, could incorporate decay and interactivity. Anderson's models could account for a variety of empirical findings, including fact retrieval (Anderson, 1974) and priming effects (e.g., McKoon and Ratcliff, 1979). An important difference, however, is that Anderson's network was

used in the service of a production system; it enabled a degree of parallel processing within the production system architecture by making a number of productions simultaneously active so as to compete with one another, and allowed a partially matching production to fire if there was no stronger competitor (see also Thibadeau, Just, and Carpenter, 1982). This is similar to the notion of *soft constraints* that is discussed in section 2.3.2 below.

Hence, the parallel propagation of activation is an idea that was used to good effect immediately prior to and into the current era of connectionist research, by J. R. Anderson and other spreading activation modelers. At least three differences distinguish spreading activation from connectionist models, however. First, Anderson retained a control structure (by means of a production system utilizing his network), whereas connectionists usually try to limit the network to its own highly decentralized local control. Second, some of the most interesting connectionist models use distinctive types of distributed representation such as coarse coding (discussed in section 2.2.4). Third, there are differences in the equations that govern the propagation of activation.

As this sketch makes clear, network designers have a variety of options available in determining both the types of activation a unit might take and how that activation is determined. The common element to all of these functions is that the new activation of a unit will be dependent in some degree on the net input a unit receives from other units. This net input is determined in part by the weights on the connections. These can be hand-tailored, but much of the interest in connectionist networks arises from their ability to modify their own weights adaptively, that is, to learn. In the next section we consider the basic idea of learning in such networks and introduce one quite simple learning procedure. Active research in this area has generated a large number of different learning procedures, however, and we will devote chapter 3 to a more detailed discussion.

2.2.3 Learning principles

For a connectionist system, learning is accomplished not by adding or modifying propositions, but rather by changing the connection weights between the simple units. Since the weights of these connections partly determine the state a network reaches as a result of its processing, these changes in weights result in changing the overall characteristics of the system. The basic goal is to provide a way of changing weights that increases the ability of the network to achieve a desired output in the future. The challenge is to have the network figure out the appropriate changes in weights without the aid of an external programmer or an internal executive in the network. Thus, the control over weight change should be entirely *local*. The information that is generally available locally to any weight is the current value of that weight and the activations of the units to which it is connected. If other information is to be employed, it must be provided to the units involved so that it is just as available as the current activations of the units.

One of the simplest such learning procedures for two-layer networks draws upon an idea proposed by Donald Hebb (1949), who suggested that learning might occur in the nervous system by strengthening the connections between two neurons whenever they fired at the same time. Expanding on this idea for connectionist networks, the strength of the connection between two units (the weight) can be increased or

decreased in proportion to the product of their activations. What is now referred to as the *Hebbian learning rule* specifies this function:

$$\Delta weight_{ui} = lrate\ a_u\ a_i \tag{12}$$

where *lrate* is a constant specifying the rate of learning, a_u is the activation of the output unit, and a_i is the activation of the input unit. Thus, whenever both units have the *same* sign (positive or negative), the connection between them is *increased* proportionally to the product of the two activations. But when the activations of the two units have *different* signs, the weight of the connection is *decreased* proportionally to the product of their activations.

To see how the Hebbian procedure can enable a network to learn, let us consider a two-layer network that is using the Hebbian rule. Assume that the activations of the output units u are determined by the simplest linear rule; that is, the output of each input unit i is identical to its activation, the activation of each output unit u is identical to its net input, and the net input to a given output unit u can therefore be obtained by simply summing the products of each i unit's activation by the weight of its connection to u:

$$a_u = \sum_i weight_{ui}\ a_i \tag{13}$$

To train the weights using the Hebbian learning rule, we supply the network with the paired input and output patterns that it is supposed to learn (we will refer to each such pair as a *case*). If the learning rate (*lrate*) is set to $1/n$, where n is the number of input units, the system will exhibit "one trial learning." That is, if it is presented with the same input again on the next trial, its weights will already be adequate to generate the appropriate output. To illustrate, suppose that a simple network with four input and four output units is presented with case A, that is, input pattern A (+ 1 + 1 − 1 − 1) paired with desired output pattern A (+ 1 − 1 − 1 + 1). If it is allowed to set its own weights according to the Hebbian principle, it will create the following matrix of 16 weights:

