LEARNING IN PARALLEL NETWORKS

BY GEOFFREY E. HINTON

Simulating learning in a probabilistic system

THE BRAIN is an incredibly powerful computer. The cortex alone contains over 10^{10} neurons, each connected to thousands of others. All of your knowledge is probably stored in the strengths of these connections, which somehow give you the effortless ability to understand English, to make sensible plans, to recall relevant facts from fragmentary cues, and to interpret the patterns of light and dark on the back of your eyeballs as real three-dimensional scenes. By comparison, modern computers do these things very slowly, if at all. They appear very smart when multiplying long numbers or storing millions of arbitrary facts, but they are remarkably bad at doing what any five-year-old can.

One possible explanation is that we don't program computers suitably. We are just so ignorant about what it takes to understand English or interpret visual images that we don't know the appropriate data structures and procedures to put into the machine. This is what most people who study artificial intelligence (AI) believe, and over the last 20 years they have made a great deal of progress in reducing our ignorance in these areas.

Another possible explanation is that brains and computers work differently. Perhaps brains have evolved to be very good at a particular style of computation that is necessary in everyday life but hard to program on a conventional computer. Perhaps the fact that brains store knowledge as connection strengths makes them particularly adept at weighing many conflicting and cooperating considerations very rapidly to arrive at a common-sense judgment or interpretation. Of course, any style of computation whatsoever can be *simulated* by a digital computer, but when one kind of machine simulates a very different kind it can be very slow. To simulate all the neurons in a human brain in real time would take thousands of large computers. To simulate all the arithmetic operations occurring in a Cray would take billions of people.

It is easy to speculate that the brain uses quite different computational principles, but it is hard to discover what those principles are. Empirical studies of the behavior of single neurons and their patterns of connectivity have revealed many interesting facts, but the underlying computational principles are still unclear. We don't know, for example, how the brain represents complex ideas, how it searches for good matches between stored models of objects and the incoming sensory data, or how it learns. In this issue, Jerome A. Feldman describes some current ideas about how parallel networks could recognize objects (see "Connections" on page 277). I will describe one old and one new theory of how learning could occur in these brain-like networks. Please remember that these theories are extreme idealizations; the real brain is much more complicated.

ASSOCIATING INPUTS WITH OUTPUTS

Imagine a black box that has a set of input terminals and a set of output

(continued)

Geoffrey E. Hinton is an assistant professor of computer science at Carnegie-Mellon University. He can be reached at the Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213.

terminals. Each terminal can be clamped into either of two states, active or inactive (1 or 0). We can show the black box what we would like it to do by repeatedly clamping a combination of 1s and 0s on the input terminals and another combination of 1s and 0s on the output terminals (each combination is called a *vector*). When we have done this for many I/O (input/output) pairs, we would like the black box to automatically set its output terminals into the correct state when we clamp a vector on the input terminals. Ideally, if there is some neat regularity in the mapping from input vectors to output vectors, we would like the black box to "capture" this regularity in its internal connection strengths in order to give the "correct" output vector for input vectors it has never seen before. This kind of black box would be a very useful module to have within an intelligent system.

If the black box contains only direct connections from input terminals to output terminals, there is a beautifully simple learning procedure that adjusts the weights on these connections until every input vector causes the appropriate output vector. The learning procedure has two phases that keep alternating. In phase 1, we clamp an input vector on the input terminals and an output vector on the output terminals. Then we increment by a small amount, δ, the weights of all connections that have both their input and output terminals active. In phase 2, we clamp the same input vector, but we let the black box decide for itself what output vector to produce, using the rule that an output terminal turns on if the sum of the weights on its connections from active input terminals is positive. We then *decrement* by δ all the connections that have both their input and output terminals active. If the network produces exactly the right output, these decrements will exactly undo all the increments we made in phase 1, because exactly the same pairs of input and output terminals will be active in the two phases. If, however, the network produces the wrong output in phase 2, some of the weights that were incremented will not be decremented or vice versa, so some weights will change.

