

The Neural Basis of Learning and Memory

When it comes to understanding mental function, intuition tends to be something of a hindrance. Intuition comes from what we are used to seeing, hearing, doing, thinking; it is the product of our interactions with the world. Despite this, many—including neurobiologists—are skeptical of any observation about the brain that seems unintuitive, as if our inescapable biases about the nature of things were the gold standard for interpreting neural function. But there is no good reason to expect the brain to work in ways we are familiar or even comfortable with. We will see below that even small neural networks are capable of rather exotic behaviors, so one can only imagine the kinds of complex behavior the brain's massive networks are capable of, and how strange they might seem in relation to our world of the few, the slow and the simple. The goal of this document is to give students an idea of how the power of massively parallel processing (neural networks) allows us to learn and remember. As the reader may have guessed, the explanation is not entirely intuitive. But we will take our time and focus on only the most rudimentary concepts.

Neurons and Synapses

The image in Figure 1 gives you some idea of what a neural network looks like, although this is just a tiny fraction of one. The bright spots are the bodies of neurons, while the thin prolongations are either axons, which send signals to other neurons, or dendrites, which receive inputs from axons. The fundamental property of a neural network is that it has many "units" connected together. I put "units" in quotes to emphasize that the behavior of

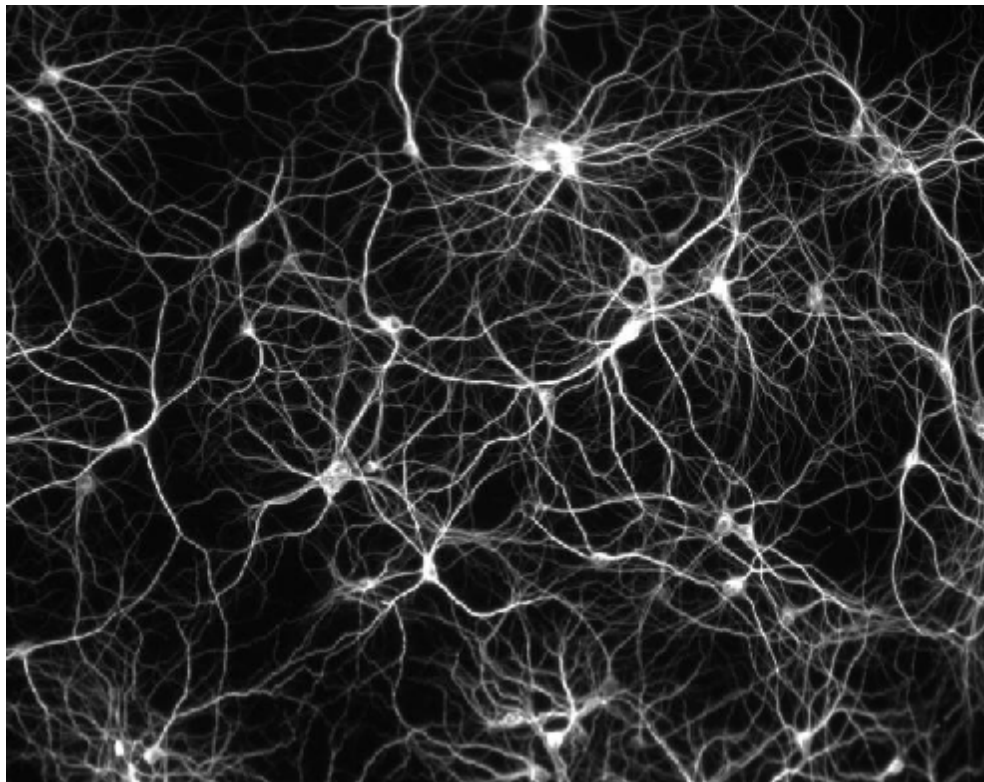


Figure 1

the neurons is not really critical. The computational abilities of a network come from the interconnections, and the properties of a network—how it behaves—are said to be *emergent*. This means that network behavior is far more complex and interesting than anything you might predict just from the sum of its parts. It is the connectedness and numerosity of networks that make them special.