CASE A

Input unit	Input activation	Output unit e	f	g	h
a	+1	0.25	−0.25	−0.25	0.25
b	+1	0.25	−0.25	−0.25	0.25
c	−1	−0.25	0.25	0.25	−0.25
d	−1	−0.25	0.25	0.25	−0.25
Desired output activation		1.00	−1.00	−1.00	1.00

For example, the value of the upper left cell in the matrix was obtained by multiplying the learning rate (1/4 = 0.25) by the input value for unit **a** (1) and the output value for unit **e** (1), yielding 0.25. We can readily see that this weight matrix will enable the network to reproduce the same output pattern if we now test it with the same input pattern. To obtain the value 1 on unit **e**, for example, the input values on **a** and **b** are each multiplied by 0.25, the input values on **c** and **d** are each multiplied by − 0.25, and the four resulting values are added together. (Note that when we train

the network, the input and output activations are fixed and the weights are calculated. When we then test the network, the input activations and weights are fixed, and the output activations are calculated. Case A, above, is illustrated in training mode; the label "desired output activation" indicates that this is a fixed output pattern supplied to the network. Though shown to two decimal places, all such values are +1 or −1.)

This same network can in fact learn to produce specified output patterns for several different inputs without decreasing its performance on those it has already learned. It can do this as long as the new inputs are not correlated with those it has already learned. (Unfortunately, this is a highly constraining assumption that is difficult to satisfy in real life.) Consider input pattern B: (+ 1 − 1 − 1 + 1). We can verify that input pattern B is uncorrelated with input pattern A by calculating that, with the current weights, presenting input pattern B would cause the network to produce the output (0 0 0 0). This tells us that we could now *train* it to produce some designated output pattern B; let us specify (+ 1 + 1 − 1 − 1). This training would result in the following modified weight matrix:

CASE B

Input unit	Input activation	Output unit e	f	g	h
a	+ 1	0.50	0.00	− 0.50	0.00
b	− 1	0.00	− 0.50	0.00	0.50
c	− 1	− 0.50	0.00	0.50	0.00
d	+ 1	0.00	0.50	0.00	− 0.50
Desired output activation		1.00	1.00	− 1.00	− 1.00

With these weights, the network will still respond correctly if presented with case A again (because the new weights are an alternate solution to that problem), but now it also will respond correctly to case B.

While the Hebbian rule produces impressive results, and is often preferred by computational neuroscientists who like its biological plausibility, we will see that there are serious limits to what it can accomplish (in feedforward networks). Thus, in chapter 3 we will explore a variety of different learning procedures that are now employed in connectionist networks. The idea that links the simplest procedures (section 3.2.1) is that learning involves changing weights in a network and that this is to be accomplished using only information that is available locally, that is, at the units linked by the connection on which the weight is placed. We will also discuss the backpropagation procedure (section 3.2.2), which gains computational power by finding a way to apportion weight changes nonlocally through multiple layers.

2.2.4 Semantic interpretation of connectionist systems

In designing a connectionist network to function as a model of human performance in a particular domain, attention must be given to the question of how the concepts relevant to that domain will be represented in the network. There are two approaches: in *localist* networks each concept is assigned to one unit; in *distributed* networks the representation of each concept is distributed across multiple units. (The use of the term *local* here should not be confused with that in the preceding paragraph.)

2.2.4.1 Localist networks The Jets and Sharks network exemplifies the localist approach. Each concept (being a burglar, having the name "Art," etc.) is represented by one individual unit of the network. The semantic networks of the 1970s were also clearly localist, and when these have been augmented by equations that specify *spreading activation*, the resulting models (particularly the network portion of J. R. Anderson's 1983 ACT* theory) have been similar in many respects to localist connectionist networks. Localist networks, from whatever research tradition, share the advantage that units can be labeled in the investigator's own language to facilitate keeping tabs on what the network is doing. This carries a danger: it is easy to forget that the label conveys meaning to the investigator, but not to the network itself. The correspondence between unit and concept relies on an external process of semantic interpretation, which must be performed with care. When the network is being used as a model of a particular domain, its success will be limited by the designer's intuitions of what concepts in that domain should be encoded (i.e., associated with a unit in the network) and by his or her skill in setting up connections appropriate to that encoding.