The learning procedure I have described is a version of the Widrow-Hoff or "perception convergence" procedure. It has a remarkable property: If we keep cycling through all the pairs of input and output vectors using this two-phase procedure for each pair, we will converge on a set of weights that causes the right output vector for every input vector *if any such set of weights exists*. The big disappointment (which led people in AI to abandon this kind of model) is that for most interesting problems there is *no* suitable set of weights. The relationship between the input and output vectors is just too complicated to be captured by a system that has direct connections between input and output terminals. At the very least, there must be intermediate layers within the black box, and units in these layers must learn to extract a hierarchy of "features" of the input vector that can eventually cause the right output.

Here is an example of a relatively simple task that requires intermediate units. The input consists of two 8-bit vectors, one of which is a shifted version of the other. Only shifts one place to the left or one place to the right are allowed. There are two output terminals, one for each possible shift, and the black box must turn on the correct output terminal for any appropriately related pair of input vectors. The task sounds easy until you consider that any one of the input bits, considered in isolation, provides *no* information about what the output should be. Moreover, simply adding up evidence from all the separate in-

(continued)

put bits is useless. The task can be done only if you consider combinations of bits in one vector with bits in the other, which requires intermediate units that extract informative combinations. Figures 1a and 1b show a collection of useful intermediate feature detectors that work well together for performing this task.

When we try to extend the simple learning procedure to networks containing intermediate units, more complications arise because we do not know in advance how we want the intermediate units to behave. So instead of just fixing some weights that will make the output terminals behave in the way that we specify, the learning algorithm must also decide under what circumstances each of the intermediate units should be active. This amounts to *creating* intermediate representations. Several more recent learning procedures can do this. I shall describe one that Terry Sejnowski and I discovered. It is only guaranteed to work in networks of a rather special kind, which I will now describe.

Networks That Minimize Their Energy

The kind of network we have been considering so far consists of layers of units in which units in one layer are connected to units only in contiguous layers. More complex networks have cross-talk within a layer and feedback from later layers to earlier ones. It is generally very hard to analyze the behavior of such networks, but John Hopfield at Cal Tech (reference 1) has shown that there is an interesting special case that behaves in a very useful way. In a Hopfield net, the units make their decisions asynchronously, the communication between units is instantaneous, and all the connections are symmetrical: the effect of unit *i* on unit *j* is the same as the effect of unit *j* on unit *i*. Given these restrictions, the various possible states of the whole network form a space like a bumpy surface and the current state of the network behaves like a ball bearing placed on this surface—it moves downhill into the nearest *local minimum*. Each point in the surface corresponds to a pattern of active and inactive units in the network, and the height of the surface at that point represents the "energy" of that pattern of activity, where the energy of a pattern is defined as minus the sum of all the weights on connections between pairs of active units. Therefore, if two units have a big positive weight between them, patterns in which they are both active will have low energy; it is patterns like this into which the network will settle. Conversely, a negative weight between two units will make a big positive contribution to the energy when they are both on, so the network will tend to avoid such states.

Figure 2 shows a small network whose lowest energy state is -8. Can you figure out which units are on and which are off in this state? You will always end up at an energy minimum if you start with a random state and then apply the following rule to each unit in turn (in any order): If the sum of the weights on the connections to other currently active units is positive, turn it on; otherwise, turn it off. If you apply this procedure a few times, you will discover that there is another minimum with an energy of -3 and that once the network has settled into this state it will just stay there.

Networks of this type can be used to associate input vectors with output vectors. To provide the input, we clamp a subset of the units into their

