3 Computational Overview

1 INTRODUCTION

What is computation? In virtue of what is something a computer? Why do we say a slide rule is a computer but an egg beater is not? These are, in a way, the philosophical questions of computer science, inasmuch as they query foundational issues that are typically glossed over as researchers get on with their projects.[1] Like the philosophical questions of other disciplines (What is the nature of life? [Biology] What is the nature of substance and change? [Physics and Chemistry]), the answers become more convincing, meaningful, and interconnected as the empirical discipline matures and gives more ballast to the theory. In advance of understanding that there are atoms, how atoms link together, and what their properties are, one simply cannot say a whole lot about the nature of substance and change. It is not, however, that one must say *nothing*—in that event, one could not get the science started. The point rather is that the theory outlining the elementary ideas of the discipline gradually bootstraps itself up, using empirical discoveries as support, and kicking away old misconceptions in the haul.

The definition of computation is no more *given* to us than were the definitions of light, temperature, or force field. While some rough-hewn things can, of course, be said, and usefully said, at this stage, precision and completeness cannot be expected. And that is essentially because there is a lot we do not yet know about computation. Notice in particular that once we understand more about what sort of computers *nervous systems* are, and how they do whatever it is they do, we shall have an enlarged and deeper understanding of what it is to compute and represent. Notice also that we are not starting from ground zero. Earlier work, especially by Turing (1937, 1950), von Neumann (1951, 1952), Rosenblatt (1961), and McCulloch and Pitts (1943), made important advances in the theory and science of computation. The technological development of serial, digital computers and clever software to run on them was accompanied by productive theoretical inquiry into what sort of business computation is.[2]

Agreeing that precise definitions are not forthcoming, can we nonetheless give rough and ready answers to the opening questions? First, although we may be guided by the example of a serial digital computer, the notion of "computer" is broader than that. Identifying computers with *serial digital* com-

puters is neither justified nor edifying, and a more insightful strategy will be to see the conventional digital computer as only a special instance, not as the defining archetype. Second, in the most general sense, we can consider a physical system as a computational system when its physical states can be seen as representing states of some other systems, where transitions between its states can be explained as operations on the representations. The simplest way to think of this is in terms of a mapping between the system's states and the states of whatever is represented. That is, the physical system is a computational system just in case there is an appropriate (revealing) mapping between the system's physical states and the elements of the function computed. This "simple" proposal needs quite a lot of unpacking.

Functions: Computable or Noncomputable, Linear or Nonlinear

Since this hypothesis concerning what makes a physical system a computational system may not be self-evident, let us approach the issue more gradually by first introducing several key but simple mathematical concepts, including "function," and the distinction between *computable* and *noncomputable* functions. To begin, what is a function? A function in the mathematical sense is essentially just a mapping, either 1 : 1 or many : 1, between the elements of one set, called the "domain," and the elements of another, usually referred to as the "range"[3] (figure 3.1). Consequently, a function is a set of ordered pairs, where the first member of the pair is drawn from the domain, and the second element is drawn from the range. A computable function then is a mapping that can be specified in terms of some *rule* or other, and is generally characterized in terms of what you have to do to the first element to get the second. For example, multiply the first by 2, $\{(1, 2), (2, 4), (3, 6)\}$, expressible algebraically as $y = 2x$; multiply the element from the domain by itself $\{(6.2, 38.44), (9.6, 92.16)\}$, expressible algebraically as $y = x^2$, and so on.

What then is a noncomputable function? It is an infinite set of ordered pairs for which no rule can be provided, not only now, but in principle. Hence its specification consists simply and exactly in the list of ordered pairs. For example, if the elements are randomly associated, then no rule exists to specify the mapping between elements of the domain and elements of the range. Outside of mathematics, people quite reasonably tend to equate "function" with "computable function," and hence to consider a nonrule mapping to be no function at all. But this is not in fact how mathematicians use the terms, and for good reason, since it is useful to have the notion of a noncomputable function to describe certain mappings. Moreover, it is useful for the issue at hand because it is an empirical question whether brain activity can really be characterized by a computable function or only to a first approximation, or perhaps whether some of its activities cannot be characterized at all in terms of computable functions (Penrose 1989).

What is a *linear* function? Intuitively, it is one where the plot of the elements of the ordered pair yields a straight line. A *nonlinear* function is one where the plot does not yield a straight line (figure 3.2). Thus when brain function is

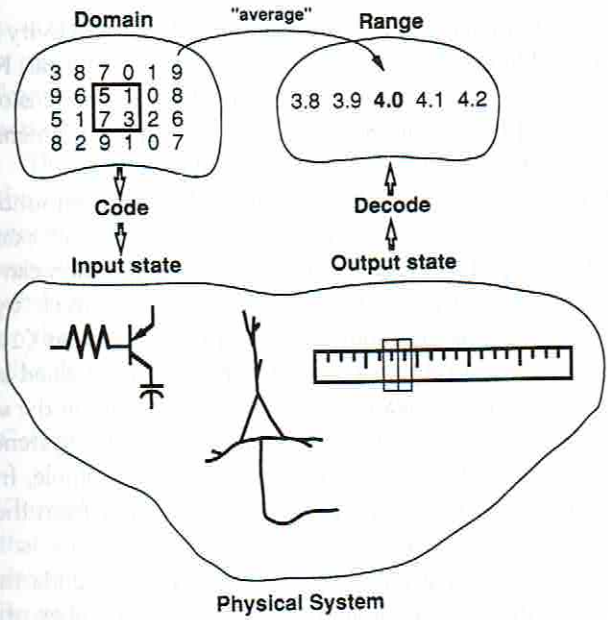

Figure 3.1 Mapping between a domain and a range can be accomplished by a variety of physical systems. There are three steps: (1) The input data is coded into a form appropriate for the physical system (electrical signal in an electrical circuit, chemical concentration in neuron, position of a slider in a slide rule). (2) The physical system shifts into a new state. (3) The output state of the physical system is decoded to produce the result of the mapping. The example shown here is the "average" map that takes four values and produces their average. Such a mapping might be useful as part of a visual system. Mappings could also be made from the domain of temporal sequences, and the range could be a sequence of output values.

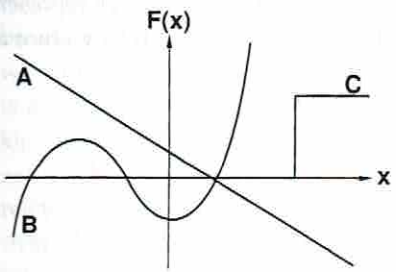

Figure 3.2 Examples of functions $F(x)$, plotted along the vertical axis, of one variable, x, plotted along the horizontal axis. Function A is a linear function. Function B is a nonlinear function. Function C is a discontinuous function.

described as "nonlinear," what this means is that (a) the activity is characterized by a computable function, and (b) that function is nonlinear. Notice also that the space in which functions are plotted may be a two-dimensional space (the x and y axes), but it may, of course, have more than two dimensions (e.g., an x axis, y axis, and also w, v, z, etc. axes).

Because the notion of a *vector* simplifies discussion enormously, we introduce it here. A vector is just an ordered set of numbers. For example, the set of incomes for 1990 of three vice-presidents in a corporation can be represented by the vector $\langle \$30, \$10, \$10 \rangle$; the eggs laid per week by five hens as $\langle 4, 6, 1, 0, 7 \rangle$; the spiking frequency of four neurons/sec as $\langle 10, 55, 44, 6 \rangle$. By contrast, a scalar is a single value rather than a many-valued set. The *order* in the set matters when we want to operate on the values in the set according to an order-sensitive rule. Systems, including the nervous system, execute functions that perform vector-to-vector mapping. For example, from the stretch receptors' values to the muscle contraction values, or from the head velocity values to eye velocity values.

A geometric articulation of these concepts compounds their value. Any coordinate system defines a state space, and the number of axes will be a function of the number of dimensions included. A state space is the set of all possible vectors. For example, a patient's body temperature and diastolic blood pressure can be represented as a position in a 2-D state space. Or, if a network has three units, each unit may be considered to define an axis in a 3-D space. The activity of a unit at a time is a point along its axis, so that the global activation of all the units in the net is specified by a point in that 3-D space (figure 3.3). More generally, if a network has n units, then it defines an n-dimensional activation space, and an activation vector can be represented as a point in that state space. A sequence of vectors can be represented as a trajectory in the state space.[4] Thus the patient's body temperature and blood pres-

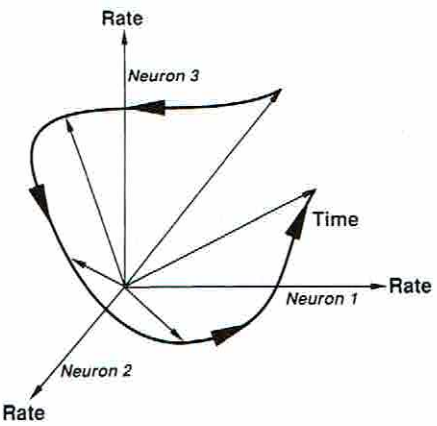

Figure 3.3 Schematic diagram of the trajectory of a three-neuron system through state space. The state of the system is a 3-*D* vector whose components are the firing rates of the three neurons. As the firing rates change with time, the tip of the vector traces out a trajectory (thick line). For more neurons the state space will have a higher dimension.

sure followed through time results in a trajectory in a 2-space. A function maps a point in one state space to a point in another state space—for example, from a point in stretch-receptor activation space to a point in muscle spindle activation space.

These notions—"vector" and "state space"—are part of linear algebra, and they are really the core of the mathematics needed to understand model networks. They are mercifully simple conceptually, and they are rather intuitively extendable from easily visualizable 2-D cases to very complex, n-D cases, where n may be thousands or millions. Although volumes more can be written on the topic of linear algebra, this is perhaps enough to ease the entry into the discussion of model neural networks.[5]

Computers, Pseudocomputers, and Cryptocomputers

The mathematical interlude was intended to provide a common vocabulary so that we might return to the question of characterizing, albeit roughly, what about a physical system makes it a computer. To pick up the thread left hanging during the mathematical interlude, let us hypothesize that a physical system computes some function **f** when (1) there is a systematic mapping from states of the system onto the arguments and values of **f**, and (2) the sequence of intermediate states executes an algorithm for the function.[6] Informally, an algorithm is a finite, deterministic procedure, e.g., a recipe for making gingerbread or a rule for finding the square root.

We count something as a computer because, and only when, its inputs and outputs can usefully and systematically be interpreted as representing the ordered pairs of some function that interests us. Thus there are two components to this criterion: (1) the objective matter of what function(s) describe the behavior of the system, and (2) the subjective and practical matter of whether we care what the function is. This means that delimiting the class of computers is not a sheerly empirical matter, and hence that "computer" is not a natural kind, in the way that, for example, "electron" or "protein" or "mammal" is a natural kind. For categories that do delimit natural kinds, experiments are relevant in deciding whether an item really belongs to the category. Moreover, there are generalizations and laws (natural laws) about the items in the categories and there are theories interleaving the laws. Nonnatural kinds differ in all these respects, and typically have an interest-relative dimension.

"Bee," for example, is a natural kind, but "gem" and "weed" are not. Objects are considered gems depending on whether some social group puts special value on them, typically as status symbols. Plants are considered weeds depending on whether gardeners (serious gardeners?) in the region happen to like having them in the garden. Some gardeners cultivate baby's breath as a desirable plant; other gardeners fight it as a weed. There is no experiment that will determine whether baby's breath is really a weed or not, because there is no fact of the matter—only social or idiosyncratic conventions.[7] Similarly, we suggest, there is no instrinsic property necessary and sufficient for all computers, just the interest-relative property that someone sees value in interpreting

a system's states as representing states of some other system, and the properties of the system support such an interpretation. Desk-top von Neumann machines exist precisely because we are keenly interested in the functions we build and program them to execute, so the interest-relative component is dyed in the wool. For this reason, and because these machines are so common, they are the prototypical computers, just as dandelions are prototypical weeds. These prototypes should not, however, be mistaken for the category itself.

It may be suggested as a criticism of this very general characterization of computation that it is *too* general. For in this very wide sense, even a sieve or a threshing machine could be considered a computer, since they sort their inputs into types, and if one wanted to spend the time at it, one could discover a function that describes the input–output behavior. While this observation is correct, it is not so much a criticism as an apt appreciation of the breadth of the notion. It is rather like a lawn-growing perfectionist incredulously pointing out that on our understanding of "weed," even dandelions might be nonweeds relative to some clime and some tribe of growers. And so, indeed, they might be some farmer's cash crop. Nor is this idle fancy. Cultivated dandelion greens now appear as a delicacy in the specialty section of the greengrocery.

Conceivably, sieves and threshing machines could be construed as computers if anyone has reason to care about the specific function reflected in their input–output behavior, though it is hard to see what those reasons might be (figure 3.4). Unlike desktop computers that are engineered precisely for their computational prowess, sieves and threshing machines are constructed for other reasons, namely their sheerly mechanical prowess in the sorting of objects according to size and shape. Not too much emphasis should be placed on the link between purposeful design and use as a computer, however, for a fortuitously shaped rock can be used as a sundial. This is a truly simple computer-trouvé, but we do have reason to care about the temporal states that its shadow-casting states can be interpreted as representing.

There is perhaps a correct intuition behind the criticism nonetheless. Finding a device sufficiently interesting to warrant the description "computer" probably also entails that its input–output function is rather complex and inobvious, so that discovering the function reveals something important and perhaps unexpected about the real nature of the device and how it works. Thus finding out what is computed by a sieve is probably not very interesting and will not teach us much we did not already know. How a sieve works is dead simple. In contrast, finding out what is computed by the cerebellum will teach us a lot about the nature of the tissue and how it works.