Despite these demands on the designer, it is generally even more difficult to set up a distributed network. Therefore, a localist network may be preferred when the task does not require the distinct advantages of distributed encoding (which are discussed below). The task of *multiple constraint satisfaction*, for example, can be adequately performed by an interactive localist network. Units represent concepts, and positive or negative connections between pairs of units represent constraints between those concepts. A positive connection between two units puts a constraint on the entire network to favor states (overall activation patterns) in which the two units have the same activation (e.g., both *on* or both *off*); a negative connection favors states in which the two units have opposite activity. To run this procedure, each unit is given a baseline activation value (often zero), and each connection weight is fixed as positive or negative. External input is optional, and acts as an additional source of constraint. The network then runs interactively until it settles into a stable state (pattern of activation values); if a global energy minimum is attained, that state will be the state that best satisfies the constraints. One of the important features of this procedure is that these are *soft* constraints. Even if there is a negative constraint between two units, if other constraints involving those units favor their having the same state, the most stable solution may have both units *on* or both units *off* (violating that one particular constraint but satisfying a number of other constraints).

2.2.4.2 Distributed networks In a distributed network, each concept is represented by a pattern of activation across an ensemble or set of units; by design, no single unit can convey that concept on its own. To convert the Jets' and Sharks' three occupations to distributed representations, for example, we might continue to use three units, but arbitrarily associate each occupation with a different pattern of activation values across those units as follows:

+ 1 + 1 + 1 = Burglar − 1 − 1 + 1 = Bookie + 1 − 1 − 1 = Pusher

In this example individual units do not have a semantic interpretation. Rather, each semantic interpretation is distributed over three units (rather than assigned to one localist unit), and each unit is involved in three different semantic interpretations.

An alternative way to achieve a distributed representation of a concept is to carry out a featural analysis of the concept, and encode that analysis across an appropriate

number of units. Featural representation is widely used in the psychological literature on concepts and categories, for example in exemplar models (Medin and Schaffer, 1978). What is new here is to situate the featural representation in a network architecture. The idea that this constitutes a distributed representation is often difficult to grasp, because the features themselves are typically encoded on individual units. That is, there is a localist representation of the features (one unit per feature), and a distributed representation of the target concept. In illustration, suppose that a large number of occupations are the objects to be represented. The network designer might craft a set of 50 units in which each unit is interpreted as corresponding to some feature that is salient to occupations, and represent each occupation as a pattern across those units. Those occupations chosen by the Jets and Sharks would all share the feature **illegal**, for example, but only the pusher occupation (like the pharmacist occupation) would have the feature **involves drugs**.

How does one obtain a featural analysis of a domain in order to build these representations? One method is to let an existing theory guide this work; for example, a particular linguistic account might be used as a basis for representing words as patterns over phonemic units. In the connectionist research program the interest lies, not so much in the particular features and assignments, but rather in how the system makes use of its distributed representation once it has been built. We might, for example, explore generalization or associative learning under the conditions offered by such a representation. A second method is to let the system perform its own analysis of the domain. When a multi-layered network is run in a learning paradigm (as described in section 3.2.2), the network designer specifies the interpretations only of the input and output units; the input–output cases used for training are selected with respect to this interpretation. The designer does not know what aspects of the input–output cases each hidden unit will become sensitive to; the learning process is in part a process of feature extraction, and observing this is one of the most intriguing aspects of connectionist research. Usually the hidden units do not arrive at a simple, localist representation of the most obvious regularities (features) in the input. Rather, each hidden unit is sensitive to complex, often subtle, regularities that connectionists call *microfeatures*. Each layer of hidden units can be regarded as providing a particular distributed encoding of the input pattern (that is, an encoding in terms of a pattern of microfeatures).

One virtue of distributed representation is that part of the representation can be missing without substantially hurting performance. Using the arbitrary distributed representations of three occupations displayed at the beginning of this section, for example, we might specify an input only for the first two units and leave the third questionable (+ 1 + 1 ?). Nevertheless, the network will treat this pattern similarly to a complete burglar input (+ 1 + 1 + 1), since this is the pattern to which the partial input is most similar. The point is even clearer in networks with more units. Consider, for example, a pattern associator network with two layers of 16 units each. Each of the units in the input will be connected to all of the units in the output layer, and depending upon the size of the weights, will contribute either a little or a lot of the information needed to determine the value of the output unit. Each output unit will be receiving inputs from 16 different input units. If there is no input from a given input unit, or the wrong input, the activity on the other input lines generally will compensate. The point here is that once we distribute information as a pattern across units, we also distribute the resources used to process it. Thus, a distributed system becomes more resilient to damage. Beyond this, we make it possible for the

system to learn new information without sacrificing existing information. For example, without adding new units, we can teach the network to respond to a new input that is different from any it has learned so far. It often is sufficient to make slight changes to a variety of weights, which do not significantly alter the way the network responds to existing patterns. (There are limits to this capacity, however; in some circumstances teaching a new input disrupts previous learning to an unacceptable degree. For a demonstration of *catastrophic interference*, see McCloskey and Cohen (1989); for connectionist responses, see Hetherington and Seidenberg (1989), and French (1992). In subsequent chapters we identify a number of contexts in which catastrophic interference seems to arise, and a number of connectionist attempts to overcome the problem.)