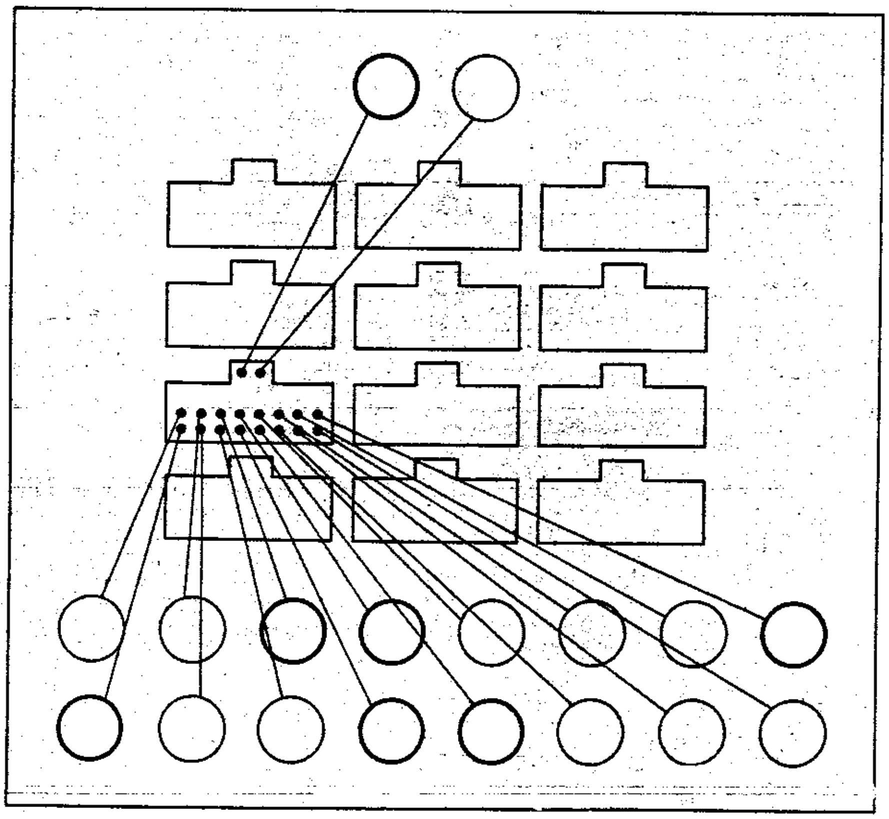

Figure 1a: A *network with* 16 *input terminals,* 12 *intermediate units, and* 2 *output terminals. The boldface units show which bits are on in a typical pair of input and output vectors. The 8-bit vector at the bottom has been shifted one place to the left (with wraparound) to produce the 8-bit vector immediately above it. With these two vectors as the input, the correct output vector (shown at the top of the figure) has the left unit active to represent a left shift. Each of the 12 intermediate units is connected to all the input and output units, but only one set of connections is shown. The intermediate units also have a fixed threshold, which is subtracted from their net input before the decision is made to turn them on or off.*

on or off states, and, once the rest of the network has settled into an energy minimum with this input vector clamped, we treat the states of another subset of the units as the output. In figure 2, for example, we could clamp the three bottom units into the active state to represent the input vector (1, 1, 1); we would get the output vector (1, 0) by letting the network settle and then reading the states of the top two units.

To teach the network a particular set of I/O pairs, we would need to create an appropriate energy landscape—we would need to choose weights so that for each clamped input vector the system had an energy minimum that yielded the correct output vector. Choosing such weights is not an easy task, and to make matters worse, we might end up with an energy landscape in which there were many different local minima for each clamped input; each input vector might give many different outputs depending on the energy minimum into which the system happened to settle. In figure 2, for example, the input vector (0, 0, 0) can generate two different output vectors depending on the initial states of the middle units and the order in which decisions get made.

A Probabilistic Network

If the same input is going to produce different outputs on different occasions, we would at least like to have some control over the probabilities. It would be nice, for example, if we could guarantee that deeper minima would be found more often than shallower ones. It would be even better if we could guarantee that the relative probability of ending up in two different minima depended *only* on their relative depths. We could then control the probabilities of getting particular outputs by manipulating the energy landscape (i.e., by changing the weights).