A computer is a physical device with physical states and causal interactions resulting in transitions between those states. Basically, certain of its physical states are arranged such that they represent something, and its state transitions can be interpreted as computational operations on those representations. A slide rule is taken to compute—for example, (Mult 2, 7) to give 14 as the output—by dint of the fact that its physical regularities are set up in such a way as to honor the abstract regularities in the domain of numbers; the system of Aubrey holes at Stonehenge computes eclipses of the sun by dint of the fact

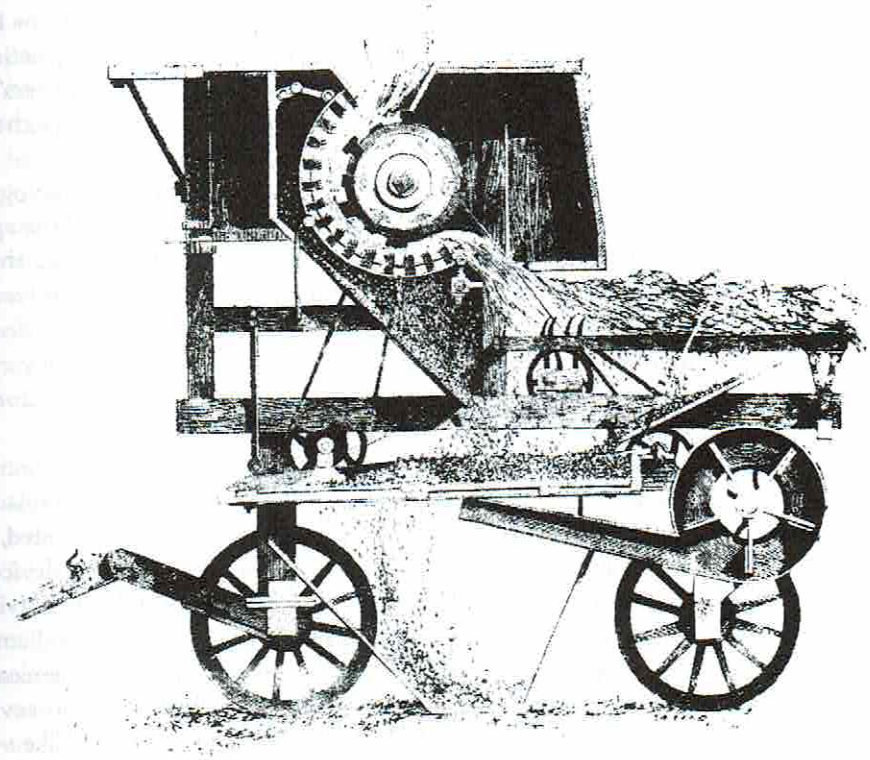

Figure 3.4 Garrett's improved threshing machine, 1851. The wheat was fed in from above, and the grain was removed by the rubbing action of the beater bars on the drum as it rotated inside the fixed concave. The grain fell onto a sieve below and the chaff was blown away by the fan system on the right. (From *The Illustrated Science and Invention Encyclopedia*. Westport, CT: H. S. Stuttman, 1983.)

that its physical organization and state transitions are set up so that the sun stone, moon stone, and nodal stone land in the same hole exactly when an eclipse of the sun occurs. Notice that this would be so even in the highly unlikely event that Stonehenge was the fortuitous product of landslides and flooding rather than human contrivance.

Nervous systems are also physical devices with causal interactions that constitute state transitions. Through slow evolution, rather than miraculous chance or intelligent design, they are configured so that their states *represent* —the external world, the body they inhabit, and in some instances, parts of the nervous system itself—and their physical state transitions execute computations. A circuit in mammalian brain stem evolved to compute the next position of the eyeball based on the angular velocity of the head. Briefly, the neuronal activity originating in the semicircular canals represents head velocity, and the interneurons, motor neurons and eyeball muscles are physically arranged such that for head velocity of a certain amount, the neurons causally interact so that the muscles of eyeball change tension by exactly the amount needed to compensate for the head movement. (For more on this circuit and its computation, see chapter 6). Loosely speaking, this organization evolved "for"

this task; a little more strictly speaking, this circuit came to be the way it is by random mutations and natural selection; in standard epigenetic circumstances and relative to the ancestor's nervous system and to the system's other components, this organization enhances somewhat the organism's chances of surviving and reproducing.

There is a major contrast between manufactured and biological computers. Since we construct digital computers ourselves, we build the appropriate relationship into their design. Consequently, we tend to take this mapping for granted in computers generally, both manufactured and evolved. But for structures in the nervous system, these relationships have to be discovered. In the case of biological computers, discovery may turn out to be very difficult since we typically do not know what is being computed by a structure, and intuitive folk ideas may be misleading.

By contrast with systems we conventionally call computers, the *modus operandi* of some devices are such that a purely causal explanation, without reference to anything having been computed or represented, will suffice. A mouse-trap or a sieve, for example, is a simple mechanical device. Purely causal explanations will likely suffice for some aspects of brain activity too, such as the ion pump in neuronal membranes by virtue of which sodium is pumped out of the cell, or the manner in which binding of neurochemicals to receptors changes the internal chemistry of the cell. Bear in mind, however, that even at this level, an ion, such as Na^+, *could* represent a variable like velocity. At this stage, no one is really convinced that this is in fact so, but the possibility is not ruled out simply because ions are very low-level entities. Effects at higher levels of organization appear to require explanations in terms of computations and representations. Here a purely causal story, even if the line is still fairly clean, would give only a very unsatisfying explanation. For example, a purely causal or mechanical explanation of the integration of signals by dendrites is unenlightening with respect to what information the cell is getting and what it does with it. We need to know what this interaction means in terms of what the patterns of activity represent and what the system is computing.

Consider, for example, the neurons in parietal cortex whose behavior can be explained as computing head-centered coordinates, taking positions of the stimulus on the retina and position of the eyeball in the head as input (Zipser and Andersen 1988). Knowing that some neurons have a response profile that causes other neurons to respond in a certain way may be useful, especially in testing the computational hypothesis, but on its own it does not tell us anything much about the role of those neurons in the animal's visual capacity. We need additionally to know what the various states of neurons represent, and how such representations can be transformed by neural interactions into other representations. At the network level, there are examples where the details of connectivity and physiology of the neurons in the network still leave many of the whys and wherefores dangling, while a computational approach that incorporates the physiological details may make contact with the broader brainscape of tasks, solutions, environmental niche, and evolutionary history.[8]

There is a nonmathematical sense of "function," according to which the job performed by something is said to be its function. In this sense, the heart is said to function as a pump, rather, than say as a noisemaker to soothe babies on their mother's breast. Though making a "ka-thump" sound is something the heart does, and though babies appear to be soothed by it, this surely is not the heart's *function*, meaning, roughly, its "primary job." Functional assignments can reasonably be made in the context of evolutionary development, what the animal needs to survive and reproduce, its environmental niche, and what would make sense given the assignment of function to related structures. In this "job" sense of function, the function of some part of the nervous system is to compute some function (in the mathematical sense), such as position for the eyeball given head velocity.

There is nothing mystical about characterizing a biological structure as having a specific function, even though neither god nor man designed the structure with a purpose in mind.[9] The teleological trappings are only that, and the teleology is eliminable or reducible without remainder in an evolutionary framework. To assign a computational role to a circuit is to specify a job of that circuit—detecting head velocity, for example. Consequently, the considerations that bear on determining the job of an organ such as the liver bear also on the assignment of computational role to neuronal structures. That the nervous system evolved, and that maladaptive structures tend to be weeded out in the evolutionary contest, restricts many functional hypotheses—in both senses of "functional"—that are logically possible but just not biologically reasonable. The crux of the matter is that many biologically irrelevant computational hypotheses can be culled out by a general functional truth about nervous systems, namely that *inter alia* they serve to help the animal move adaptively in the world.[10]

In this chapter we shall characterize a range of computational principles that may be useful when addressing the question of computation in nervous systems. As we shall see, moreover, the computational perspective will allow us to ask questions of biological systems that might not otherwise have been asked. The computational principles introduced here will be applied first to a number of examples chosen for their pedagogical value rather than for immediate biological salience. They allow us to introduce the basic ideas in a simple fashion, and this is their single, overriding virtue. They are not meant to be hypotheses concerning the mechanisms underlying the computational properties of real nervous systems. In chapters 4 to 6 neurobiological realism will be of paramount concern, but an understanding of the basic concepts is the entry ticket to these chapters.

2 LOOKING UP THE ANSWER

Conceptually, the simplest computational principle is "look up the answer." A look-up table is simply some physical arrangement in which answers to specific questions are stored. The engineering trick is to rig the table so that access to answers is fast and efficient, for if it is slow and clumsy, calculating the answers

de novo might be preferable. Inasmuch as look-up tables are really repositories of precomputed answers rather than devices for working out the answer on the spot, it may be suggested that they are not genuine computers at all. For the purist, however, accepting this semantic refinement promotes confusion. A look-up table does after all effect a mapping, it instantiates a rule, and its states represent various things. That, given our groundfloor criteria, qualifies it as a computer. Call it unglamorous, call it humdrum, but a look-up table embedded in a mechanism for delivering answers can as properly be called a computer.

The easiest way to think of a look-up table is simply as an array of boxes each of which says, in effect, "if x is your problem, then y is your answer," for specific x and y. In other words, it does a matching job. For example, the truth table for exclusive "or" looks like this:

P	Q	XOR
T	T	F
T	F	T
F	T	T
F	F	F

This mode of representing the truth conditions happens to be very convenient, though many other, less convenient arrangements are easily imagined. And as students are usually told, it requires no significant intelligence to use this look-up table: just ask your question (e.g., what is the value when P is true and Q is false?), go to that row, and scan the answer.

A second but more powerful look-up table is the slide rule. Actually it is a multiplexed look-up table, since it stores answers not only for multiplication tasks, but also for finding sines, cosines and logarithms. When the task is multiplication, one enters the question (what is 3×7?) by sliding the center piece and cursor, and scanning the answer at the cursor. Moreover, while the truth table can handle only discrete functions, a slide rule can do continuous functions. To accommodate the variety of arithmetic questions and answers on two pieces of wood, the look-up table is metrically deformed (figure 3.5). As before, there are other ways of physically structuring a look-up table to perform exactly these tasks, but the flat, pocketable slide rule is in fact a wonderfully convenient and efficient way to do so.

Extending the idea a bit further, consider the Tinkertoy look-up table constructed in 1975 by a group of MIT undergraduates to play the game of tic-tac-toe[11] (figure 3.6). Making the "table" part of this device consists in storing a set of ordered pairs, where the first element is a possible game position, and the second element is the correct move, given that game position. In operating, the machine looks for a match between its current position and one of the possible positions sitting in storage. Finding the match will automatically divulge what to do next.

The first step in building the Tinkertoy look-up table was to decide on a representation for the state of the board using just the resources of Tinkertoy pieces. The second step was to use rotation and reflection symmetries to reduce the total number of game positions in the table, since the more entries

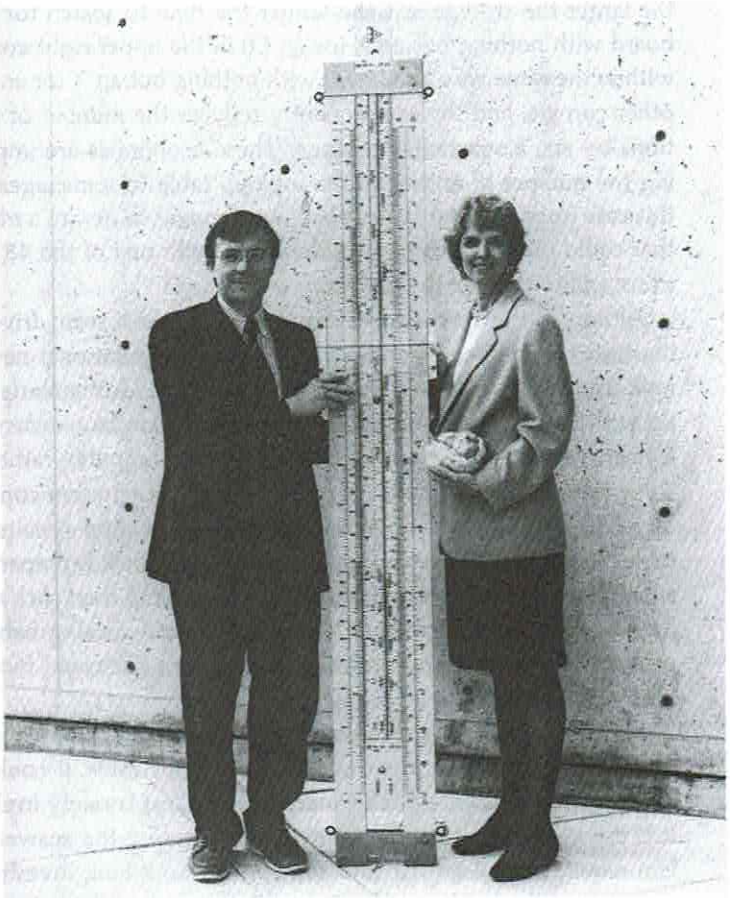

Figure 3.5 The object in the center is an oversized slide rule. The authors are on either side.

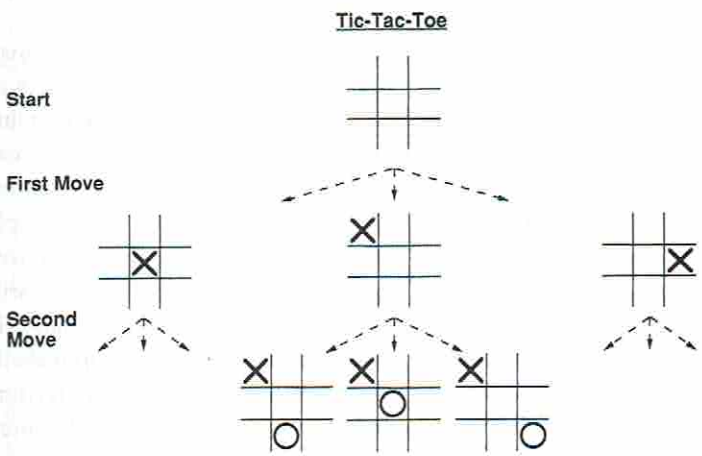

Figure 3.6 The first three levels of the game tree for tic-tac-toe. The 3 × 3 board at starting position is in the center, and the first move must be in one of three board positions (arrows) irreducible by mirror and reflection symmetries. The next level gives several possible replies by the opponent.

the larger the storage and the longer the time to search for a match. Thus a board with nothing but an X (or an O) in the upper right corner can be dealt with in the same way as a board with nothing but an X (or an O) in any of the other corners, and this consequently reduces the number of stored first positions by six, a substantial savings. These economies are important in reducing the number of entries in the look-up table to a manageable number—in this case from 300,000 to 48. The next step was to design a mechanical system that could match a position on the board with one of the 48 irreducible positions, and to retrieve the correct move (figure 3.7).