Figure 2 depicts a synapse, which is the interface between an axon and a dendrite. Synapses are to neural function what makeup is to Pam Anderson—completely indispensable. They do not simply connect neurons together; they connect them with a certain amount of efficiency. The basic message neurons send to each other is the action potential, an electrical pulse that zips down the axon and causes the synapse to conduct. If you imagine sending action potentials down an axon, then watching the receiving neuron release



Figure 2

action potentials, you would see that this happens a fraction of the time. If the efficiency were absolute, then each action potential from the sending neuron would cause one action potential from the receiving neuron. But in reality this only occurs a small fraction of the time, and this fraction is the efficiency of the synapse. This efficiency can change for a given synapse, and indeed this is the basis of learning and memory. We will see how this works in a moment. For now suffice it to say that networks are *plastic*, which means the many synapses within them can change in their efficiency, and this is what occurs in learning and memory.

Plasticity also explains the way stroke patients recover much or all of their brain function. A stroke happens when a small artery either infarcts (ischemic stroke) or ruptures (hemorrhagic stroke). In the first case the watershed area of the artery dies from lack of oxygen; in the latter the intracranial pressure increases and circulation is impaired near the rupture. Depending on the affected tissue, strokes may cause paralysis, aphasia, visual deficits, etc, but commonly these deficits wane and sometimes disappear. People used to think recovery involved the growth of new neurons, or that axons were spreading out and forming new connections. Instead, what happens is the other connections—the ones that survived—adjust their efficiency in a way that tends to compensate for the damaged tissue. In a sense, the networks retune themselves, much as a post office might redistribute responsibilities after several employees are killed by a psycho.

The Hopfield Net

John Hopfield was at Caltech until recently. He did a lot of things but he is best known for a simulation study in which he demonstrated surprisingly sophisticated behavior from a tiny neural network. Real brain networks are quite different, but some basic principles apply to both. Understanding these principles will help you perceive neural function in a

way that's closer to reality than anything your intuition might have afforded you up to now.

Characteristics of Hopfield's Net

Hopfield was something of a visionary, having a sense for how theoretical principles might be manifest in the brain. In 1982 he published a seminal paper called *Neural networks and physical systems with emergent collective computational abilities*. I have uploaded it to the Chalk site if you're interested (but it's not required). In this study he put together a very small network—about 100 units—using very simple units and connection rules. The units were simple because they could only have one of two states, on or off, which he denoted 1 or 0. The connection rule was as follows: Every unit was connected to every other unit, and each unit decided, at a given moment, whether to be on or off based on the sum of its weighted inputs. Every connection between two units has a certain *weight*, which is precisely the same as efficiency; it is the connection strength. So, when I say that each unit looks at the sum of its weighted inputs, I mean that each input is the state of the sending neuron (0 or 1) times the weight of the connection. Let's take an example with 3 neurons.

In Figure 3, the numbers shown in red are the connection strengths. The number inside each circle is the activation state (1 or 0; on or off). Looking at the top neuron, its total input is $(0.2 \times 0) + (0.8 \times 1) = 0.8$. The bottom neuron's input is $(0.2 \times 1) + (-0.2 \times 1) = 0$. The connection rule stated that if the sum of weighted inputs was greater than zero, the neuron should become active. Otherwise it would be inactive.