Once again, a physical analogy is helpful: If we have a ball bearing on a bumpy surface and we shake the whole system up and down, the ball bearing will be able to jump over the

(continued)

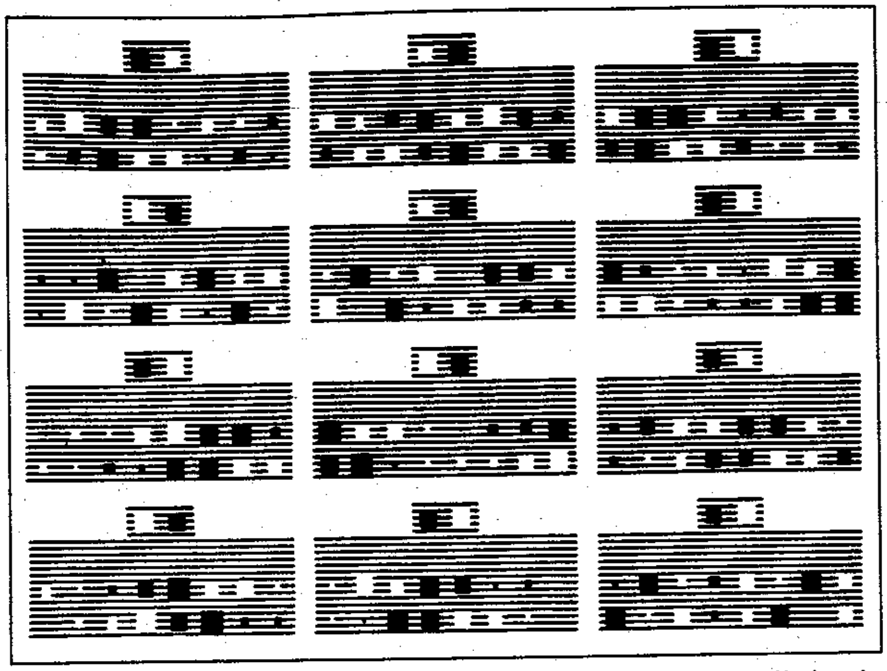

Figure 1b: *The weights that are learned by the 12 intermediate units. The black and white rectangles in the bottom two rows of a unit represent the weights on its connections to the input terminals. The sizes of the rectangles indicate the magnitudes of the weights. Black indicates a negative weight. The two weights at the top of each unit show how it affects the two output terminals. The weights all start at 0 and change by very small steps. Notice that all 12 units detect different combinations of active input terminals and that these combinations are generally sensible predictors of the global shift for which the unit "votes."*

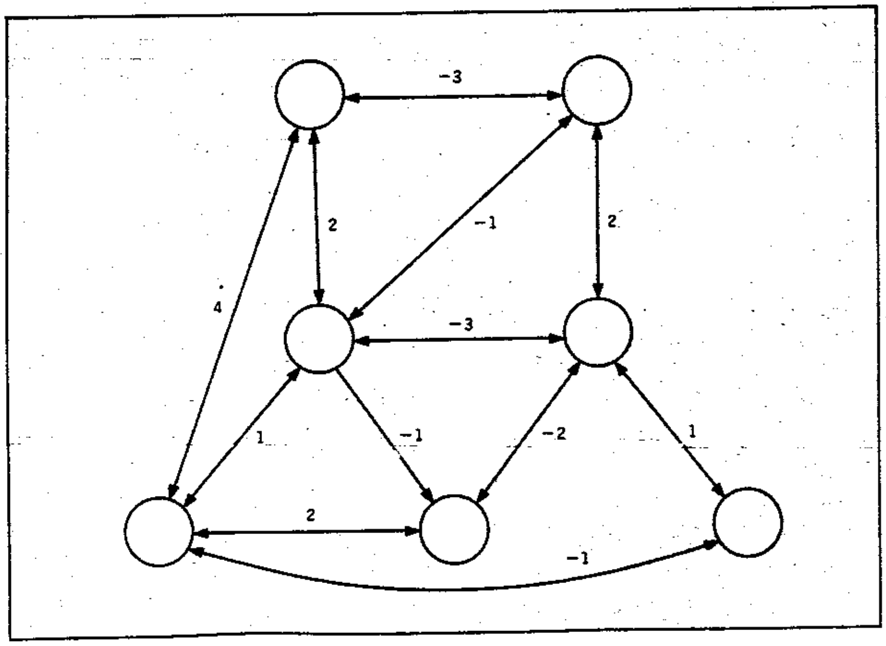

Figure 2: A *simple network with three input units at the bottom, two intermediate units in the middle, and two output units at the top. All the connections are symmetrical.*