Although the tic-tac-toe example may at first seem frivolous, it clearly illustrates a number of points relevant to computational neuroscience. First, look-up tables can be constructed from unorthodox materials but still come up with the same answer as do conventional electronic circuits. Second, the Tinkertoy computer is not a general-purpose computer; rather, it was built to solve one specific problem. So far as nervous systems are concerned, the analogy is that the genes probably wire up some neural circuits on the look-up table blueprint with the consequence that the animal is prepared at birth to do such things as snuffle around for a warm spot and then suck at whatever soft, mouth-sized thing sticks out. Circuits that yield sucking behavior in rats are probably not general-purpose devices, but are dedicated more or less exclusively to sucking.

The theoretical lesson is that if a problem is conceptually reducible to a look-up table problem, then, cost and efficiency aside, it could in principle be implemented by look-up table mechanisms. Cost is rarely irrelevant, however. It is especially pertinent here, since precomputing the answers for each problem requires a substantial, and sometimes exorbitant, investment in the construction of the machine. From an evolutionary point of view, it might be too costly or too difficult to precompute certain tasks, such as semantics or place-in-the-social-scheme or dinner-whereabouts, and hence many things must be learned by the infant organism.

How practical really is the look-up table approach? The answer depends on a number of factors, including the complexity of the problem to be solved, the architectural pliancy of the available materials, and the size limits of the look-up table. Chess, unlike tic-tac-toe, appears to be a poor candidate for the look-up table solution. There are approximately 10^{40} game positions—far more than the capacity of any existing machine.[12] The complexity factor can be reduced considerably using the economy described above; namely, take advantage of the underlying symmetries and position similarities in the problem to reduce the number of entries. Given this possibility, it remains to be seen whether the look-up strategy is indeed utterly unrealistic for chess. For many real-world problems, as in the problem of visually recognizing an object, advantage can be taken of translation, rotation, and scaling invariances, as well as smoothness and continuity constraints, to reduce the number of stored categories. The possibility to consider is that look-up craftsmanship may be seen at various stages in nervous system processing, even if it is not ubiquitous.

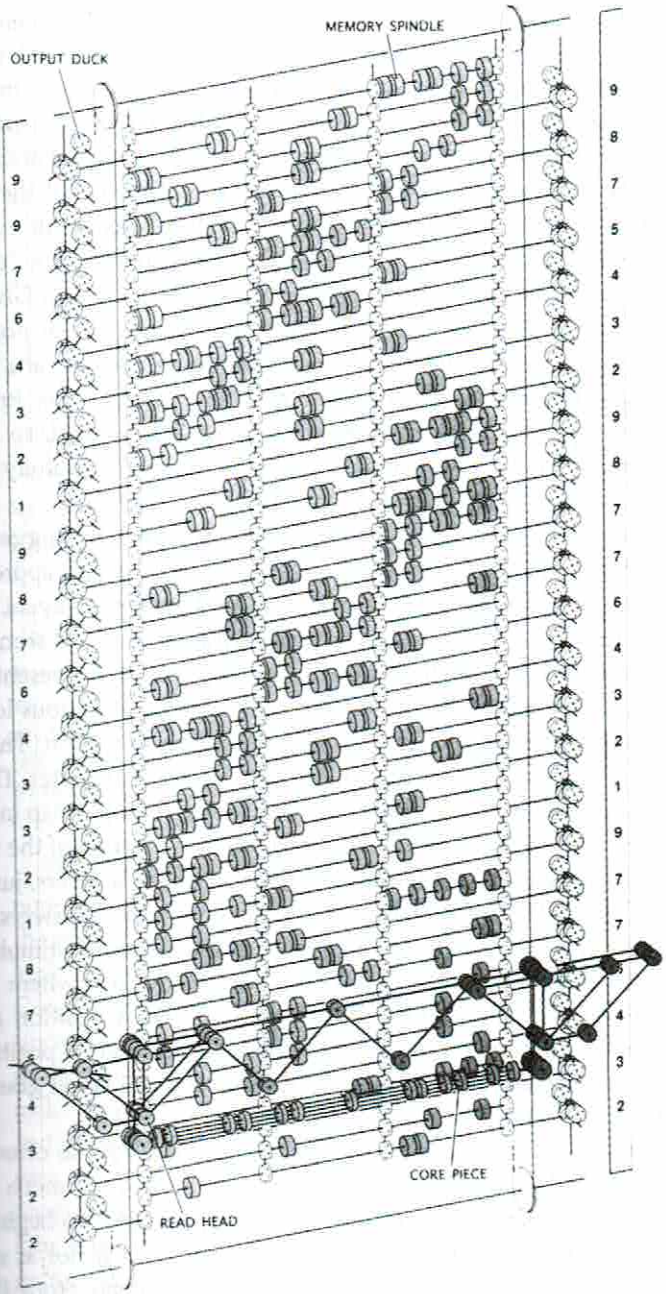

Figure 3.7 Tinkertoy computer for playing the game of tic-tac-toe. Each memory spindle encodes a possible game position, along with the optimal response. The read head, loaded with the current game position in the core piece, moves down the memory spindles, one by one, until there is a match. This activates the output duck, which drives the correct move. Compare this special-purpose computer with the general mapping scheme in figure 3.1. (From Dewdney [1989] Computer recreations: a Tinkertoy computer that plays tic-tac-toe. Copyright © 1989 *Scientific American*.)

If the number of stored question–answer pairs is large, then the search-for-a-match time may be prohibitively long. The Tinkertoy look-up machine, for example, compares the current board position to each of the stored board positions, one at a time, until the match is found. This is a rather ponderous business, especially if the search procedure is sequential. Parallel search could provide time economies, as we shall see later. Time is not the only consideration; wiring too must be kept within bounds. Consider, for example, the size of a look-up table needed to handle ordered pairs of the form ⟨edible goodie at coordinates x,y,z moving at velocity v/body position #⟩. Given the number of independently movable body parts and the number of possible locations and speeds, the look-up table would have to be massive, at a minimum. The wiring cost scotches the idea. For nervous systems, brief times and sparse wiring are generally preferable, other things being equal, so the question is whether there are any neural structures that can be usefully understood as look-up tables.

Until rather recently, the superior colliculus in cats suggested itself as a neural instantiation of a look-up table, at least to a first approximation. The simple story runs like this. The colliculus has a number of layers, or laminae. On its upper layer, the colliculus represents location of visual stimuli on a retinotopic map, while on its bottom layer is a "motor map" representing position of the eyeball muscles suitable for foveating the eye to various locations. Other layers in the structure represent whisker-stimulus location. The maps are deformed with respect to each other so that they are in register. The effect is that a line dropped from the visual map intersects the motor map in a location that causes the eyeballs to move so as to foveate the location of the visual stimulus. The anatomy itself facilitates the look-up of motor answers, just as the "anatomy" of a slide rule facilitates look-up of mathematical answers. The organization enables the system to foveate a peripheral visual stimulus quickly and accurately. It is a kind of two-dimensional slide rule, where the visual and motor surfaces are appropriately aligned so that a position of a peripheral stimulus is mapped onto a where-the-eyeballs-should-be position. According to this conjecture, the anatomy executes a kind of "visual grasp" transformation (figure 3.8).

What is wrong with the colliculus look-up story? As so often in biology, as more data come in, the whole situation begins to look much more complex than the unencumbered, unqualified hypothesis asserts. To begin with, the relation between the visual input and the motor output is not as straightforward as simple "visual grasp"; there are descending fibers from the cortex that affect collicular output, and attentional processes play an important if poorly understood role in collicular function. Although there do exist ostensible "drop line" connections between mapped layers, it is not yet known exactly what these connections do. In particular, the long time delay between the signal entering the sensory layer and a signal reaching the lower motor layer undermines the hypothesis that these connections straightforwardly execute "visual grasp." So withal, the colliculus cannot be taken as an unproblematic case of a neural look-up table.

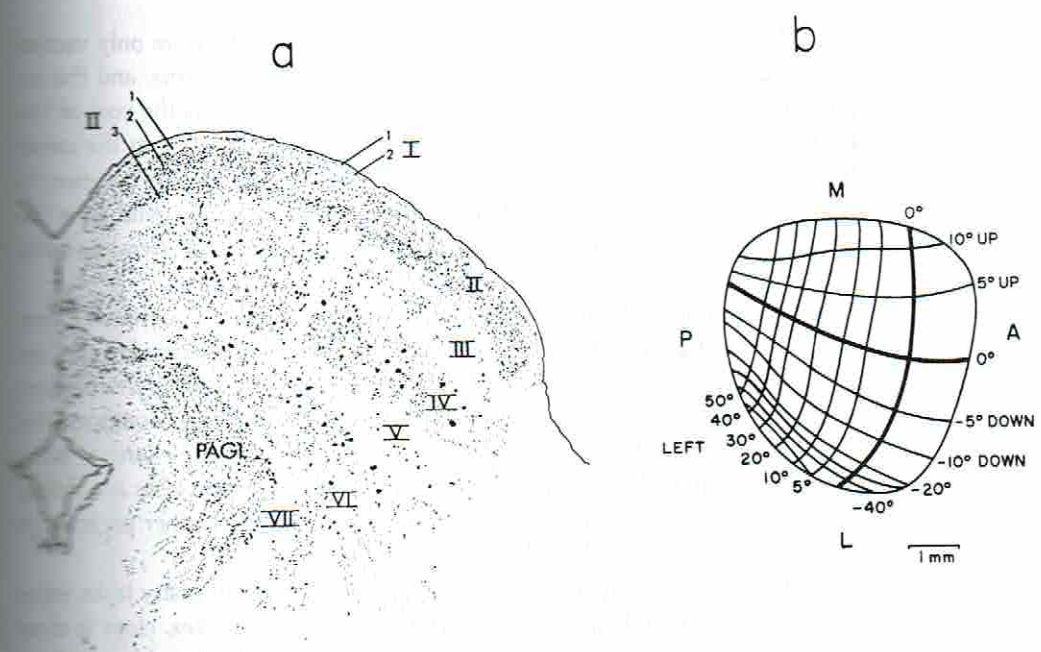

Figure 3.8 Organization of the cat superior colliculus. (a) Cross section through the colliculus showing cell bodies of neurons in each lamina (numbered). (b) Map of the eye movements produced by stimulating the deep layers of the colliculus with an electrode. The coordinates refer to the deviation of the eye from its center of gaze that is produced by electrical stimulation. M, medial; L, lateral; A, anterior; P, posterior (Adapted from Schiller 1984.)

The example is instructive nonetheless, for the discrepancies between the pure look-up configuration and the complicated anatomy and physiology of the colliculus suggest that it well behooves evolution to fancy up a true-blue look-up table into something that can do rather more complicated things. Consider first that the colliculus needs also to take head position into account, since the eyes move relative to the head, and hence relative to the ears and the whiskers. In addition to its "in register" drop lines, the system may find it can make good use of connections to other areas of the map, and to the whisker and ear-position maps interleaved in the neural stack, noting that whiskers and ears too can move relative to the head (in some mammals) but with their proprietary degrees of freedom. But if the colliculus takes these other matters into account, pure look-up conforming to the slide-rule style is not what is going on. Were the eyes stuck fast in the head, and were "foveation" to whisker and auditory stimuli absent, the colliculus might approximate more closely a look-up table. As it is, the complexities of the colliculus suggest that even if evolution had fashioned a pure look-up table, it would soon evolve to master these complex and interrelated operations. That is, neurons would have to perform additional computational steps between input and output.

To return to the matter of computer design, one means for reducing storage space consists in allowing the structures doing the transformations to adjust

themselves to existing conditions. Thrift bids the system to store only vectors (game positions) it actually uses rather than all possible vectors, and this requires that the system learn which vectors these are. What is the cost of this flexibility? If the system is to adapt, the adaptation should be in the *correct* direction. Alas, we cannot very well have a look-up table for the question "is my modification in the right direction?" without going hog-wild over the space limitations. So adaptation pulls the system even further away from the slide-rule paradigm.

But if nervous systems are not using pure look-up tables, what are they doing? The fast answer, to which the rest of the book is an extended elaboration, is this: they are computing with nets. As we shall see later in this chapter, neural nets may have certain properties akin to look-up tables. Consequently, this look-up table prologue is not merely a "first-we-crawl" exercise, but a foundation that will help us understand what actual neural nets are doing. Before moving to a discussion of how nets compute, one further preliminary point must be laid on the board.

Might a close but imperfect match sometimes suffice? For many tasks, especially recognition and categorization tasks, the answer is "yes, close is close enough." Accordingly, an additional modification to the true-blue look-up table consists in storing not every possible entry, but rather storing prototypes or exemplars of the categories. With this stratagem, we trade off a degree of precision for a saving of space, but the system must now take some computational steps to determine similarity to stored vectors. Eventually we shall want a net that avails itself of both economies: it stores prototypes, and it has the plasticity to learn prototypes.

To embody prototypes in a computer, related items are clustered near or less near to an exemplar, according to their degree of similarity. Items stored in this manner define a similarity space in the machine, and distance from the prototype defines a similarity metric. This is known as a *nearest-neighbor* configuration, and there are many possible architectures for realizing it and many possible ways to style nearest-neighbor algorithms to exploit the organization for computational purposes. If a conventional digital machine is chosen for the implementation, then the machine will have to be spatially prodigal to accomplish complex tasks, for it has to store all the entries, make the distance measures, find the match, and deliver the answer. Clever ways to store data in hierarchical trees have been devised, but even these bog down when the going gets realistic. This gloomy prospectus may be sufficiently discouraging to degrade the look-up idea to nothing more than a charming curiosity, rather like an ornithopter[13]—conceivable perhaps, but practical, probably not. On the other hand, though a digital machine may be impractical in this sphere, there is an architecturally very simple organization that will do the job, and do it cheaply, efficiently, and in satisfyingly few steps. That is a net. In the next several sections, we shall look at a number of types of networks, starting with very simple examples and moving on to networks of greater power and sophistication.