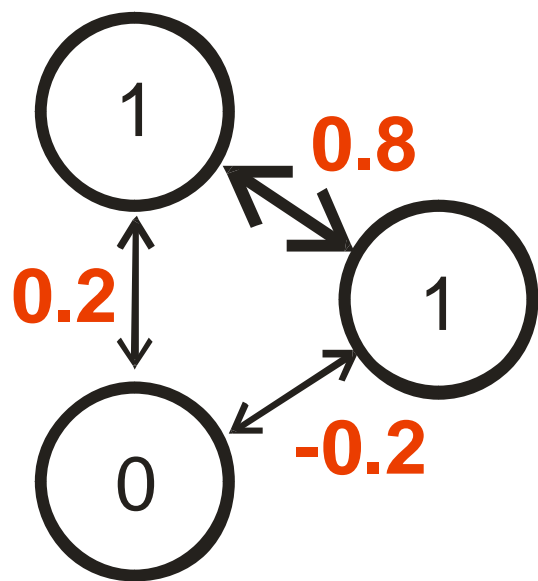


Figure 3

As you might suspect, a network could easily get stuck. For example, if the top and right units in the diagram don't change, then the bottom unit won't change either. But the top won't change because the bottom and right aren't changing, and the right won't change because the top and bottom aren't changing. So Hopfield made his network *asynchronous*. This means that each unit updates its status at random times, which in turn means that the collective units do not update simultaneously. When it works like this, the network doesn't get stuck, but as we will see, it does tend to fall into a stable state.

Before the reader starts wondering what this could possibly have to do with the brain, I point out that you're about to see how this incredibly simple network is able to store memories, and even think, in a simple way. Please read on.

Training the Net

Hopfield used a special technique called the *storage algorithm* to set all the connection weights. It doesn't matter how this works, only that it is a prescription for constructing a

certain number of *stable states*. Let us imagine that we set all the connection weights randomly, then start running the network (start the asynchronous updates). If the connection weights have been programmed with the storage algorithm, the network doesn't just flop around forever; it settles into one of the stable states. In fact, it will settle into the one closest to the starting point. To understand stable states, we must understand what a state is.

A state is simply the set of all unit activities (1 or 0) at a given moment. For example, in the 3-unit network above, the state is (0, 1, 1). We can actually draw that state on a graph by showing a point whose coordinates are (0, 1, 1). So, even though there are 3 units in this network, its state can be expressed with a single point in 3-dimensional space (Figure 4). In fact, no matter how many units there are, a network's state can be seen as a single point in a many-dimensional space. If there are n units, the space is n -dimensional. Of course we can't

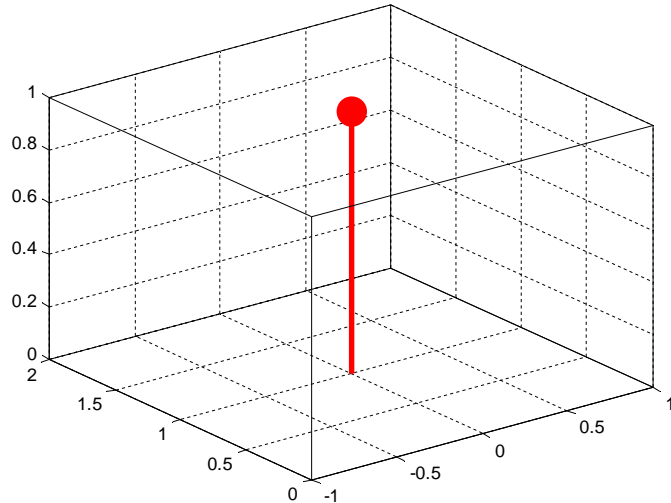


Figure 4

visualize anything greater than a 3-dimensional space. This is actually a good example of the shortcomings of intuition. A 4- or 5- or 100-dimensional space is just as valid as a 2- or 3-dimensional space, but our physical experience has never seen anything greater than 3-dimensional. So we can't visualize 4 dimensional. If you think you can, try intuiting the fact that a 3-dimensional object is the shadow of a 4-dimensional object.