For a final virtue of distributed networks, consider a distributed network that has encoded a structured domain of knowledge, in which there are regularities in the input–output pairings. If the network is presented with a new input pattern, a reasonable response should appear on the output units. Humans often exhibit a similar capacity when provided with only partial information about a new entity. Generally, we infer some of the other properties the entity might have, based on its similarity to entities we already know. In their discussion of distributed representations, Hinton, McClelland, and Rumelhart (1986, *PDP:3*) provide the following example: if we are told that chimpanzees like onions, we will probably revise our expectations about whether gorillas like onions as well. This reflects a general tendency of organisms to make *generalizations* (and specifically exemplifies what Shipley (1988) calls an *overhypothesis*). Distributed representations are well suited to producing and investigating generalization.

There is one additional approach to achieving distributed representations that is counterintuitive in many respects, but is ingenious and exhibits some very useful properties. This is a technique known as *coarse coding*. The basic idea is that, rather than deploying units so that each unit represents information as precisely as possible, we design each individual unit (called a *receptor* in this context) to be sensitive to many different inputs (which constitute its *receptive field*). Each unit is sensitive to (activated by) a different set of inputs, and each input has a number of units that are sensitive to it. In this architecture, the fact that a particular unit is active is not very informative; but if a high percentage of the units that are sensitive to a particular input are active, the presence of that input can be inferred with high confidence.

To consider a concrete example, coarse coding was used by Touretzky and Hinton (1988) in a connectionist system that they constructed to implement a production system. One of their purposes was to show that connectionist systems can indeed represent and use explicit rules; for discussion of this aspect of their paper, see section 6.2. Their other purpose was to illustrate certain advantages of coarse coding. The production system that they implemented was designed to follow rules involving meaningless triples composed from a 25-letter vocabulary. The triples themselves were not directly encoded in the network; rather, each coarse-coded triple was presented by turning on all of the units (28 on average) that were designated as its receptors in the *working memory* network that was one component of the system. Each such unit was a receptor for a large number of different triples. For example, one of the units that was a receptor for the triple (F A B) was also a receptor for any other triple that could be formed from the following receptive field by selecting one letter from each column in the order shown:

Position 1	Position 2	Position 3
C	A	B
F	E	D
M	H	J
Q	K	M
S	T	P
W	Y	R

Hence that particular receptor unit was turned on (became active) if (F A B) was to be presented; but the same unit would be turned on if (C A B) or (S H M) or any one of 216 (6^3) different triples was to be presented. The particular sets of letters were determined randomly and were different for each unit. For example, another of the receptor units for (F A B) might have had the following receptive field:

Position 1	Position 2	Position 3
F	A	B
I	C	D
K	H	E
P	K	J
S	R	M
V	W	U

From the perspective of the triple itself, those two receptor units and their distinctive receptive fields were only two of the approximately 28 different units that would be activated as the means of presenting (F A B) to the working memory network. From the perspective of a single receptor unit, it might have been turned on as a means of presenting any one of the 216 different triples in its receptive field. To know which triple was actually being presented, one would need to know the particular combination of 28 activated units and to consult the external listing of their receptive fields (which would reveal that they had in common only that F was one of the six letters in position 1, A was one of the six letters in position 2, and B was one of the six letters in position 3).

This may seem like a very strange way to design a memory, but Touretzky and Hinton pointed out several advantageous or human-like properties gained from coarse coding. First, the number of units needed to store all possible triples is minimized. There are 15,625 possible triples, about half a dozen of which are present in working memory at any given time. But the working memory is composed of only about 2,000 different units. Second, the memory is tolerant of noise (i.e., a few units can be in the wrong state without materially affecting performance). Third, the memory does not have a rigid, fixed capacity; rather, its ability to distinguish triples will gradually decline as the number of stored items increases. Fourth, active triples will gradually decay as new triples are stored. Fifth, a degree of generalization is exhibited: if two of the triples that have been presented to the memory network both happen to have F as their initial letter, more than the usual number of receptors will be active for other F-initial triples.