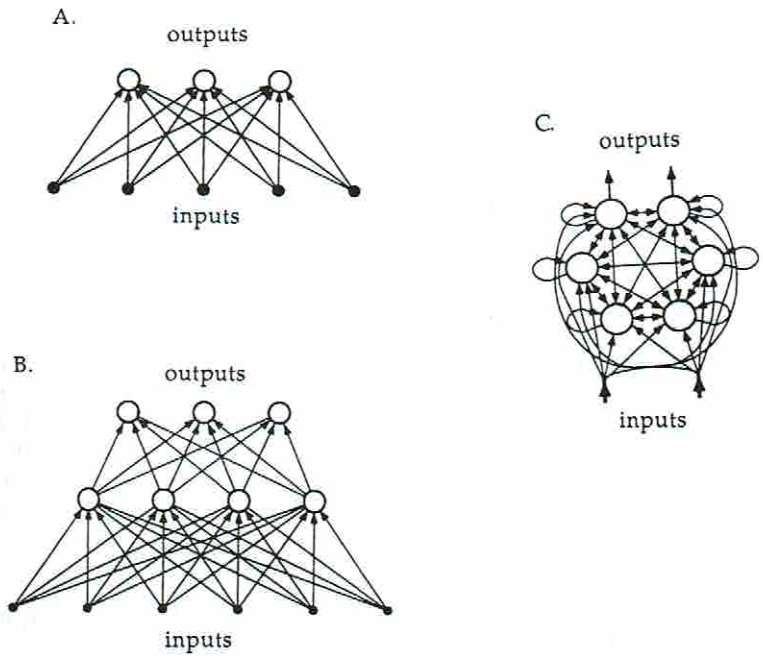

Figure 3.9 Three types of networks. (A) Feedforward network with one layer of weights connecting the input units to the output units. (B) Feedforward network with two layers of weights and one layer of hidden units between the input and the output units. (C) Recurrent network with reciprocal connections between units. (Adapted from Hertz et al. 1991.)

3 LINEAR ASSOCIATORS

What is a net? The architecture of the canonical net consists of units, loosely based on neurons, connections (generously speaking, axons) between the units, and weights (generously speaking, synapses) on the connections (figure 3.9). Some units receive external input, some deliver the output, and some may do neither. Because there is more than one input unit and more than one output unit, the ingoing and outcoming representations are vectors, meaning ordered sets of values (e.g., $\langle 3.2, 668.9, 0 \rangle$) rather than single values (scalars, e.g., 668.9). Signals with various magnitudes are passed between units. That is the nub of a net. How can a net compute anything? The abstract explanation is reassuringly simple: the weights on the units are set in such a way that when the input units are given certain values, the output units are activated appropriately. This means that a mapping is achieved and hence that a function is executed. Now to follow the recipe for making a net in a concrete case, we have to decide how to set the weights, whether they are modifiable and if so how, what range of activity values a unit may take and how they are determined, how to represent the input vectors, and the nature of the connectivity between units (network topology). Obviously this means that the canonical description carves out a vast area of computational space, within which specific nets occupy small regions. The canonical description thus stands to a running

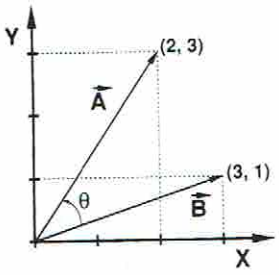

Figure 3.10 Computing the inner product between two vectors is a fundamental operation in a feedforward associative network. The example given here is for two-dimensional vectors, but the same relationships apply to vectors with more components. The inner product is defined as $\mathbf{A} \cdot \mathbf{B} = A_x B_x + A_y B_y = 2 \times 3 + 3 \times 1 = 9$, where A_x is the x component and A_y is the y component of vector **A**. The angle θ between the vectors can be computed from the relationship $\cos\theta = (\mathbf{A} \cdot \mathbf{B})/(\|\mathbf{A}\| \cdot \|\mathbf{B}\|)$, where $\|\mathbf{A}\| = \sqrt{A_x^2 + A_y^2}$ is the magnitude of **A**. In the network shown in figure 3.9a, vector **A** might represent the activity levels of the input units and vector **B** could be the weights from the input units to an output unit. The inner product can then be interpreted as the sum of the weighted inputs to the output unit.

neural network model like a dictionary definition of an airplane stands to a veritable machine itself.

In the 1970s, more or less independently, a number of people were developing associative nets, including Leon Cooper, James Anderson, Teuvo Kohonen, Gunther Palm, Christopher Longuet-Higgins, and David Willshaw.[14] How do associative nets work? In a nutshell, they associate an input vector with an output vector, essentially following the "parallel architecture/similarity-measure/look-up table" format outlined above. The key mathematical task that networks can perform is computing inner products; that is, taking two vectors and multiplying them component by component and then adding up the products. So if one vector represents the input from a set of units and the other vector is a stored prototype, then the inner product yields a measure of the overlap between them, and hence of their similarity. Geometrically, the inner product is proportional to the cosine of the angle between the vectors, so when there is perfect congruence of the vectors, the angle would be zero (figure 3.10). This is a readily manipulable measure of vector similarity. How are the vectors (prototypes) stored? They are stored in the weights connecting the input units to an in individual output unit. This means that each component in the prototype vector is assigned to one weight, and these weights are attached to a summing unit, which adds up all the products of these two vectors (weight vector and input vector). This summing unit is the output unit.

In the garden variety network, the output of the summing unit is proportional to the sum of the products. The output therefore is a linear transformation of the input, and the network is called a *linear associator*. For example, for a small network with three input lines and three output units, there are nine possible weights, which can be written as an array of numbers. One such 3 × 3 weight matrix is:

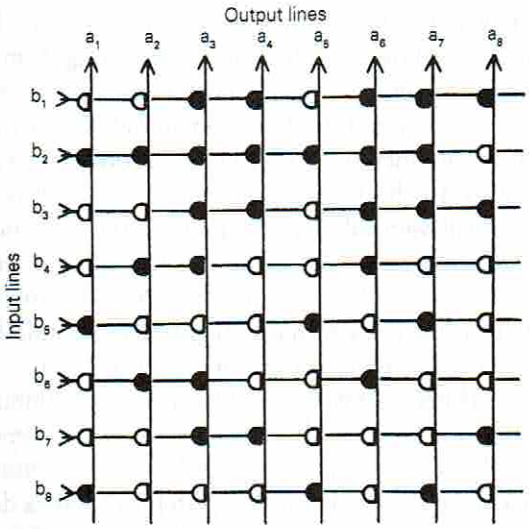

Figure 3.11 A Willshaw net showing the input lines (horizontal), the output lines (vertical), and the connections between them. The weights on the connections are binary and can be zero (open circles) or one (closed circles). Thus, the picture is a graphical representation of the weight matrix. (From Willshaw, 1989.)

$$w_{ij} = \begin{bmatrix} 0 & -1 & 2 \\ 1 & 0 & 1 \\ 2 & -1 & 0 \end{bmatrix}$$

The components of the output vector of the network are given by:

$$y_i = \sum_j w_{ij} x_j \tag{1}$$

For the input vector $x_i = (1, 1, 1)$, the output vector is given by $y_i = (1, 2, 1)$. For the input vector $x_i = (1, 2, 3)$, the output vector is given by $y_i = (4, 4, 0)$. (Multiply the first component of the vector by the first item in the top row $[1 \times 0]$, the second component by the second item in the top row $[2 \times -1]$, the third by the third $[3 \times 2]$. Add the products $[= 4]$. Repeat for each row.)

As you would expect, there are many variations on the simple linear associator theme, and changes are rung on whether the input and output units take on continuous or binary values, and whether the weights are continuous or binary (figure 3.11). Notice that many inner products can be computed in parallel, one for each summing unit, and the more the summing units, the bigger the net. Additionally, each of the products (weights × inputs) can be computed in parallel. The result is that only one step is required to produce an output vector that is associated with the input vector. What is described here is the paradigmatic net for matching a sample to a stored prototype. This is evidently a classification task: for each category a representative example must be provided, and this is encoded as a vector. As a rule, the less overlap between the prototype vectors the better, since it is preferable that an input unambiguously match a prototype.

The explanation so far shows how to get *a* mapping from input to output, but it has not addressed how to set the values of the weights in order to get the *correct* mapping from input to output—to get the net to give the correct answer when it is asked a question. This is obviously of the essence if the net is to be of any use. The method by which the prototype vector is encoded matters enormously to the success and efficiency of the net, and different coding strategies will variously ease or gum up the operations. It is not in general known how to go about organizing the preprocessing for nervous system tasks such as vision and speech recognition, but some information is available for simpler processes such as visual tracking during head movement (chapter 6). As we discuss later, if one wants to get some insight into the characteristics of the preprocessing for good matching of input to stored vectors, then the brain will be a valuable source of ideas. The general point about preprocessing is this: at the level of sensory input, the vectors can look very different because there are many possible patterns of, say, a dog. The preprocessing has to be done in a such a way that if the many different patterns all trigger the output vector "dog," they must be mapped onto the weights so this happens. In other words, the system needs a many : 1 mapping. Exactly how to set this up varies from case to case, and more discussion of this topic follows.

A slight modification of the paradigm net yields a network that performs *autoassociative content-addressable memory*. This means that the net can produce an output that is as close as possible to a prestored vector given only part of the vector as input. This is a vector completion task, in other words. To do this, you need as many output units as input units, so that the weight matrix is square. The weights w_{ij} from unit j to unit i are constructed from an outer product of the stored vectors

$$w_{ij} = \sum_{\alpha} x_i^{\alpha} x_j^{\alpha} \tag{2}$$

where x_i^{α} is the ith component of the α-th stored vector. For example, if one of the stored vectors is (1, 5, 2), then its contribution to the square weight matrix is

$$w_{ij} = \begin{bmatrix} 1 & 5 & 2 \\ 5 & 25 & 10 \\ 2 & 10 & 4 \end{bmatrix}$$

These input vectors could be presented to the network one at a time, and the weights could be computed incrementally by adding each contribution to produce the sum. This is perhaps the simplest and best known of all learning rules, the Hebb rule, so-called because it reflects Hebb's hunch that connection weights between two units should strengthen as a function of the correlated activity of the connected units. Note that the weight is built up from the product of input activities and desired output activities.

The Hebb rule is loosely speaking a "get-similar" rule. It says, "Make the output vector the same as the one you saw before which it most closely resembles." Consider what will happen, then, when an incomplete or noisy version of an input vector x_i is presented to the network with a weight matrix

configured according to eq. (2). If we substitute the weights in eq. (2) into eq. (1), then the output can be rewritten as:

$$y_i = \sum_{\alpha} x_i^{\alpha} \left[\sum_j (x_j^{\alpha} x_j) \right] \tag{3}$$

Roughly speaking, this means the output vector is composed from a linear combination of the stored vectors, x_i^{α}, where each vector is weighted by the term in the square brackets, representing the inner product, or overlap, between the input vector and the stored vector. In the *auto*associative network the desired output vector is the closest stored input vector; at best, the output vector will be identical to the stored input vector. Such a network can perform vector completion or vector correction, but it cannot associate two different vectors. The basic autoassociative net can be modified quite simply to handle *hetero*associations. Whereas the autoassociative net stored the outer product of $(x_i^{\alpha}, x_j^{\alpha})$, the heteroassociative net needs to store the outer product of two nonidentical vectors, $(z_i^{\alpha}, x_i^{\alpha})$. Once trained, the heteroassociative network will produce an output proportional to z_i^{α}, when x_j^{α} is present in the input vector.

What happens if the noisy vector matches more than one stored vector? Then the output will be a weighted sum of the stored vectors (i.e., ambiguous inputs will produce hybrids of the nearest matching vectors). As more and more vectors are stored according to the Hebb rule, there will be more and more ambiguities and the performance of the network will correspondingly degrade. Indeed, an important fact about all such matrix associators is that they work very well so long as only a small number of patterns are stored, but their capacity is limited and they do get filled up surprisingly quickly. To meet this storage room problem, modifications of the Hebb rule have been proposed that increase a little the capacity of the network.

The first strategy is to use only a small subset of units to represent any given item. This is also known as making the vectors sparse (Willshaw 1981). If there are n units, Willshaw found that optimal storage occurred when $\log n$ units are used to represent each item. The advantage of making the vectors sparse is that there is less overlap between representations, and hence a given network can store a greater number of representations.[15]

A second way to economize on space is to normalize the incoming activation by inhibitory connections, assuming that all the input connections are excitatory. In a Hebb network, the number of synapses that are most highly enhanced by an input vector is proportional to the square of the number of units that the input activates. It follows that by going to a sparse representation, the number of units activated by any given pattern is reduced. Feedforward inhibition achieves this by normalizing the total activity so that all input patterns produce, on average, equal excitation. This achieves an economy because it prevents a single, highly active input vector from hogging the synapses.

A third way to increase the capacity of the network is to introduce a nonlinear threshold for the output units. This means that should the summed inputs fail to reach a prescribed value, then there is no output. Thus only the

most strongly activated units produce an output. A sort of cleaning out of marginal activity is thereby achieved, making room for business that counts. While housecleaning is not to be scoffed at, what turns out to be really progressive about adding nonlinearity is that it makes the net far more powerful, permitting it to perform computations much more complex than anything a linear net can handle. In other words, a whole panoply of computable functions hitherto impossible for the net becomes within its range. Nonlinearity in the response functions of the units is the next development in the evolution of invented nets.

4 CONSTRAINT SATISFACTION: HOPFIELD NETWORKS AND BOLTZMANN MACHINES

What sorts of problems demand a net with nonlinear properties? Consider the problem of recognizing an object in a visual image. The object may be in an unusual perspective, it may be partially occluded in a cluttered scene, lighting conditions may be poor, or the object may be an individual that has never been seen before. One of the first steps in visual processing separates the object from the surrounding clutter. This segregation of figure from ground has important consequences for the interpretation of an object in an image, as illustrated in figure 3.12 showing the classical vase/face reversal. Depending on which part of the image is considered the figure and which the ground, the silhouette can be interpreted as either as a vase or as two faces in profile. This also illustrates an important feature of how the visual system deals with ambiguity, namely that only one of the interpretations can be perceived at any given time. Furthermore, one can flip between the two interpretations by shifting attention. Note that this shift need not be an overt shift in gaze, but rather an internal attentional shift that at least sometimes is under conscious control. Could figure–ground segmentation be performed by a look-up table?