barriers that separate shallow minima from deep ones: the ball bearing will spend most of its time in the deeper minima, even though it will occasionally sample higher energy states. If we shake for a while in just the right way, a useful simplification occurs: We approach a condition called "thermal equilibrium" in which the ball bearing is still moving from place to place, but the *probability* of finding it at any one place on the surface is stable and depends only on the height of the surface at that point—it doesn't depend on where the ball bearing started or on the shape of the energy landscape. More precisely, the log of the probability ratio of finding the ball bearing in two different states is proportional to the energy difference of those two states. Scott Kirkpatrick at IBM introduced the idea of using "thermal noise" to escape from local minima and to increase the chances of finding the deeper minima (see reference 2). He has shown that for large problems in which the cost of a solution is the analog of energy, an effective method for finding low-cost solutions is to start with a lot of thermal noise and gradually reduce it—a process that he calls "simulated annealing."

In our parallel networks it is easy to introduce the analog of thermal noise. We just modify the decision rule that is used by the individual units. They still compute the sum of the weights on the connections coming from other active units, but instead of always turning on when this sum is positive and off when it is negative (which always reduces the energy of the network), they behave probabilistically, as shown in figure 3.

Using this probabilistic decision rule, we can run networks in the following way: Clamp an input vector, let the remaining units turn on and off probabilistically until the network has reached thermal equilibrium, and then read the output vector. At equilibrium the output units will continue to change states, but each output vector will have a fixed probability that does not vary with time.

Research teams in fields as diverse as statistics (Stuart and Donald Geman), neuroscience (Terry Sejnowski), psychology (Paul Smolensky), and artificial intelligence are now investigating networks of this kind. Statisticians call them Markov Random Fields. At Carnegie-Mellon University we call our particular version the Boltzmann Machine, in honor of Ludwig Boltzmann, one of the founders of statistical mechanics.

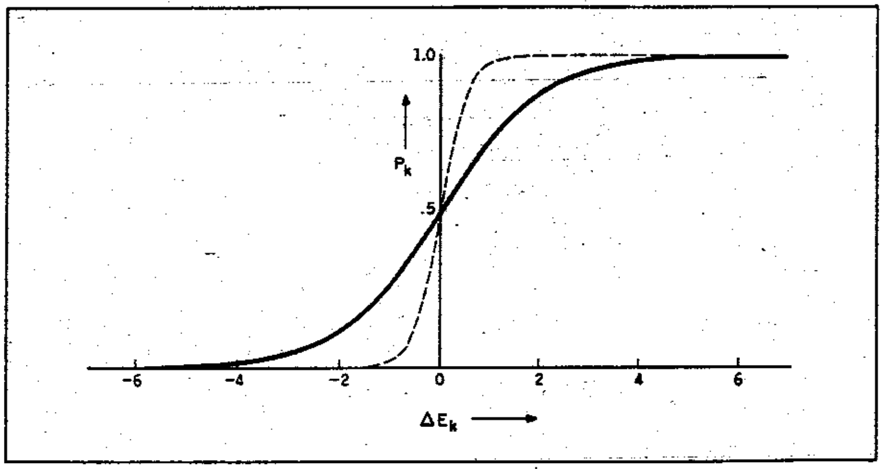

Figure 3: *This shows the probability p_k with which the kth unit is active. The quantity ΔE_k is the sum of the weights on connections between the kth unit and other currently active units. The equation is*

$$p_k = \frac{1}{1+e^{-\Delta E_k/T}}$$

T is the level of thermal noise in the network. The solid curve is for T = 1 and the dotted curve is for T = 0.25. If the value of T is decreased, the unit becomes less probabilistic. When T = 0, the curve becomes a deterministic step function.

Learning Again

We can now return to the issue of learning. First, we redefine the learning task in probabilistic terms. For each possible input vector, we want to produce each possible output vector with a certain probability. (Generally, most of these probabilities will be close to 0 and a few will be close to 1.) We can then train the network to behave in this way by alternating between two phases that are very similar to those used in the earlier learning rule.