Coarse coding is one of the most distinctive, nonobvious techniques made possible by the use of distributed (rather than localist) representations. Hinton, McClelland, and Rumelhart (1986, *PDP:3*) provide further discussion of coarse coding, including design considerations that must be attended to (e.g., a tradeoff between resolution and accuracy in setting the size of the receptive fields). They also make the following intriguing suggestion, with which we will close this section:

> Units that respond to complex features in retinotopic maps in visual cortex often have fairly large receptive fields. This is often interpreted as the first step on the way to a translation invariant representation. However, it may be that the function of the large fields is not to achieve translation invariance but to pinpoint accurately where the feature is! (1986, *PDP:3*, p. 92)

2.3 The Allure of the Connectionist Approach

One reason many people are attracted to network models of cognition is that they seem to exhibit many properties found in human cognition that are not generally found in symbolic models. In this section we will review, without much critical discussion, some of the properties that have been cited.

2.3.1 Neural plausibility

Certainly one of the major features that has attracted researchers to network models is that they seem more compatible than symbolic models with what we know of the nervous system. This is not surprising: network models are *neurally inspired*. Pitts and McCulloch, for example, built their models using a simplified conception of how neurons work. Hence, the state of activation of a unit (especially of units that only acquire discrete activations, 0 and 1) was intended to correspond to a neuron either resting or firing. The connections between units were conceived on the model of the axons and dendrites of neurons. Thus, the propagation of activation within a network is, at least on this very general level, similar to the kinds of processing that we observe in the nervous system.

Of course, connectionist networks do not capture all features of neural architecture and processing in the brain. For example, little attention is paid to trying to model the particular pattern of connectivity of neurons in the brain. Nor is there any attempt to simulate the differences between various neurotransmitters, or the very intricate way in which excitations to neurons are compounded to determine whether the neuron will actually fire. Thus, networks only capture aspects of the coarse architecture of the brain. Conversely, there are aspects of connectionist networks that do not clearly map on to what is known about the nervous system; the back-propagation procedure for learning (section 3.2.2) is a particularly important example.

These differences present no difficulty, and in fact may be desirable, if one focuses on connectionist models as cognitive (rather than biological) models. Some investigators, however, prefer to push the neural analogy as far as possible. For example, it is known that neural systems carry out basic processing (e.g., recognizing a word) very quickly, that is, within a few hundred milliseconds. Since it takes each neuron several milliseconds to fire, it has been argued that basic cognitive tasks cannot require more than a hundred steps of sequential processing (Feldman and Ballard, 1982). This, it is claimed, poses a serious problem for traditional symbolic architectures, since to model even simple tasks often requires programs embodying several thousand instructions. But since network processing relies on performing many different operations in parallel, it seems much easier for networks to satisfy the 100-step constraint. For example, the network described in section 2.1 could identify a gang member on the basis of some of his properties and then determine

can be interpreted as serving the same function that a rule might serve in a symbolic system. But even when we find such an interpretable connection, it is only a soft constraint that the system as a whole might override. Hence, connectionism attempts to avoid some of the problems posed by exceptions to rules by using a system of *soft* constraints rather than *hard* rules. The goal is to account for exceptions as well as regularities within the same system. (See chapter 5 for opposing views of the success of one such attempt.)

2.3.3 Graceful degradation

One of the notable features of the human brain is that it seems to be an extremely reliable device. Like any mechanism, though, it has its limits. It can be overloaded with too many demands or too much information, or it can be impaired by physical damage. But when its limits are exceeded, generally it does not crash. It simply begins to perform sub-optimally. When confronting a task that makes too many demands, it begins to ignore some of the demands or some of the information. The more it is overloaded, or the more it is impaired, the less well it functions. The same gradation of effect is found when some components are destroyed. In a very few situations, a clearly delineated behavioral deficit will arise. For example, the classical work on aphasia correlated particular lesions (e.g., in Broca's area) with particular behavioral deficits (inability to produce articulate speech). But in general the brain is rather resilient in the face of damage. Nerve cells die every day, but generally this does not leave a trace in terms of specific impairment of performance. Loss of even large numbers of neurons may lead not to specific losses but to a nonspecific gradual impairment of function. For example, we do not forget how to divide 12 by 2 and yet remember the rest of the division tables; rather, we gradually become more limited in our numeric abilities. This characteristic of gradually failing performance is generally referred to as *graceful degradation*.