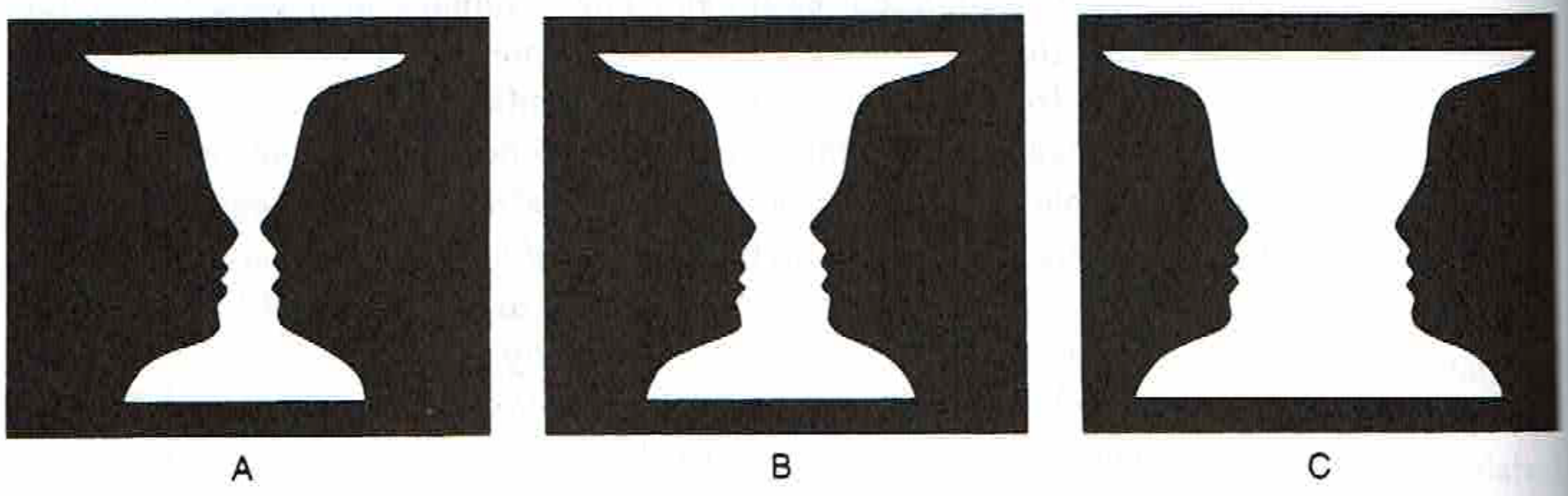

Figure 3.12 Figure-ground reversal. There are two perceptual interpretations of these images: a pair of black faces, or a white vase. The perceptual interpretation can be influenced by conscious attention and biased by features in the image. Thus, the faces interpretation is usually favored in A and the vase interpretation is favored in C. One interpretation appears to exclude the other (try to imagine a face "kissing" a vase). (With permission from Coren and Ward [1989]. *Sensation and Perception*, 3rd ed. Copyright © 1989 Harcourt Brace Jovanovich, Inc.)

Figure 3.23 Image of a figure with an incomplete outline that may still be perceived as a familiar object. (With permission from Coren and Ward [1989]. *Sensation and Perception*, 3rd ed. Copyright © 1989 Harçourt Brace Jovanovich, Inc.)

fact the state of the network with the lowest energy. The procedure is also forgiving, in the sense that the network will fill in properly even should the input be somewhat degraded, that is, should there be a few gaps in the boundary. In this respect, it resembles human perception (figure 3.23) (Mumford et al. 1987).

This network is an instance of a more general computational approach known as "relaxation labeling" that has been applied to a variety of problems in computer vision (Waltz 1975, Hummel and Zucker 1983). In the figure-ground example of constraint satisfaction performed by a Boltzmann machine, the weights on the units were set by hand, meaning that the modeler had to figure out by rather long and laborious trial-and-error procedures what weights were needed to ensure that the global energy minimum really is the solution for any figure–ground problem. Ideally, what one would like is an automated procedure for setting weights, perhaps triggered by showing the network examples of question–answer pairs, and then letting it "figure out" what weights would do the job. How could a network do that? How could a network automatically adjust its weights appropriately?

5 LEARNING IN NEURAL NETS

Learning[22] algorithms for automated weight-setting in networks come in two basic molds: supervised and unsupervised. The basic difference concerns

whether the net infers a weight modification from a report on its behavioral performance. Supervised learning relies on three things: input, the net's internal dynamics, and an evaluation of its weight-setting job. Unsupervised learning uses only two: input, and the dynamics of the net; no external report on its behavior vis-a-vis its weight-setting progress is provided. In either case, the point of the learning algorithm is to produce a weight configuration that can be said to represent something in the world, in the sense that when activated by an input vector, the correct answer is produced. Nets using unsupervised learning can be configured such that the weights embody regularities in the stimulus domain. For example, if weights are adjusted according to a Hebb rule, then gradually, without external feedback and with only input data, the net structures itself to represent whatever systematicity it can find in the input, such as continuity in boundaries. This means that unsupervised nets are useful in creating feature detectors, and consequently unsupervised nets can be the front end of a machine whose sensory input must be encoded in some perspicuous fashion before it is sent on for use in such tasks as pattern recognition and motor control.

We emphasized that unsupervised learning has no access to external feedback, and we now take up the possibility that it nevertheless allows for internal error feedback. Because there is a confusion in the literature concerning error feedback and the convention for applying the label "supervised," we propose explicit labels for the distinction between external and internal feedback. When the feedback is external to the organism, the learning is called *"supervised"*; when there is an internal measure of error, we call the learning *"monitored"* (figure 3.24). Consider, for example, a net required to learn to predict the next input. Assume it gets no external feedback, but it does use its previous inputs to make its predictions. When the next input enters, the net may be able to use the discrepancy between the predicted input and the actual input to get a measure of error, which it can then use to improve its next prediction. This is an instance of a net whose learning is unsupervised, but monitored.[23] More generally, there may be internal measures of consistency or coherence that can also be internally monitored and used in improving the internal representation. The confusion in the literature is owed in part to the fact that the very same algorithms used for supervised learning can be suitably internalized for monitoring. Clarity of the semantics is especially important in discussing feedback modes in nervous systems, where certain kinds of supervised learning may be unbiological, but error detected by a monitor in one part of the nervous system is a plausible teaching signal for another part of the nervous system.

Is there a type of system that acquires organization even though it is neither supervised nor monitored? In real nervous systems, some aspects of development, such as establishing connections between nerve cells, might be considered candidates for self-organization of this kind. In models with unsupervised learning, such as competitive learning, it initially appeared that no internal objective function served as a monitor. Subsequent analysis has shown, however, that objective functions can be found for each of these models. Optimization of these implicit functions is responsible for the ability of the network to

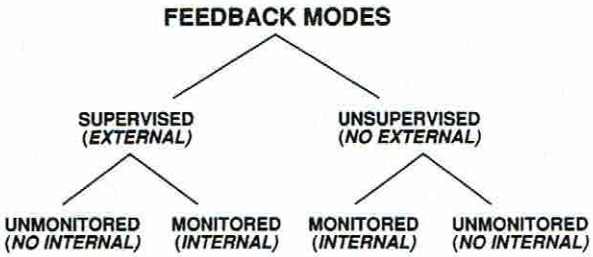

Figure 3.24 Taxonomy of learning procedures. Supervised learning occurs when there is feedback on the performance of the system from the external environment. If the feedback is a scalar reward, it is called reinforcement learning. The learning is called monitored if the system has an internal measure of error. These distinctions refer to the *system* and not the algorithms used: thus backpropagation of error, which is normally used in supervised, unmonitored (S& ~ M) systems, could also be used in an unsupervised, monitored (~S&M) system. For example, a feedforward net with fewer hidden units than input units can be trained to reproduce input patterns—a form of image compression (Cottrell et al. 1987). A more sophisticated example of an ~S&M net uses information-theoretic measures internal to the net to train the hidden units to predict the values of neighboring hidden units (Becker and Hinton 1989). An example of a supervised, monitored (S&M) system is the associative search network with internal predictor (Barto et al. 1981). In this system, the internal predictor learns to anticipate the reward; the difference between the predicted reward and the actual reward is used to adjust weights to hidden units. The internal monitor can be quite a sophisticated high-dimensional error signal, as in motor learning when a distal measure of performance (missing the basket) is used for adjusting a complex motor program (jump shot) (Jordan and Rumelhart 1990).

organize itself into a computationally successful state (Durbin and Willshaw 1987, Linsker 1990b). The conjecture, therefore, is that all successful self-organizing systems, including biological ones, have an implicit objective function that is optimized during the learning process (Sejnowski 1987). For an example, see section 5.9 on the development of ocular dominance columns.

Supervised learning comes in various grades as a function of the report card format. The report-card may (1) merely say "Good answer" or "Bad answer" (Sutton and Barto 1981, 1990), (2) specify a measure of the size of the error with some degrees of precision, or (3) give rich detail, saying, in effect, "You said the answer was abcd; the answer should be ahcp." Given the range available in (2), this allows a continuum of report-card formats. Regardless of the format of the report card, the point of feedback is to give the net opportunity to reduce the error in its output.

In the original Hopfield nets, the learning rule was an update rule in the Hebbian mold. The problems given the net were, however, carefully chosen to be solvable by it. The class of problems solvable by these nets using the Hebb rule is rather narrow, embracing only first-order statistical problems. That is, problems where the question is "do feature A and feature B correlate?," which in machine terms means "are the A-unit and the B-unit on together and off together?" Problems beyond its scope are higher-order statistical problems, e.g., "what is the correlation story for {A, B, C, D}, or for {EF, EH, GH}?" Going beyond these narrow limitations is desirable, since many problems cannot be solved using only lower-order statistics. To target high-order problems, the

basic architecture of the machine must be expanded to include units that intervene between external input and behavioral output. Called "hidden units," they typically connect to the input units, to each other, and to output units when there are any. Adding one or more layers of hidden units allows the net to handle high-order statistics, for, crudely speaking, the extra set of connections and extra dimension of interactions is what permits the net a global perspective despite its local connectivity.

The ability of hidden layers to extract higher-order information is especially valuable when the number of input units is large, as it is, for example, in sensory systems. Suppose an input layer has n units in a two-dimensional array, as does the retina, or a one-dimensional array, as does the cochlea. If the units are binary, then the total number of possible input patterns is 2^n. In fact, neurons are many valued, so the problem is really somewhat worse. Suppose all patterns (state combinations) were equally likely to occur, and suppose one hidden unit represents exactly one input pattern. This would make it possible to represent any function in the output layer by suitable connections from hidden units. The trouble arises when n is very large, e.g., a million, in which case the number of possible states is so large that no physical system could contain all the hidden units. Since not all possible input patterns are equally likely, only a very small subset of all possible input patterns need be represented by the hidden units.

Accordingly, the problem for the hidden units is to discover what combinations of features are ignorable, which features systematically occur together or are otherwise "cohorted," and among *those*, which are the combinations to "care" about and represent. The information for this last task cannot be garnered from inside the net itself, but must be provided from the outside. The division of labor in a net with hidden units looks like this: unsupervised learning is very good at finding combinations but cannot know which subset to "care" about; supervised learning can be given criteria to segregate a "useful" subset of patterns, but it is less efficient in searching out the basic combinations. So by means of unsupervised learning, a basic sorting is accomplished; by means of supervised learning, a subset of basic combinations can be extracted as the "useful" ones.

As the net runs, hidden units may be assigned states according to either a linear or a nonlinear function. If the hidden units are linear, there is an optimal solution called the *principal components*[24] (figure 3.25). This procedure can be used to find the subset of vectors that is the best linear approximation to the set of input vectors. (As we shall see in chapter 5, in a model devised by Miller and Stryker (1990) for development of ocular dominance columns in visual cortex, Hebbian learning finds principal components.) Principal component analysis and its extensions are useful for lower-order statistics, but many of the interesting structures in the world—the structures brains "care" about—are characterized by high-order properties. If luminance is taken as the 0th order property, then boundaries will be an example of a first-order property, and characteristics of boundaries such as occlusion and three-dimensional shape will be higher-order properties. Nonlinear hidden units are needed to represent

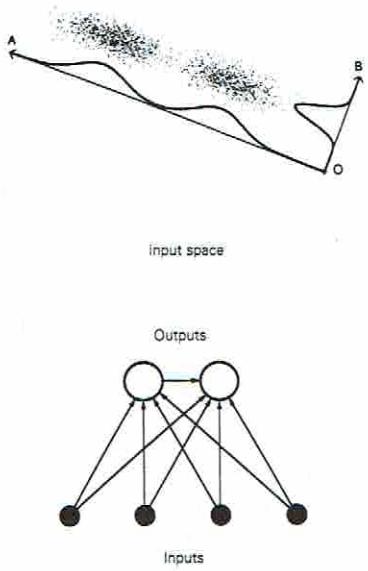

Figure 3.25 A feedforward, unsupervised network that performs principal component analysis. The input vectors, represented by dots in the plane, form two elongated clusters. The first principal component direction, along the line A, is the projection that maximizes the variance of the inputs. Discriminating along this direction is likely to be helpful in separating inputs into two or more classes. The direction of the second principal component, B, is the axis with maximum variance in the subspace orthogonal to A. These directions can be easily extracted from the inputs by the network below, which uses a modified form of the Hebbian learning rule on the feedward weights (Oja 1982) and an anti-Hebbian learning rule on the lateral connection between the output units (Rubner and Tavan 1989, Leen 1991). Following learning, the weights to each output unit correspond to the direction of a single principal component. Moreover, multilayered networks with localized receptive fields can successively extract more complex features from the input space (Linsker 1986, Kammen and Yuille 1988). (From Hertz et al. 1991.)

these higher-order properties. If hidden units are to self-organize so that they can represent these properties, procedures more poweful than principal component analysis must be found.

How then should weights of hidden units be adjusted so that the net can do higher-order problems? Finding a suitable weight-change rule looks really tough, because not only are the units *hidden*, but they may be *nonlinear*, so trial and error is hopeless, and no decision procedure, apart from exhaustive search, exists for solving this problem in general. Moreover, any solution to the weight-adjustment problem depends critically on the architecture and dynamics of a given net. For most architectures and dynamics, the solution is simply not known.