How's that working out? Anyway, a network can be programmed to have, say, 10 stable states¹. Each stable state can be thought of as a memory; it is a specific set of unit activities, and it tends to occur more often than random states because of the *pattern of connection weights*. When we set a network in motion, the various units update their status periodically, and each time one of them does this, the state of the network changes. As the state of the network changes from moment to moment, we can think of this as a point wandering around in n -dimensional space, where n is the number of units. Each of the stable states corresponds to a certain location in the n -space, and when our wandering point (the current network state) comes close to one of these stable state locations, it is drawn towards it, and it tends to stay there. When this happens, the network has just retrieved a memory. Figure 5 shows a random state (left) and two stable states. This

¹ In fact, for a Hopfield-type network, 13 stable states can be stored for every 100 units. "Local" networks in the cortex—heavily connected groups of neurons within ~1 mm of each other, have on the order of 100,000 units.

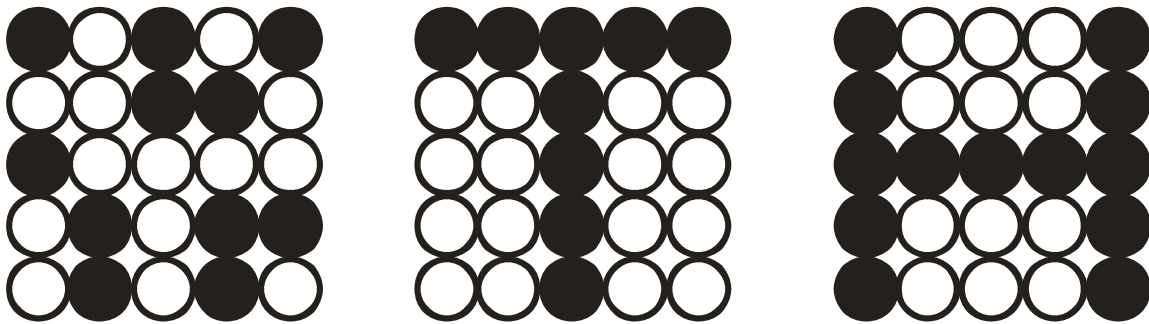


Figure 5

illustrates how a simple network could be used to store memories of the alphabet, the idea being that each letter corresponds to a single stable state.

To "stimulate" a network is to set it to a certain state. Since the network is refreshing asynchronously, it tends to quickly move away from that state. But if we start it in a state corresponding to one of its stable states, it will not move away from that state. The stable states are *attractors*; they attract the network towards themselves. This isn't much use if we have to set the network to a particular stable state in order to get it to go into that state. But networks *generalize*. For example, if we set one or two of the dots in the "T" to white instead of black, that changes the state of the network. But after jiggling around for awhile, the network will end up looking like a T again because it has been attracted to that stable state. However, this will not happen if the network's starting state is closer to one of the other stable states, for example the "H."

Let us think again about the point in n-dimensional space which represents a network's current state. Now think of a number of dark regions in this space, like so many black holes scattered about in the space. These black holes attract the network, and the network tends to fall into the black hole it is closest to. This falling is the network equivalent of remembering something, because given a certain starting point, the network has chosen a stable state—it has chosen a memory; it has remembered.

Noise, Rambling and Thinking

Real neurons are noisy, which means their activity tends to fluctuate randomly from moment to moment. Think of it as a flickering light bulb. So, if a network is in a given stable state, it may come out of that state after a while because a number of units have randomly changed their status. Thinking again of a point in n-space representing the network state, we imagine it to be jittering somewhat, and sometimes it jitters so much that it ends up closer to one of the other stable states than the one it was in. When this happens, the network will be drawn into this new stable state. Hopfield found that when a certain amount of noise was added to the units of his network, the network will flop from stable state to stable state, spending some time in each one of them, and occasionally hovering around one of them. This progression of stable states may be a reasonable model for the way we think.