A traditional symbolic system does not exhibit graceful degradation. If any of its elements are lost, the information they encode is no longer available to the system. This is particularly clear if we consider what happens if a rule is eliminated. The system is simply not able to respond to any of the situations in which that rule was needed. It is possible to develop implementations of symbolic systems that are more resistant to damage, for example, by storing information at redundant locations or using error-checking techniques to recover from damage. Such an implementation still may fail to exhibit the more subtle phenomenon of graceful degradation.

A connectionist network, on the other hand, does exhibit graceful degradation. Destruction of a few connections or even of a few units (except in networks which only have a few units to begin with) generally does not significantly impair the activity of the system. In a localist system, destroying a unit will destroy a particular piece of that system's information, with possibly serious consequences, but destroying connections instead will result in graceful degradation. For example, using the Jets and Sharks network which employs a localist encoding, we destroyed at random 53 of the 1,062 connections in the network (5 percent of the total) and then explored its performance on the same tasks that we discussed at the beginning of this chapter. On two of the tasks its performance was qualitatively the same: it still correctly identified Art's properties and it still correctly identified the individuals that met the specification of being in their twenties and being pushers. It did perform differently

when queried about Sharks. It still responded by activating particular Sharks, but it offered different judgments as to who were the most prototypical Sharks (Nick, Neal, and Dave). The reason the network was still able to identify Art's properties was that none of the connections between Art's person unit and his properties happened to be broken. But as an additional experiment we broke one of these connections (between **ART** and **junior high** education). The network now answered incorrectly that Art had a **high school** education. What is interesting is how it arrived at this answer: those individuals who were most similar to Art in other respects tended to activate the unit for high school education. Thus, even when disabled, the network still offered plausible judgments. It did not crash.

In systems using distributed representation, we can eliminate a number of units in the system and the system will still behave in only a slightly distorted fashion. For example, if an input normally consists of a distributed representation over eight units, and one of these is disabled, the system will still respond normally to most input patterns. With more damage, the system will increasingly make errors; however, even these will not be random, but rather be associated with closely related patterns to which the distorted input is now more similar. Hence, a connectionist cognitive system inherently displays graceful degradation as a consequence of its own architecture. It will display that property whether it is implemented in a nervous system, on a parallel machine, or even on a serial computer.

2.3.4 Content-addressable memory

The human ability to remember information is quite remarkable. Frequently information that we need comes to mind spontaneously. We identify a book that we need, and we remember that we loaned it to a student. Sometimes, though, we need to work at recalling information: we remember we loaned the book to a student, but now have to work at trying to recall who the student was. This may involve retrieving cues that will help us identify the person. Typically, we can retrieve the same piece of information from a variety of different cues that constitute part of the contents of the memory itself. Since such memory is accessed through its content, it is generally termed *content-addressable memory*. Designing this type of memory access into a symbolic system is a challenge, and requires maneuvering around the architecture of the system rather than taking advantage of it.

A common model for a symbolic memory is a filing system: we store information on paper, place the paper in a file folder, and position the folder in a cabinet sorted according to some procedure we take to be reasonable. If each folder is positioned by an arbitrary index, or by the serial order of its creation, there is no content addressability in the filing system. More frequently, each folder will be positioned in accord with one or at most two aspects of its content. For example, suppose that we keep track of students in our classes by placing information about them in file folders arranged alphabetically. Sometimes this works very well. If we want information on a particular student, we can rapidly access the file with that student's name. If, however, we seek to recover the information by taking a different route, the task is more difficult. Suppose that the information we want to access about a student is her name, and the cues we start with are what class she took and what grade she got. Now we face a serious problem: since the information is not organized in this manner, the only way to retrieve the student's name is to go through each

folder until we find a student who took the class in question and received the specified grade. If we had known in advance that there might be different ways in which we would want to access the information in our filing cabinet, we could have developed an indexing system that would have told us where information satisfying certain descriptions would be found. For example, we might have constructed an index identifying by name the students in each class. But then it is necessary to identify in advance all the ways we might want to access the file. Furthermore, if we make errors in recalling the contents that are indexed (e.g., confusing our course on research methods with our course on statistics), the index is of little or no use.