In the specific case of the Boltzmann machine, however, a procedure whereby its nonlinear hidden units can learn to extract higher-order properties is known. The crux of the procedure depends on Boltzmann machines having an interesting property at equilibrium, namely their states have a Boltzmann distribution, eq. (10), which gives, for any global state of the system, the probability of that state occurring at equilibrium. This means that at equilibrium we

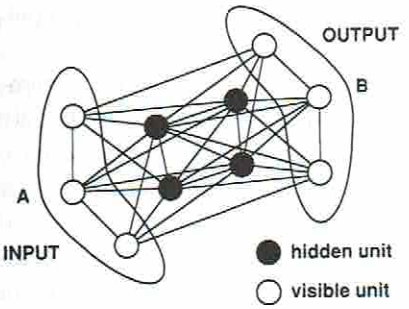

Figure 3.26 Schematic diagram of a Boltzmann machine. The units in the network have binary values and the connections between them are reciprocal. The weights on the connections can be trained by presenting patterns to the input units in the presence and the absence of output patterns and applying the Boltzmann learning rule. During the learning process, all of the weights in the network are modified, including those among the hidden units, which do not receive direct information from outside the network. The hidden units develop features that allow the network to perform complex associations between input patterns and output patterns. Hidden units give Boltzmann machines powerful internal representations not available to networks containing only visible units.

know the global consequences of any local weight change (which will change the energy). Now take the converse of this rule, and we have, for any desired global state, a simple procedure for increasing the probability of that state occurring by changing a weight locally.

It is important to emphasize that all the globally relevant information needed to update the weight is available locally. This may seem contradictory, given that units may be connected only with their immediate neighbors. It is, however, a simple consequence of the net's connectivity: a unit's neighbors are connected to its neighbors, who are in turn connected to all their neighbors, and so forth. Given the connectivity and the Boltzmann distribution at equilibrium, it is guaranteed that information from synaptically distant units is propagated throughout the net. The weight modification rule, therefore, is:

$$\Delta w_{ij} = \varepsilon[\langle s_i s_j \rangle_{\text{clamped}} - \langle s_i s_j \rangle_{\text{free}}] \tag{11}$$

where ε is the rate of learning, s_i is the binary value of the ith unit, and $\langle \ldots \rangle$ indicates that the quantity ... should be averaged over time after the network has reached equilibrium.[25] In the clamped condition, both the input and output units are fixed to their correct values, and in the free condition, only the inputs are so fixed (figure 3.26). This is in the supervised mode; in unsupervised mode, none of the units is fixed in the free condition.

Each application of the learning rule is a cycle with three steps: (1) clamp value of inputs, let machine come to equilibrium in the clamped condition, then compute co-occurrences of pairs of units; (2) compute co-occurrences of states in the absence of the inputs (unclamped condition and let the machine find equilibrium again); (3) subtract the two, and adjust the weight proportionally to the difference (the compare condition). Although the net is certain to learn correctly, the drawback is that it requires many, many three-step cycles to

learn one input pattern, and it must repeat this for every input pattern it is to learn.

We have seen how a Boltzmann machine could do pattern completion, in that once trained-up, if the net were given an incomplete pattern as input, it would go into a state representing the complete pattern. Can we get it to do an input–output mapping? Yes, and here is how to move from an unsupervised Boltzmann machine to a supervised Boltzmann machine without changing architecture or algorithm but only by "externalizing" the feedback. First, arbitrarily split the array of input units into two groups, A and B. The only procedural modification consists in feeding group A the same input pattern # during *both* the clamped phase and the compare phase. The second group, B, of input units is more conventional; it is fed its pattern * only during the clamped phase. In effect, then, the hidden units are coming to associate pattern # with pattern *. This can look like pattern completion on the part of the hidden units, but because the hidden units also have reciprocal connections to the inputs, it can look like the hidden units give output * for input #. That is, think of the hidden-to-group-B input connection as a kind of output. Then we have the arrangement whereby in the trained-up net we can feed the net #, and via the hidden reciprocal connection pattern * appears in group B. This means that group B is, for all intents and purposes, an external teacher, telling the net (during group B's clamped phase) what new thing to associate with #.

It takes only one counter-example to sunder an impossibility claim. Boltzmann learning in a net with hidden units—and even with nonlinear hidden units—was a counter-example to the received wisdom according to which the learning problem for multilayered networks was intractable (Minsky and Papert 1969). With the door open, weight-adjusting rules other than that used in the Boltzmann machine were sought. It is now clear that there are many possible solutions to the weight-adjusting problem in a net with hidden (and possibly nonlinear) units, and other solutions may draw on nets with a different architecture and with different dynamics. Thus nets may have continuous valued units, the output function for a unit may have complex nonlinearities, connections between units need not be symmetric, and the network may have more interesting dynamics, such as limit cycles and constrained trajectories. Weight-adjusting problems are really solved by an ordered triple: ⟨architecture, dynamics, parameter-adjusting procedure⟩. In the final section of this chapter we shall outline a very general approach to handle all of these cases.

6 COMPETITIVE LEARNING

Because including a "teacher" as an adjunct to a network is informationally expensive, not to mention biologically unrealistic, it is important to explore the domain of unsupervised learning procedures. As a first pass, a rule of thumb for identifying those features in the sensory input stream that are likely to be useful in categorizing is this: the more frequently a feature occurs in various input vectors, the more likely it is to be salient in categorizing an input as belonging to a certain class. For example, if for a certain bottom creature the

presence of predators typically co-occurs with a looming darkness, then it would make sense for units in the network to extract changes in intensity at all possible locations in the visual field and represent them in its output. Extracting what is criterial rather than performing all-feature coverage also has an obvious advantage for image compression where the goal is to represent an image with the fewest bits of information. Thus, if boundaries of objects are marked by discontinuities in luminance, then a network might most efficiently represent objects by allowing a single unit to represent a long length of luminance border. In short, it is cheaper to represent a stretch of border with a single unit rather than have many units representing small segments of a straight border. In this sense we get information compression.

In addition to the Boltzmann style of unsupervised learning, other kinds of interactions can self-organize so that the network embodies compressed representations.[26] Consider a simple two-layer network with a set of input units, and one layer of weights connecting them to a set of output units, laterally connected by mutual inhibition. This arrangement is competitive in the sense that the mutual inhibition will create a winner-take-all profile at the output level: if an input pattern happens to excite unit 1 to a greater degree than any of the other output units, then unit 1's consequent activity will tend to suppress the others to a greater degree than they are suppressing unit 1. This is an example of relaxation of a network into a stable pattern. As with the earlier examples of relaxation, this network is guaranteed to converge to a final state in which unit 1 is strongly active when input pattern A is presented, while its cohort output units are suppressed. It is assumed that the activity level of the output unit is 1 in case it is a winner, and 0 otherwise. This is what permits characterization of the output unit's representation as winner-take-all.

The description so far has been confined to the net's *activities* given an input. The next matter concerns using this base to set the weights so that when pattern A is next presented, the network will go straightaway to the correct representation. Learning in this net can be accomplished by changing the weights to the winner unit i according to this rule:

$$\Delta w_{ij} = \varepsilon x_j \tag{12}$$

where x_j is the jth component of the input vector, and the ith output unit is the winner. The rule is essentially Hebbian inasmuch as weights increase when pre- and postsynaptic units are concurrently active. As the label implies, in winner-take-all mode, only the winning output unit is active. Although this rule for weight adjustment should work in principle, in practice it is inadequate because the weights can grow without bound and eventually one unit will dominate the rest, and will be activated for any input pattern, thus losing the capacity to discriminate between patterns. How can we rig it so that the unit is excited when and *only when* particular input vectors are presented? Recall that the activity of an output unit is the inner product of the input vector and the weight vector. The strategy is to adjust the weights so that the weight vector for each unit becomes congruent with the input vector it specializes in. This can be accomplished without hand-setting by modifying eq. (12) as follows:

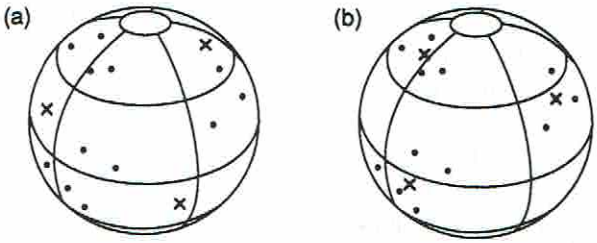

Figure 3.27 Competitive learning. The dots represent input vectors normalized to unit length so they lie on a sphere. The weights are similarly normalized, and the input weights for the output units are indicated by crosses. (a) Organization of the network before learning. (b) After learning, each output unit has migrated to a cluster of input vectors. (From Hertz et al. 1991.)

$$\Delta w_{ij} = \varepsilon(x_j - w_{ij}) \tag{3.3}$$

The main effect of this algorithm is to move the weight vector directly toward the input vector. Accordingly, if the weight vector and input vector are already congruent, no change will be made and the weight vector will top out.

So far we have considered the behavior of the network when it has but one pattern to represent. Suppose now that it is given many different input vectors and hence that there are many different patterns to be represented. How will the net manage? When there are fewer output units than input vectors, each output unit will become specialized for clusters of overlapping input vectors, as illustrated in figure 3.27. In this way, the network will tend to develop output units that are sensitive to features common to its preferred input vectors, with each output specializing for a different particular feature. Consequently, what any given unit can be said to represent is a *prototype* of the range of nonidentical but overlapping vectors that turn it on.

There are three general weaknesses with networks of this type so far as adequate representation of patterns is concerned. First, sometimes critical information in a pattern may not correspond to the most frequently occurring feature, and so may fail to be represented by the net. A bottom creature which represents predators as looming shadows may thus be fooled by a predator with a thin dangling stinger, and this could be trouble if the predator is especially deadly, however rare. The second weakness is that this procedure picks out the lowest-order features, but it may be the higher-order features such as those that characterize the differences between faces that are critical for classification. Finally, relational invariances such as rotation, dilation, and translation need to be extracted before the patterns can be compared. Strategies for overcoming these drawbacks will be considered in chapter 4.

The third difficulty concerns stability of the weight configurations. The weights may shift even when the input is relatively familiar but the order of input vectors varies, and the instability problem is yet more acute should the network be given novel input vectors. In the real world, some forgetting may be advantageous, but on the other hand, it is often essential that previous learning not be wiped out by new encounters. Some provision needs to be made to retain relevant and important aspects of what has already been learned.

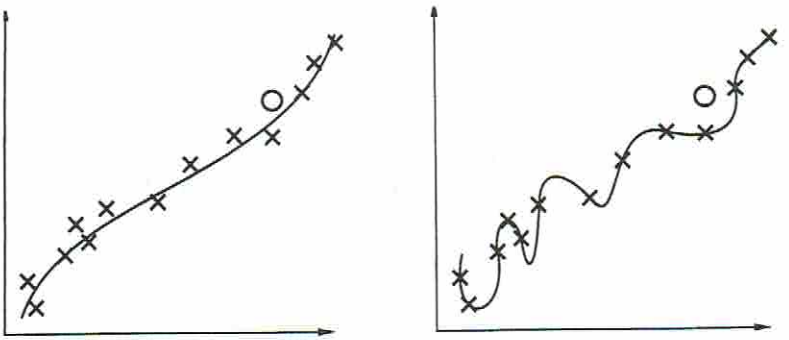

Figure 3.28 Curve fitting and overfitting. The data points are given (X) and the goal is to pass a curve through them. It is known that the data contain noise. A smooth curve (left) does a better job of predicting a new data point (O) than does a kinky curve (right). The degree of smoothing and the choice of interpolating function depend on the data and are central issues in approximation theory.

while learning new things. How nervous systems manage to do this is not understood, but one solution for artificial networks, explored by Carpenter and Grossberg (1987), is to add new units when novel inputs are encountered. There are many variations of the competitive network theme, including generalizing the basic architecture to multilayered networks (Fukushima 1975).

7 CURVE FITTING

The classic example of fitting parameters to a model is curve fitting, that is, fitting a host of noisy data points with a smooth function, using the least squares method (figure 3.28) (i.e., minimize squared error for the whole set). For a fit with a straight-line function, the squared error E is given by this equation:

$$E(m, b) = \tfrac{1}{2} \sum_{i}^{N} [mx_i + b - y_i]^2 \tag{14}$$

where m is the slope, b is the intercept with the y-axis, and there are N data points, (x_i, y_i). When the error is at the minimum, the gradient of E with respect to the parameters (m and b) $= 0$. That is,

$$\text{for m:} \quad \frac{\partial E}{\partial m} = \sum_{i}^{N} (mx_i + b - y_i)x_i = 0$$
$$\text{for b:} \quad \frac{\partial E}{\partial b} = \sum_{i}^{N} (mx_i + b - y_i) = 0 \tag{15}$$

These are two simultaneous equations with two unknowns. This is a relatively easy problem. Nevertheless, as the number of dimensions of the state space increases, and the number of parameters needed to fit the data increases, then the problem becomes much more difficult. Thus we may be looking for a curve, not in a *2-D* space, but in a *10-D* or *100-D* or *10,000-D* space. The traditional solution consists in solving the equations algebraically, which is quite manageable so long as the number of parameters and the number of data points are

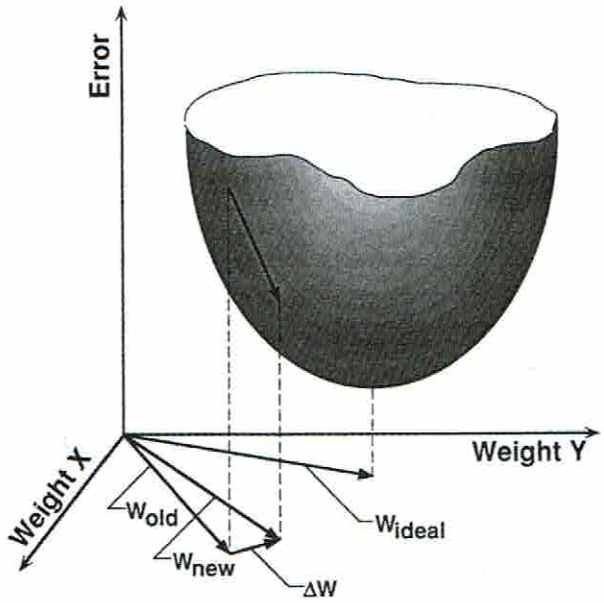

Figure 3.29 Error surface and gradient descent. The goal is to find the set of weights that gives the minimum error. For a given set of parameters (shown here for two weights plotted in the $x-y$ plane) the gradient of the error surface is computed to find the direction of steepest descent. The weights are incrementally changed by ΔW along this direction, and the procedure is repeated until the weights reach W_{ideal}, which gives minimum error. For a nonlinear network the error surface may have many local minima.

small. As soon as either is large, then the algebraic method is nasty and unmanageable. Fortunately, there is a way, other than the traditional algebraic technique, to find the solution. This involves using knowledge of the gradient to iteratively update the estimates for the parameters.[27] This is essentially like iteratively updating weights in a net, and can be given geometric representation (figure 3.29). In iterative curve fitting we update according to this rule:

$$\Delta m = -\varepsilon \frac{\partial E}{\partial m}$$

$$\Delta b = -\varepsilon \frac{\partial E}{\partial b}$$

where Δm is the change in m, Δb is the change in b, and ε is the learning rate.