Natural Training

Real neural networks don't know the storage algorithm, so how do they learn? That is, how do they acquire memories? It's surprisingly simple, and it's called Hebb's rule. Remember that every unit is connected to every other, in each case with a certain strength, or connection weight. Now let us say the network is stimulated repeatedly with a T. Recall that for this kind of simple network, stimulating it with something means setting it to a state corresponding to that something. Hebb's rule says that whenever two units are active simultaneously, the connection strength between them increases just a little bit. So, if we stimulate with a T, the units you see in black in Figure 5 are all active simultaneously, so all the connection strengths between these units tend to increase. This means that the units in black—which form the T—tend to reinforce each other. For instance, if the bottom unit of the T is missing, it will likely become active because all the other units of the T have a strong connection to it and they are trying to activate it. So, if we show the T over and over, or for a long time, it tends to be *burned into the network*. This is how stable states are formed; they are just recreations of stimuli the network has experienced often.

This, of course, resembles the way we learn. I emphasize that, at least according to this model, learning is not a process where memories are stored in some separate place, later to be retrieved from that place. Instead, learning probably involves the burning of stable states in the various networks involved in any particular sensory, motor or cognitive experience. So, for example, the memory of an object we see would take the form of a stable state in the visual cortex. Remembering this object would involve somehow coaxing the networks towards that stable state until they fall into it. So memories are not objects stored separately, but *tendencies* of vast networks to fall into certain states.

Emergent Properties of the Hopfield Net

We have seen that a very simple network can have surprisingly interesting properties, including memory storage and retrieval, generalization, and even a behavior similar to thinking. These properties are strictly emergent because the units themselves have no memory and certainly no capacity to generalize and think; it is only the formation of a network that gives rise to these capabilities. Even a 100-unit network displays these emergent properties, so one can imagine the computational abilities of the brain's neural networks, which may contain hundreds of thousands of neurons. Hopefully the student will realize that complex mental functions like memory and cognition are well within the reach of a sizeable network.

Feedforward Nets

The Hopfield net is a single layer of units, all connected together. We have seen that this kind of net has remarkable properties that resemble biological learning, memory and thought. However, Hopfield nets cannot really *compute*, which is to say they cannot transform a particular input to a particular output. All we can do with a Hopfield net is set all the units to some starting activity, then watch the network ramble around a bit until it falls into the nearest stable state. In computer science this is called *content addressable memory*. This is to be contrasted with *random access memory*, where the user supplies an address and the computer fetches the memory stored at that address. With content addressable memory, the user supplies an actual memory, or part of an actual memory,

and the system looks through all its memory banks to see if that chunk is stored somewhere. This is what happens when you use the "find" function in a word processor, or when you type something in the search field of a search engine like Google: the system returns a memory, in the form of a word or a web page, that contains the chunk you typed in. In a Hopfield net, you can set the units of the network to some starting point representing a complete or partial memory, and the network will "return" whichever memory it has stored resembling the input. By return we mean the network will fall into that state.

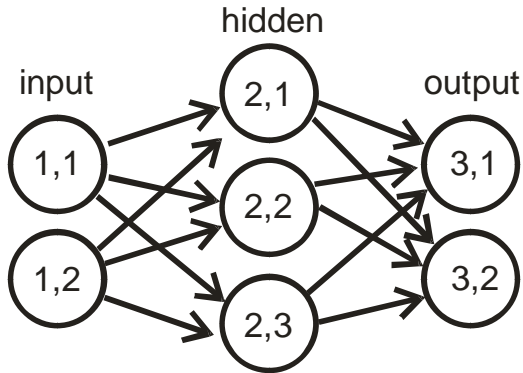


Figure 6

Now consider the feedforward network shown in Figure 6. Instead of just one layer, it has 3 layers, although similar principles apply to a 2-layer network. The 3-layer network is used very commonly. Let's imagine that this network is being used to decipher bar codes on products in a grocery store. The network's task is to take a 2-dimensional image of a bar code and from it produce a single number which is a unique tag for that product. The network would in fact need many more units in all the layers in order to do this, but the simple net shown in Figure 6 captures the basic concepts.

Like in the Hopfield net, all connections are characterized by a specific weight. In Figure 7, we consider a single unit from the output layer of the net shown in Figure 6.