The disadvantages of the filing cabinet system are exhibited in a variety of memory systems. In computer systems, for example, information is stored at register locations, and the only way to access information directly is by means of the address of the location. Symbolic systems that are implemented on such computers often (although not necessarily) make some of the same assumptions about storage and retrieval. Serial search through separate items therefore figures prominently in memory retrieval. Some such systems attain superior performance by means of intelligent search procedures that mitigate this difficulty.

Connectionist networks offer a relatively natural alternative means of achieving content-addressable, fault-tolerant memory. The Jets and Sharks network provides a simple illustration. Properties could be retrieved from names, names from properties, and so forth. We might even make a mistake on one property and still retrieve the right person. For example, we gave the network the task of remembering George's name and we described him as a Jet, in his thirties, junior high educated, and divorced. As an experiment, we deliberately made a mistake about one of George's properties (he is in fact still in his twenties). No one, in fact, precisely fits this description. But, since the connections only constitute soft constraints, the network proceeds to find the best match. The units for Jim and George become most active (0.31 after 70 cycles), while Al is slightly less active (0.30). Jim and George actually have identical properties and match on three out of four cued properties, while Al has different properties, but also matches on three out of four. Thus, even with erroneous cues, the network has recalled the persons who best match what cues were given.

The advantages of content-addressable memory are particularly evident in systems employing distributed representations; in such systems it is often possible, given part of a pattern, to reconstruct the whole pattern. A question arises, however, as to how we should characterize this sort of memory. Within symbolic systems remembering is a process of retrieving a symbol that has been stored away. But in connectionist networks, remembering is carried out by the same means as making inferences; the system fills in missing pieces of information. As far as the system's processing is concerned, there is no difference between reconstructing a previous state, and constructing a totally new state (confabulating):

> One way of thinking about distributed memories is in terms of a very large set of plausible inference rules. Each active unit represents a "microfeature" of an item, and the connection strengths stand for plausible "microinferences" between microfeatures. Any particular pattern of activity of the units will satisfy some of the microinferences and violate others. A stable pattern of activity is one that violates the plausible microinferences less than any of the neighboring patterns. A new stable pattern can be created by changing the inference rules so that the new pattern violates them less than its neighbors. This view of memory makes it clear that there is no sharp distinction between genuine memory and plausible reconstruction. A genuine memory is a pattern

> that is stable because the inference rules were modified when it occurred before. A "confabulation" is a pattern that is stable because of the way the inference rules have been modified to store several different previous patterns. So far as the subject is concerned, this may be indistinguishable from the real thing. (Hinton, McClelland, and Rumelhart, 1986, *PDP:3*, pp. 80–1)

2.3.5 Capacity to learn from experience and generalize

A final feature of networks that makes them attractive is their capacity to learn from experience by changing the weights of connections. In addition to the Hebbian approach introduced in section 2.2.3, network researchers have developed a variety of procedures for gradually adjusting the weights of a network so that, if an adequate set of weights exists, the network will find them. One advantage of these approaches is that they generally allow networks to generalize beyond the training sets to give correct responses to new inputs. These learning procedures are the focus of the next chapter. They give connectionism a resource that may enable it to explain such things as learning a language or learning to do arithmetic, kinds of learning which are awkward to model symbolically.

2.4 Challenges Facing Connectionist Networks

In the previous section we have examined some of the touted advantages of connectionist networks. But there are, as well, some well-known challenges facing connectionist researchers. Some of these, such as catastrophic interference, we have already noted. Others, such as accounting for the productivity and systematicity of thought, are discussed in detail in subsequent chapters. A few others, though, should be noted here. One is that there are a large number of parameters connectionist researchers can manipulate, explicitly or implicitly, in setting up their networks. These reflect numerous decisions involving the architecture of the network, the activation and learning functions, and the means of encoding tasks to be performed. While manipulating this range of parameters may enable connectionist modelers to account for a wide diversity of cognitive performances, it also represents a potential weakness of connectionist modeling. If a given simulation succeeds in accounting for behavior with a fortuitous set of parameters, this may not be very informative as to how humans generate the behavior.

Of particular note in this respect is the manner in which researchers encode the inputs and desired outputs for their simulations. By carefully crafting these, a researcher may simplify the challenge to the network and the resulting simulation may not significantly advance our understanding of human performance. Even if the simulation accurately characterized human performance, our explanation of the performance would then require us to determine how it is that the inputs and desired outputs for the cognitive system came to have that form. A greater risk is that the encoding used by humans is very different from that supplied by the researcher to the network and as a result the simulation does not correspond to the way humans perform the task.