Instead of computing the exact gradient, the parameters can be adjusted after every sample, or after averaging a few samples. This speeds up convergence, which can be guaranteed if the learning rate, ε, approaches zero sufficiently slowly. This procedure is called gradient descent because at every step the parameters are changed to follow the gradient downhill, much as a skier might follow the fall line. We encountered gradient descent earlier in the context of Boltzmann learning procedures, where iterative application of the weight update rule gradually led us to better performance.[28]

To apply gradient descent, one must have a mathematically well-defined measure, such as mean squared error, to optimize, as well as an efficient way to calculate the gradients. Models that are dynamical and have many parameters have to be represented in a many-dimensional state space. Accordingly, the amount of computer time to perform the calculations may be astronomical, so it is essential to find efficient ways of computing. Exploring such procedures for some classes of models, such as feedforward nets and recurrent nets with linearly summing weights and nonlinear input–output functions led to important breakthroughs in the 1980s. (See section 8.)

When the curve-fitting task is fitting a *straight line*, gradient descent guarantees convergence to the global minimum of the error function. By contrast, there is no such guarantee for the general nonlinear problem, where the error surface may have many local minima, though if one is lucky, they will be close enough to the global minimum that it will not matter much if the net stops in one. For most of the problems presented in the later chapters, finding the true global minimum is unnecessary, and there are many equally good local minima. Thus if a net is started running with its initial weights randomly set to small values, it is likely to end up with one of these good solutions.

8 FEEDFORWARD NETS: TWO EXAMPLES

In feedforward networks the input leads directly to an output without feedback. Because feedforward nets have significant advantages in speed and simplicity, it is worthwhile exploring what computations a feedforward network can handle. In this section, we show that a feedforward network with one layer of weights cannot compute an extraordinarily simple function. Understanding the whys and wherefores of the limitations is instructive, and yields insights into the geometrical nature of feedforward networks. This is important, because it is the geometry of a network that determines what can be represented and how things can be represented; in appreciating that, we can see how to overcome these limitations.

Exclusive "or"

The look-up table for a function called the exclusive "or" (XOR) was introduced earlier. The question now is whether this very function can be executed by a net; more specifically, the question concerns what sort of network architecture is necessary to execute this function. Finding a suitable architecture turns out to be instructive because of the failures and what they imply for a whole range of functions. To milk the lesson from the failures, we consider first the archetypal simple net. It consists of one output unit, taking value 0 or 1 (representing "false" and "true"), and two input units that carry the values of the component propositions.[29] This simple net has three free parameters: the two weights from the input units to the output units, and a single threshold or bias on the output unit. The architecture of this net determines the function for the output:

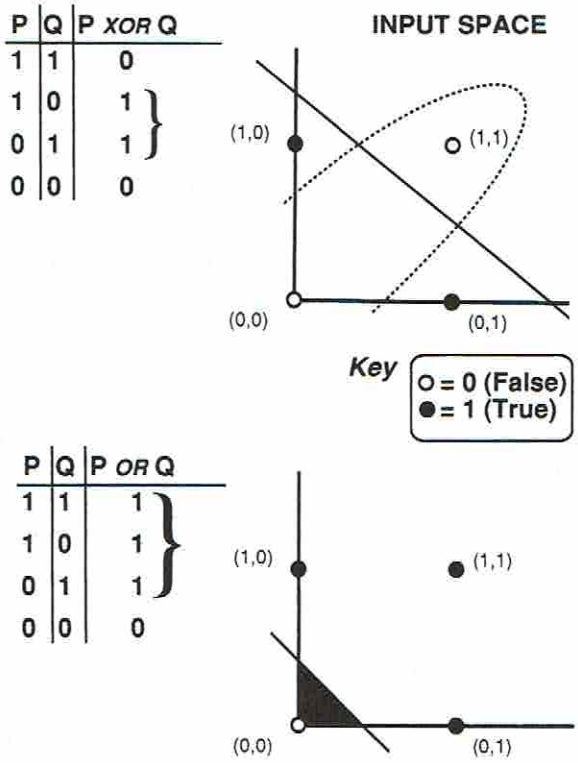

Figure 3.30 *XOR* is not a separable function. The truth table for *XOR* is shown on the upper left and is plotted on the upper right. No line in the plane can separate all of the closed dots (true, or 1) from all of the open dots (false, or 0). In contrast, the OR function, shown below, can be separated by the line shown in the bottom right. A network with one layer of weights can be found to represent a function if it is separable, but no such network can be found if the function is nonseparable, like *XOR*.

$$o = H(w_P P + w_Q Q + b) \tag{17}$$

where o is the output, H(input) is a binary threshold function (figure 3.19), w_P and w_Q are the weights for the binary inputs P and Q respectively, and b is the bias.

Does there exist any configuration of weights and the bias such that the output unit correctly assigns values to the compound? For this simple net, the answer is "no," and figure 3.30 (top) illustrates why. The input space has an axis for each of the two units, and points representing the four possible input vectors. If the output unit can solve the problem, it must be able to group together the inputs that give 1, and in another group, the inputs that give 0. Geometrically speaking, the function restricts the border between these two regions of the input space to a straight line. On one side of the decision border, all inputs drive the output over threshold, giving 1; on the other side, the inputs are below threshold, giving 0. No such straight line exists.

For contrast, the output unit of the simple net can execute the inclusive "or" (*OR*), and the input space for the same network architecture clearly does admit

of a straight line appropriately dividing the inputs (figure 3.30, bottom). Why does the decision border have to be straight? Because the architecture of the net is consistent only with a linear function, and hence limits how the inputs can be segregated for training to give the correct output. In other words, the exclusive "or" is not a *linearly separable function*, and hence no learning algorithm can find a solution to the weight configuration problem in the simple net because none exists. Intuitively, this can be seen by reflecting on the logic of the *XOR*. Analyzed in English, it comes out thus: (*P XOR Q*) is true if and only if either *P* is true or *Q* is true, *but not both P and Q are true*. It is this extra twist on the back end of the basic OR function that needs to be accommodated. The point is that we need take the output of the more basic OR function, and operate on it again to get a higher-order property. Can the simple net be embellished to handle what is in effect a function on the output of a function? Yes, and the modification is both blindingly obvious in retrospect and was frustratingly opaque in prospect.

The crucial modification consists in adding an intervening unit—a hidden unit—that is interposed between the inputs and the outputs. This supplements the weights by three, and adds another bias unit. With a new total of seven parameters, the new question is this: are there weight-settings such that the net will solve the problem? This time the answer is "yes," for the role of the hidden unit is to handle the second-order operation; that is, it gets its input from *P* and *Q*, and recognizes when not both *P* and *Q* are true. In fact, more than one weight configuration will suffice to do the job. The next question, of course, concerns automated training: is there a gradient-descent procedure for adjusting the weights? Insofar as the units are binary, gradient descent cannot be used to adjust the weights between the input units and the hidden unit. The trouble is that small changes to these weights will have no effect on the output, save in the case where it is close to the threshold of the hidden unit.

In the 1960s, model net research had developed to the point where the need for hidden units to handle nonlinearly separable functions was understood, but how to automate weight-setting, especially for the hidden units, was not (Rosenblatt 1961, Minsky and Papert 1969). As we saw earlier, automated adjustment of hidden unit weights was achieved in the Boltzmann machine in the early 1980s. How to do this for a feedforward network remained baffling. In 1986, Rumelhart, Hinton, and Williams, discovered[30] that the trick is to push the output of each hidden unit through a squashing function, that is, a smoothly varying function that maps the hidden units' input onto output (figure 3.31). Hitherto, the output from the hidden units was a step function of the input. The smoothly varying function, however, means that small changes on the weights of the hidden unit's inputs allow the hidden unit to abstract the higher-order property in small error-correcting steps, and hence to learn to recognize when both *P* and *Q* are 1. The net effect of adding hidden units and putting their output through a squashing function is that the input space can now be divided by a curvy decision border.

The backpropagation algorithm starts with small, randomly chosen weights and proceeds incrementally, just as in the earlier curve-fitting example. The

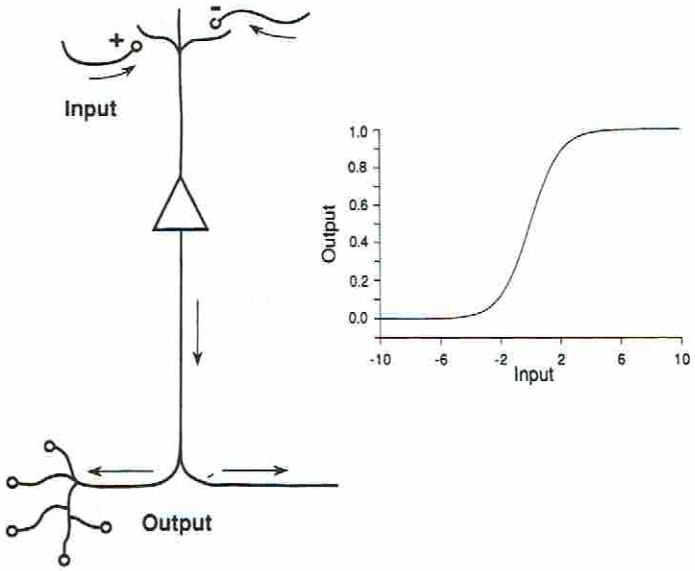

Figure 3.31 Nonlinear squashing function for a processing unit. The inputs are weighted and summed, and a bias is added (or threshold subtracted) before passing the total input through the sigmoid, shown on the right. The output of the unit is close to zero for large negative inputs, has a roughly linear region around zero input, and saturates for large positive inputs. This type of nonlinearity characterizes the firing rate of some neurons, such as motor neurons, as a function of the injected current from synaptic inputs on its dendrites, as shown on the left. Neurons also have complex temporal properties that are not captured by this static form of nonlinearity.

main difference is that the error surface for least-square curve fitting has a single minimum (figure 3.29), whereas the error surface for a network with hidden units may have many local minima. Consider a feedforward network such as the one in figure 3.32 with units having a nonlinear squashing function $\sigma(x)$ as in figure 3.31:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{18}$$

For an input pattern that produces an output value o_i, the error is defined as

$$\delta_i(\text{output}) = (o_i^d - o_i)\sigma_i'(\text{output}) \tag{19}$$

where o_i^d is the desired value of the output unit provided by a "teacher," and $\sigma'(output)$ is the derivative of the squashing function. This error can be used to modify the weights from the hidden layer to the output layer by applying the delta rule[31]:

$$\Delta w_{ij} = \varepsilon \delta_i(output) h_j \tag{20}$$

where h_j is the output from the jth hidden unit. The next problem is to update the weights between the input units and the hidden units. The first step is to compute how much each hidden unit has contributed to the output error.[32] This can be done by using the chain rule. The result is:

$$\delta_j(hidden) = \sigma_j'(hidden) \sum_i w_{ij} \delta_i(output) \tag{21}$$

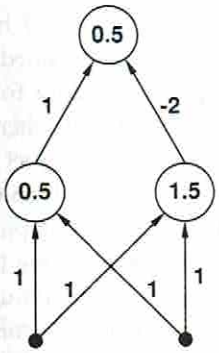

Figure 3.32 *XOR* network. The weights are shown on the connections (lines), and the thresholds are shown inside the units (circles). Thus, an input of ⟨0, 1⟩ produces a pattern of ⟨1, 0⟩ on the hidden units, which in turn activate the output unit. An input of ⟨1, 1⟩ produces ⟨1, 1⟩ on the hidden units, and the output unit will be turned off. In effect, the right hidden unit becomes a feature detector for the ⟨1, 1⟩ pattern and overrides the influence of the left hidden unit. Other solutions are possible.

Once we know the error for each hidden unit, the same delta rule used for the output units (eq. 20) can be applied to the weights from the input layer to the hidden layer. This procedure (backpropagation of error and suitable weight modification) can be applied recursively through arbitrarily many layers of hidden units. As a practical point, the gradients for the weights can be accumulated for several patterns and the weights updated according to the average gradient.[33]

Let us now shift gears and think about this in terms of an error surface. Think of the error surface as a state space where the vertical axis represents percentage of error, and each of the remaining axes represents a weight in the system (figure 3.29). As the weights change, the position in error space will change. The error surface for fitting straight lines to data by minimizing the squared error is a concave basin with a single minimum. In contrast, the error surface of the *XOR* network with hidden units has a more complex error surface, one whose topography has ravines and assorted potholes known as local minima. If, however, the net starts out with small, randomly chosen values for the seven parameters, adjusting each weight to minimize the error over the set of four input–output conditions, then the system eventually ends up at a set of parameters that truly solves the problem (figure 3.32). In fact, there are many combinations of parameters that will do equally well, and depending on the random starting place, the system will find one or the other. If the initial weight-settings are too large, then the net may land in a local minimum that is not a solution to the problem, so the lore is to start the weights low. Although the backpropagation of error can be used to create networks that can solve the *XOR* problem, that problem is sufficiently simple that many other techniques can also be used to solve the problem.