In phase 1 we tell the network about the desired probabilities by clamping pairs of input vectors and output vectors with the corresponding frequencies. Each time a pair of input and output vectors is clamped, we run the network until it is close to thermal equilibrium; we then run the network for a little extra time, modifying the weights in the following way: For each unit of time during which two units are both active, we increment the weight between them by δ.

In phase 2 we clamp input vectors and let the network decide for itself what output vector to give. Once it has approached equilibrium, we run a little longer, as before, and now *decrement* by δ the weights between pairs of active units. If we keep alternating between phase 1 and phase 2, showing the network all the various pairs of input and output vectors, the net change in the weight between any two units will be proportional to the difference between the probability that the two units are both active in phase 1 and the probability that they are both active in phase 2 (averaged over all I/O pairs). It is remarkable that when these probabilities are measured at thermal equilibrium, their dif-

(continued)

ference is exactly the right quantity to use for changing the weights to make the behavior of the network in phase 2 (when it is deciding for itself) mimic the behavior in phase 1 (when it is being forced to behave in the desired way). To prove this it is necessary to define a measure of the difference between the probability distribution that is forced on the network in phase 1 and the probability distribution that it exhibits in phase 2. Once the correct measure has been defined, it can be shown that the measure is decreased by changing each weight according to the above procedure. The proof can be found in reference 3.

Figure 1 shows what the learning procedure can do when the task is to "recognize" the shift that was applied to one 8-bit vector to produce a second 8-bit vector. If you think this is an easy problem, remember that the network starts off with no preconceptions. It has no idea that neighboring input bits will have anything to do with each other, and it is not expecting this task any more than it is expecting any other. If the very same network is presented with a completely different combination of input and output vectors, it will create a different set of feature detectors that are appropriate for the different task.

MAKING IT FASTER

The first learning algorithm I described just changes weights to make units behave in prespecified ways. It cannot figure out what to do with internal units whose required behavior is not specified from outside. The second learning algorithm is potentially much more powerful because it is able to decide how to use the internal units to help achieve the required I/O mapping. It actually constructs simple internal representations. Unfortunately, there is a heavy price to pay for this added power. The algorithm is currently extremely slow; the example in figure 1 requires hours of computer time.

To speed things up, Blake Ward, a graduate student at Carnegie-Mellon, has built a parallel machine containing six Omnibyte 68000 boards, each of which has a copy of the entire network. Each board runs with a different input vector, and then all the boards agree on how to change the weights. This helps, but ultimately we would like to implement networks of these probabilistic units directly in silicon. Unlike current computers, these networks are rather tolerant of localized hardware failures or fabrication errors; Carver Mead has pointed out that an analog implementation of the processing elements would positively thrive on the kind of thermal noise that comes from running transistors at very low power. This might make it possible to build much larger chips than is currently feasible. However, developments like this are still a long way off, and they do not remove the need for more theoretical progress. Our current simulations are slow for three reasons: It is inefficient to simulate parallel networks with serial machines, it takes many decisions by each unit before a big network approaches equilibrium, and it takes an inordinate number of examples of I/O pairs before a network can figure out what to represent with its internal units. Better hardware might solve the first problem, but more theoretical progress is needed on the other two. Only then will we be able to apply this kind of learning network to more realistic problems. ■

REFERENCES

1. Hopfield, John J. "Neural Networks and Physical Systems with Emergent Collection Computational Abilities," *Proceedings of the National Academy of Sciences*, 1982, vol. 79, pages 2554–2558.
2. Kirkpatrick, S., C. D. Gellatt, and M. D. Vecchi, "Optimization by Simulated Annealing," *Science*, 1983, vol. 220, pages 671–680.
3. Ackley, D. H., G. E. Hinton, and T. J. Sejnowski. "A Learning Algorithm for Boltzmann Machines," *Cognitive Science*, 1985, vol. 9, pages 147–169.

ACKNOWLEDGMENTS

Terry Sejnowski and I performed the research described here while we were supported by grants from the System Development Foundation. I thank the members of the Boltzmann Group at Carnegie-Mellon for helpful comments.