A final challenge facing connectionism concerns learning. As we will see in the next chapter, to overcome the limitations with Hebbian learning, connectionists

frequently adopt a form of gradient descent learning in which a repetitive process of changing weights ultimately enables networks to perform their task. But not all learning is so gradual. At least in the case of human beings, some information can be learned rapidly in one or two encounters. Also, some information is encoded in relative isolation rather than as part of a highly connected system. In illustration of both of these points, suppose that I am told verbally, "To make this thing work, push a candy bar through the slot in the center." I am highly likely to remember that rather bizarre instruction. This kind of learning is relatively easy to model in symbolic models, since it only requires encoding a new rule, but not in networks that learn by gradient descent. The Hebbian procedure we introduced in this chapter provides one way networks can achieve one-trial acquisition of idiosyncratic responses. There are other approaches to connectionist learning, such as Kohonen's procedure for self-organizing feature maps which we discuss in section 7.5, that enable one-trial learning of distinctive information. However, it is not obvious how to integrate these approaches with the more frequent use of gradual learning. Since humans are capable of both sorts of learning, one would hope that a unified account will eventually be attained.

2.5 Summary

In this chapter we have presented a simple connectionist network (the Jets and Sharks network), and examined some of the basic architectural features that can be employed in connectionist networks more generally. We have also examined some of the features of connectionist systems that have served to attract interest in them. In the next chapter we will examine in more detail the ability of connectionist systems to learn; then in chapter 4 we will turn to a cognitive task, pattern recognition, for which connectionist networks appear particularly adept.

Notes

1 Usually the word "cluster" is used for sets of items that are similar in some way, whereas here the items in each cluster form a contrast set.
2 In fact, as processing continues, the activation of the person unit **MIKE** begins to drop again. The reason is that as **ART** and the person units for gang members most similar to Art grow in activation, they send increasingly inhibitory inputs to **MIKE**.
3 The activation pattern across a layer of n units can be treated as a vector (directed line segment) in an n-dimensional space.
4 The hidden units did not receive external input, so that term would always have a zero value for those units.
5 Hopfield's E should not be confused with the measure of mean squared error that is used in deriving the delta rule. In boxes 1 and 2 of chapter 3 we call this measure *Error*, but it is often called E.
6 When equation (9) is applied to a feedforward network, temperature does not affect the time to reach a solution (because each output unit's activation is calculated just one time). If a learning rule is also being applied, however, high variability of response across different presentations of the same input will make learning slower (which often is desirable).

Sources and Recommended Readings

Anderson, J. A. (1995) *An Introduction to Neural Networks*. Cambridge, MA: MIT Press
Ballard, D. H. (1997) *An Introduction to Natural Computation*. Cambridge, MA: MIT Press.
Ellis, R. and Humphreys, G. (1999) *Connectionist Psychology: A Text with Readings*. East Sussex: Psychology Press.
Levine, D. S. (1991) *Introduction to Neural and Cognitive Modeling*. Hillsdale, NJ: Erlbaum.
McClelland, J. L. and Rumelhart, D. E. (1988) *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*. Cambridge, MA: MIT Press. (Includes **PDP** simulation software on disk.)
McClelland, J. L., Rumelhart, D. E. and the PDP Research Group (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 2: *Psychological and Biological Models*. Cambridge, MA: MIT Press.
McLeod, P., Plunkett, K. and Rolls, E. (1998) *Introduction to Connectionist Modelling of Cognitive Processes*. Oxford: Oxford University Press. (Includes **tlearn** simulation software on disk; also downloadable from crl.ucsd.edu/innate/tlearn.html.)
O'Reilly, R. C. and Munakata, Y. (2000) *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*. Cambridge, MA: MIT Press. (Keyed to **PDP++** simulation software, downloadable from www.cnbc.cmv.edu/PDP++/PDP++.html.)
Plunkett, K. and Elman, J. L. (1997) *Exercises in Rethinking Innateness*. Cambridge, MA: MIT Press. (Includes **tlearn** simulation software on disk; also downloadable from crl.ucsd.edu/innate/tlearn.html.)
Rumelhart, D. E., McClelland, J. L. and the PDP Research Group (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1: *Foundations*. Cambridge, MA: MIT Press.