Many problems resemble XOR in the respect that they are not linearly separable; indeed, one might say that most interesting computational problems

have that property. The discovery concerning the role of hidden units in extracting higher-order features and the versatility obtained by squashing functions on their outputs have therefore opened the door to network solutions to many complicated problems.[34] Even when a researcher does not have a clue what function maps input onto output, he may construct a network that does solve the input–output problem, whereupon he may then work backward to track down the function the network has learned to compute, though this may not always be a simple matter. In the next section, we illustrate an instance where a network was successfully trained even though the input–output function executed by the trained-up net was quite unknown to the modelers.

Finally, the *XOR* problem may seem pedagogically germane but biologically superfluous. Despite appearances, this turns out to be a hasty judgment, for *XOR* nets can be iterated into systems that have unexpectedly useful properties from a biological point of view. As we shall see in chapter 4, the basic *XOR* net is the electronic equivalent of a "gear" in the sense that it admits of many variations, and many *XOR* nets can be assembled into one large net that can solve very complex problems. Notice too that the negation of (*P XOR Q*) means the same as (*P* if and only if *Q*), and thus training an interconnected array of negated XOR (NXOR) units is a way of finding necessary and sufficient conditions on certain higher-order representations.

Having shown that *XOR* can be represented and learned in a feedforward network, we now consider what other functions a feedforward net might compute. The surprising answer is that a feedforward net with a sufficiently large number of hidden units can be trained to approximate with an arbitrarily small error any mathematically well-behaved function (White 1989). To be sure, this is a reassuring theoretical result, but what does it really mean in practical terms? Networks with hundreds of hidden units and hundreds of thousands of weights have been trained successfully on a wide range of problems. In the next section, we present one example to illustrate the general approach. One important practical consideration, however, concerns how much computer time and how many examples a net needs to learn a function as the number of hidden units becomes very large. This is the *scaling* problem, and we shall return to this difficulty later in the chapter.

Discriminating Sonar Echoes

Consider a feedforward net trained up by the backpropagation of error method to distinguish between sonar echoes of rocks and sonar echoes of mines (Gorman and Sejnowski 1988a, b) (figure 3.33). This is, in fact, a rather difficult problem because to the untrained ear, at least, no difference is discernible. The net has an input layer with 60 units, a hidden layer with 1–24 units, and two output units. To prepare the sonar echoes for input to the net, a given sonar echo is run through a frequency analyzer and is sampled for its relative energy level in 60 different frequency bands. These 60 values, normalized so that 1 is maximum, are then entered as activation levels in the respective input units

12 MODELS: REALISTIC AND ABSTRACT

In the previous section we leaned rather heavily on the desirability of constraining a model net with neurobiological data to increase the probability that the model can tell us something about the real neural net. There is an important dimension in which that must be qualified and explained. In neuroscience, as anywhere else in science, no model is 100% realistic. A good and useful model of the solar system, for example, will not necessarily have actual gas clouds drifting around Jupiter; a good and useful model of magnetism will not necessarily provide for the rusting of iron. The central point is this: what goes into the model depends on what one is trying to explain, and in the nervous system, that is intimately related to the level of organization one is targeting. (Recall levels of organization in chapter 2, and figure 2.1.) A little more exactly, if one is modeling a function or task of a given level of brain organization, the model should be sensitive to structural constraints from the level below and to input–output properties described for the level above.

Not uncommonly, a model will be criticized as unrealistic for failing to include very low-level properties. A model of a neural circuit such as the vestibulo-ocular reflex (VOR) (see chapter 6) may include only the pathways required by the model to perform equivalently to the real net, it may average over dendritic integration, and it may ignore altogether the details of membrane channel properties. Does this mean the model is too unrealistic to be useful? We shall show more directly and specifically in chapter 6 why the model can be useful nonetheless, but for the moment, suffice it to say that lower-level properties such as channel properties are not necessary to simulate a neuron's contribution to that aspect of the VOR up for explanation, namely image stabilization. The model certainly needs to incorporate constraints regarding latencies and feedback loops (level above) and constraints regarding the net effect at synapses and the firing rate of neurons in the circuits (level below), but it does not need to be sensitive to the precise mechanisms by which a neuronal membrane operates to yield these firing rates (*two* levels below). There may be other aspects of the VOR circuit where these properties will be required for the explanation sought—for example, exactly how synaptic plasticity is managed. When synaptic plasticity *is* what is being modeled, however, then these properties should be included. More generally, if one were modeling how a neuron integrates signals, then membrane properties are relevant and must be included. In that event, however, higher-level properties such as receptive field and feedback connections (two levels up), are probably not relevant.

Some modeling discussions seem to presuppose a kind of "realism pecking order." For example, it may be argued that until the whole neuron is thoroughly and completely modeled, modeling even a small circuit such as the VOR is premature. According to this argument, a circuit model will have to idealize the neuron, leaving out such details as the membrane channel types and their physiology. This, it will be complained, makes the model "unrealistic," and hence dismissable. An even more pure realist can dismiss the whole

neuron-modeling project on grounds that there is no point in modeling the neuron until one has completely modeled the dynamics of transmitter release, including such constraints as numbers of vesicles that fuse, spatial layout of receptors, and production and packaging of neurotransmitters. Undoubtedly the biophysicist can top that; he wants first a model of protein folding. But this is surely silly. Do we really need a model of protein folding in order to get a grip on the essentials of how the VOR achieves stabilization of a visual image during head movement? Part of what perpetuates the realist pecking order is this: any given modeler tends to think his favored modeling level is *the* important level, that lower levels are properly ignorable, and modeling levels higher than his favored level is premature and unrewarding.

Realist one-upmanship needs to be put in perspective. First, models that are excessively rich may mask the very principles the models were built to reveal. In the most extreme case, if a model is exactly as realistic as the human brain, then the construction and the analysis may be expensive in computational and human time and hence impractical. As noted in chapter 1; modeling may be a stultifying undertaking if one slavishly adheres to the bald rule that the more constraints put in a model, the better it is. Every level needs models that are simplified with respect to levels below it, but modeling can proceed very well in parallel, at many levels of organization at the same time, where sensible and reasonable decisions are made about what detail to include and what to ignore. There is no decision procedure for deciding what to include, though extensive knowledge of the nervous system together with patience and imagination are advantageous. The best directive we could come up with is the woefully vague rule of thumb: make the model simple enough to reveal what is important, but rich enough to include whatever is relevant to the measurements needed.

13 CONCLUDING REMARKS

The look-up table was introduced as our first and simplest parade case of computation. Because look-up tables appeared to be rather limited in their capacity to solve difficult computational problems, other computational principles were sought. It is surprising then to realize that a trained-up network model can be understood as a kind of look-up table. Once the parameters are set, the network will give the output suitable to the input, and the answer to any given question is stored in the configuration of weights. Not, to be sure, the way answers are stored on a slide rule, but in the sense that an input vector pushed through the matrix of weights (and squashing function) is transformed so that the output vector represents the answer. Not, notice, by way of many intervening computational steps, but merely by vector-matrix transformation.

By analogy with the ⟨board position/next position⟩ pairs prestored in the Tinkertoy computer, the net's weight configuration, characterized as a matrix through which the input vector is pushed, can be thought of as "storing" prototypes as ⟨input vector/hidden unit activation⟩ pairs, "storing" being in scare quote to reflect the extension of its customary sense. As we saw above, when the net has learned to distinguish rock echoes from mine echoes so no

further weight-adjusting is required to deliver correct answers, the weights partition the activation space of the hidden units in such a way that activation values of the hidden units fall on one side or the other of the partition (see next chapter, figure 4.17). The key difference between the run-of-the-mill look-up table and the trained-up network is that the network can correctly classify novel signals and thus has the capacity to generalize from training cases to new cases. In this respect, trained-up networks have a flexibility denied to run-of-the-mill look-up tables. This flexibility of a net is not mysterious. It derives from the net's assorted design features; for example, the output values of its hidden units may be continuous, and working as groups the units can interpolate smoothly between samples.

These networks might be considered examples of "smart look-up tables." They operate in very high-dimensional spaces, since weight space will have as many dimensions as there are weights, and activation space for the hidden units will have as many dimensions as there are hidden units. The consequence of these design features is that though the training set is finite, the network will give good answers to inputs it has never seen before. The inputs will, however, have some resemblance to previously seen inputs, and that will be enough to ensure correct categorization. It must be emphasized that the learning process is not itself a look-up table operation—rather, parameter-adjusting is a relaxation process. Similarly, fitting the Tinkertoy pieces together is not itself a look-up table operation. In the network, it is only the *result* of parameter-adjusting procedures that produces something construable as having a look-up table configuration. Moreover, a single (smart) look-up table can be traded in for a hierarchy of (smart) mini-look-up tables, so that an approximate answer can be cranked out at one stage, then shunted to a further stage for finer tuning. In this event, speed is traded for spatial miniaturization. Some of the speed can, however, be bargained back by adopting parallel search.

The insight that trained-up nets are look-upish raises the question of whether this might be useful in understanding circuits in nervous systems. Is it possible that some parts of the brain are taking advantage of the look-up table principle in some highly evolved version of that genre? Time considerations suggest they might well be. The time delays for conduction of a signal down an axon and across a synaptic cleft, together with the time delays for signal integration in the dendrites and cell body, add up to about 5 or 10 msec per neuronal step. If a nervous system is to give a motor response to a sensory stimulus with a latency of a few hundred milliseconds or less, then for certain stages in processing, the neuronal anatomy is probably configured for look-up short-cuts. Visual pattern recognition, for example, can be done in about 200–300 msec, which means there is only time for about 20–30 neuronal steps from retinal stimulation to motor output (Thorpe and Imbert 1989, Thorpe 1990).

Evidently the response latency for many tasks, including visual recognition, shows that there is insufficient time for the brain to be engaged in following the massive set of 3000–50,000 steps (or more) found in conventional computer vision programs (Feldman and Ballard 1982). Figuring out the next move

in chess or figuring out how to make a bridge with popsicle sticks is, by contrast, relatively slow, and clearly involve many steps, but whether these are steps in a long sequence of look-ups representing possible ⟨move-next move⟩ pairs, or whether some rely on different principles altogether remains to be seen. Given their finite capacity, nervous systems cannot store answers for every possible contingency. To cope with novelty, nervous systems must cycle through multiple states to search for an adequate solution. With practice, however, the new problem–solution pairs can be compiled into a look-up table.[41]

Selected Readings

Abu-Mostafa, Y. (1989a). Complexity in neural systems. Appendix D of Mead (1989), *Analog VLSI and Neural Systems*. Reading, MA: Addison-Wesley.

Arbib, M. (1987). *Brains, Machines, and Mathematics*. Berlin: Springer-Verlag.

Arbib, M. A., and J. A. Robinson, eds. (1990). *Natural and Artificial Parallel Computation*. Cambridge, MA: MIT Press.

Carbonell, J. G., ed. (1989). *Artificial Intelligence* (special volume on Machine Learning), vol. 40, nos. 1–3; see especially the chapter by Geoffrey Hinton on connectionist learning.

Durbin, R., C. Miall, and G. Mitchison (1989). *The Computing Neuron*. Reading, MA: Addison-Wesley.

Hanson, S. J., and C. R. Olson, eds. (1990). *Connectionist Modeling and Brain Function: The Developing Interface*. Cambridge, MA: MIT Press.

Haugeland, J. (1985). *Artificial Intelligence: The Very Idea*. Cambridge, MA: MIT Press.

Hertz, J., A. Krogh, and R. G. Palmer (1991). *Introduction to the Theory of Neural Computation*. Redwood City, CA: Addison-Wesley.

Johnson-Laird, P. (1988). *The Computer and the Mind*. Cambridge, MA: Harvard University Press.

Kelner, K., and D. E. Koshland, eds. (1989). *Molecules to Models: Advances in Neuroscience*. Washington, DC: American Association for the Advancement of Science. (See especially chapter V, Neural Modeling.)

Kohonen, T. (1987). *Content-Addressable Memories*, 2nd ed. Berlin: Springer-Verlag.

Lippmann, R. P., J. E. Moody, and D. S. Touretzky (1991). *Advances in Neural Information Processing Systems 3*. San Mateo, CA: Morgan Kaufmann.

McClelland, J., and D. Rumelhart (1988). *Explorations in Parallel Distributed Processing*. Cambridge, MA: MIT Press. Comes with computer programs that simulate many of the networks discussed in this chapter.

McClelland, J., D. Rumelhart, and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 2. Cambridge, MA: MIT Press.

Mead, C. (1989). *Analog VLSI and Neural Systems*. Reading, MA: Addison-Wesley.

Miller, K. D., guest ed. (in press). *Seminars in the Neurosciences*. Vol. 4, no. 1. *Use of Models in the Neurosciences*.

Nilsson, N. J. (1990). *The Mathematical Foundations of Learning Machines*. San Mateo, CA: Morgan Kaufmann. Reprint of the first edition (1965) with a new introduction by T. J. Sejnowski and H. White.

Poggio, T. (1990). A theory of how the brain works. In *Cold Spring Harbor Symposium on Quantitative Biology: The Brain*, vol. 55, ed. E. Kandel, T. Sejnowski, C. Stevens, and J. Watson. 899–910.

Rumelhart, D., J. McClelland, and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1. Cambridge, MA: MIT Press.

Schwartz, E. L., ed. (1990). *Computational Neuroscience*. Cambridge, MA: MIT Press.

Touretzky, D. S., ed. (1989). *Advances in Neural Information Processing Systems*, vol. 1. San Mateo, CA: Morgan Kaufmann.

Touretzky, D. S., ed. (1990). *Advances in Neural Information Processing Systems*, vol. 2. San Mateo, CA: Morgan Kaufmann.

Wasserman, P. D., and R. M. Oetzel (1990). *NeuralSource: The Bibliographic Guide to Artificial Neural Networks*. New York: Van Nostrand Reinhold.

Widrow, B., and S. D. Stearns (1985). *Adaptive Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall.

Selected Journals

Neural Computation: a bimonthly journal published by MIT Press; contains reviews, articles, views, notes, and letters on the theoretical principles of neural circuits from the biophysical level to the systems level.

Network: Computation in Neural Systems: A physics-oriented journal on biological and artificial neural networks. Bristol, UK: IOP Publishing Ltd.

IEEE Transactions on Neural Networks: Engineering-oriented journal on artificial neural networks. New York: Institute of Electrical and Electronic Engineers, Inc.

Neural Network: Official journal of The International Neural Network Society. New York: Pergamon Press.