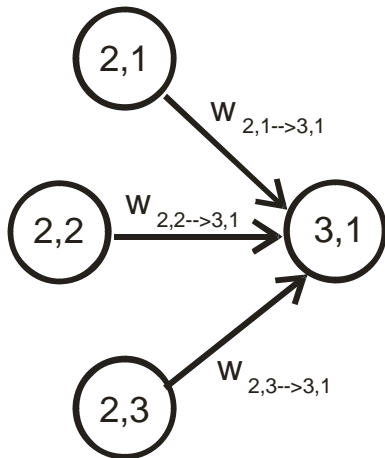


Figure 7

Like all units from layers 2 and 3, this unit's input is the sum of the weighted inputs from the previous layer; specifically

$$input = A_{2,1} \times w_{2,1 \rightarrow 3,1} + A_{2,2} \times w_{2,2 \rightarrow 3,1} + A_{2,3} \times w_{2,3 \rightarrow 3,1} \quad (1.1)$$

where A is the activity of the unit in question. This activity can be 0 or 1 if you want to constrain it like that, but usually activity is allowed to have any value between 0 and 1.

Some of you may have heard the term "linear combination." In fact that is precisely what Eq. (1.1) is. A linear combination is just the sum of several numbers, each one multiplied by some constant. It is also called a weighted sum. There is a fundamental property of linear combinations, which is that if you do several of them, the order doesn't matter. Not only that, but several linear combinations in sequence can always be represented by a single linear combination. Why does this concern us? Because it does no good to have a multilayer network if the transformations at each stage are linear. In that case we may as well have a single layer network. So to take advantage of

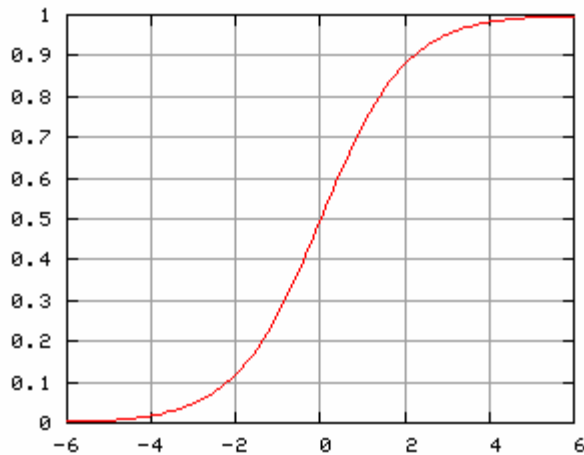


Figure 8

the multi-stage nature of a feedforward network, we need to make the transformation between stages nonlinear. Any sort of nonlinearity will suffice; the most common is to take the weighted sum and make this quantity the input to a sigmoid function. For example, in Figure 8 the x axis represents the linear combination of inputs (weighted sum), and the y axis shows the sigmoid transformation of this sum. So, to be accurate, we must say that each unit in layers 2 and 3 takes a weighted sum of its inputs, then corrects this input with a sigmoid

transformation. In the Hopfield net described above, the nonlinear transformation is not sigmoidal but rectangular—it steps up abruptly as the x value goes from negative to positive.

So far all this talk about feedforward networks has seemingly little to do with learning and memory. But to understand that link we must know how a feedforward network is "trained." And to understand that, we must emphasize again that computation is always a *transformation* of some input to some output. Let us assume, as before, that our feedforward network is being used to recognize bar codes. This means it is making a transformation from a 2-dimensional image (the input) to a single number (the output). This number could be easily represented in the output layer by making the activity of one unit represent the number of thousands, where another unit represents the hundreds, another represents the tens, and another represents the ones. I hasten to point out that our conventional way of numbering things is based on a linear combination, but I'll leave it to the student to figure out why that is so.

To make a network useful, it has to be trained. This is done with a *training set*—a set of input-output pairs. For example, we might take one bar code image and pair it with the number it is known to represent. Another pair would be a different bar code image and another number. Let's assume we have ten such pairs, which collectively give a good idea of the relationship between bar code images and single numbers. If a human were very smart, she could look at these pairs and figure out how to convert a bar code image to a number. In that case she would be learning by example. She would be learning how to transform a bar code to a number.

Networks learn this too. We must understand that any computation (transformation) can be realized if we know how to set all the weights (connection strengths) between neurons in the different layers. It took a long time for people to figure out how to do this, but they finally came up with a method called *backpropagation*. The idea is this: you set the input layer in some specific way; for example you could make the input layer actually bitmap the image of a certain bar code. The network's connectivity will naturally transform this input to something else, represented in the output layer. The networks

usually begin by setting weights randomly, so the output will be some random transformation of the input. Then, we compare the output with the *desired output*. In our case, the desired output would be the representation of a number. Now, based on the difference between the actual output and the desired output, we go through the network and tweak the various weights, just a little bit, in such a way as to reduce the difference between actual and desired outputs. We then repeat this process for all the pairs in the training set, ultimately arriving at a set of connection weights that accurately creates an appropriate number for each bar code image in the training set.

What happens next is one of the most remarkable things in the history of computer science. It is called *generalization*. If we have trained our network by adjusting weights for each pair of the training set, we can then show it new bar codes that it has never seen before, and (assuming sufficient training) it will know how to compute a unique number for each of them. That is, the network can take what it has learned by example and generalize this in such a way that previously unknown inputs can be correctly transformed. In fact, the ability to do this is a good way of defining *intelligence*: the ability to learn by example, then generalize to new cases.

Learning in Biological Networks

When discussing Hopfield networks we learned that Hebb's rule provides a plausible mechanism for learning in neural networks. But is biological learning really like this?

The answer is, probably. This is because of a well-established phenomenon called *long-term potentiation*, or LTP. An experimenter can isolate a pair of neurons connected by a synapse, then use electrical current to stimulate the first neuron. This stimulation causes the synapse to conduct frequently and for a long time, conveying as it does each action potential coming from the first neuron. After driving the synapse like this, one can measure the efficiency of the synapse—its ability to convert incoming action potentials to action potentials in the target neuron. This efficiency, it turns out, is increased. Not only that, the increased efficiency lasts a long time—on the order of hours or even days, rather than seconds or minutes.

So what? Well, we need to think of synaptic efficiency as a connection weight. In a Hopfield net, it is easy to understand how the Hebbian principle would burn certain memories into a network. Things are more complicated with a feedforward network, but nevertheless not hard to understand. In the learning process, as in backpropagation, biological networks would adjust themselves by sending feedback projections from terminal layers to earlier layers. Certain input connections would then be strengthened (or lessened by a mechanism called long-term depression, or LTD) in such a way that the overall transformation effected by the networks achieves its goal. So, generally speaking, a *memory* in a biological network amounts to a set of connection strengths, namely the connection strengths between all the layers of the network. Computer scientists refer to these connection strengths as *data*, because they are the product of previously acquired information. But, unlike the Hopfield net which principally stores memories, feedforward nets store the capacity to transform. And, as we have seen, they probably implement this capacity in the form of LTP.

Conclusion

If we hypothesize that biological networks adjust their connection weights through LTP, then we must predict that memories (including the ability to make certain transformations) will fade with time. This is because LTP is not infinite, so unless the network is put to use occasionally, such that the stronger connections are reinforced, it will forget. This, of course, resembles biological memory.

It is a common question, some would say philosophical, to ask whether computers can think. Based on the above, the appropriate answer would seem to be *of course, what else are they doing?* However, some maintain that there is an ethereal quality to human thought, mysterious and magical, and even go so far as to claim that animals cannot think. By now the student should appreciate that the fundamental aspects of thought—generalization, learning by example, memory storage and recall—can all be simulated with a neural network (which can be simulated with a modern computer). The only difference is that the human capacity to do these things far surpasses the capacity of a neural network, but this is a quantitative difference, not a qualitative one.