Chapter 6

How Neural Networks Learn from Experience

Geoffrey E. Hinton

The brain is a remarkable computer. It interprets imprecise information from the senses at an incredibly rapid rate. It discerns a whisper in a noisy room, a face in a dimly lit alley and a hidden agenda in a political statement. Most impressive of all, the brain learns—without any explicit instructions—to create the internal representations that make these skills possible.

Much is still unknown about how the brain trains itself to process information, so theories abound. To test these hypotheses, my colleagues and I have attempted to mimic the brain's learning processes by creating networks of artificial neurons. We construct these neural networks by first trying to deduce the essential features of neurons and their interconnections. We then typically program a computer to simulate these features.

Because our knowledge of neurons is incomplete and our computing power is limited, our models are necessarily gross idealizations of real networks of neurons. Naturally, we enthusiastically debate what features are most essential in simulating neurons. By testing these features in artificial neural networks, we have been successful at ruling out all kinds of theories about how the brain processes information. The models are also beginning to reveal how the brain may accomplish its remarkable feats of learning.

In the human brain, a typical neuron collects signals from others through a host of fine structures called dendrites. The neuron sends out spikes of electrical activity through a long, thin strand known as an axon, which splits into thousands of branches. At the end of each branch, a structure called a synapse converts the activity from the axon into electrical effects that inhibit or excite activity in the connected neurons. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.

Artificial neural networks are typically composed of interconnected "units," which serve as model neurons. The function of the synapse is modeled by a modifiable weight, which is associated with each connection. Most artificial networks do not reflect the detailed geometry of the dendrites and axons, and they express the electrical output of a neuron as a single number that represents the rate of firing—its activity.

Each unit converts the pattern of incoming activities that it receives into a single outgoing activity that it broadcasts to other units. It performs this conversion in two stages. First, it multiplies each incoming activity by the weight on the connection and adds together all these weighted inputs to get a quantity called the total input. Second, a unit uses an input-output function that transforms the total input into the outgoing activity.

The behavior of an artificial neural network depends on both the weights and the input-output function that is specified for the units. This function typically falls into one of three categories: linear, threshold or sigmoid. For linear units, the output activity is proportional to the total weighted input. For threshold units, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value. For sigmoid units, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurons than do linear or threshold units, but all three must be considered rough approximations.

To make a neural network that performs some specific task, we must choose how the units are connected to one another, and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

The most common type of artificial neural network consists of three groups, or layers, of units: a layer of input units is connected to a layer of "hidden"

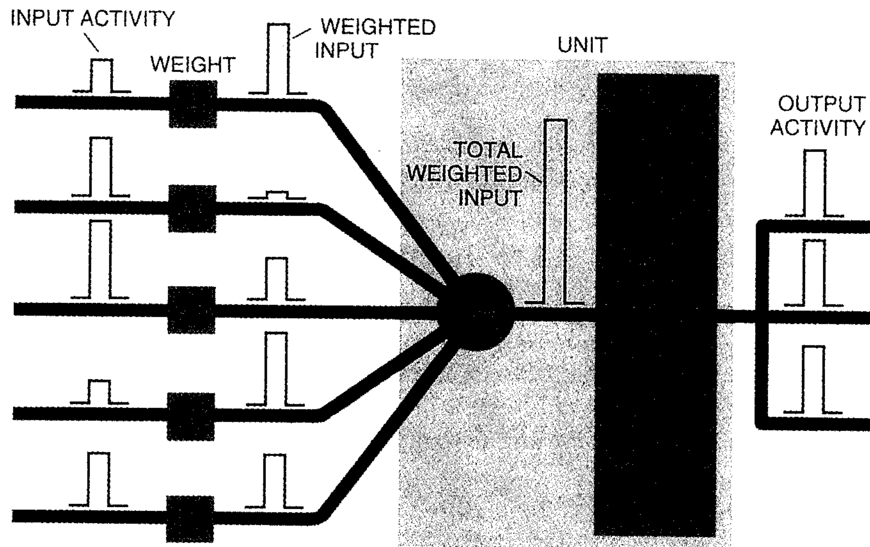


Figure 6.1
Idealization of a neuron processes activities, or signals. Each input activity is multiplied by a number called the weight. The "unit" adds together the weighted inputs. It then computes the output activity using an input-output function.

units, which is connected to a layer of output units. The activity of the input units represents the raw information that is fed into the network. The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and hidden units. Similarly, the behavior of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We can teach a three-layer network to perform a particular task by using the following procedure. First, we present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units. We then determine how closely the actual output of the network matches the desired output. Next we change the weight of each connection so that the network produces a better approximation of the desired output.

For example, suppose we want a network to recognize handwritten digits. We might use an array of, say, 256 sensors, each recording the presence or absence of ink in a small area of a single digit. The network would therefore need 256 input units (one for each sensor), 10 output units (one for each kind of digit) and a number of hidden units. For each kind of digit recorded by the sensors, the network should produce high activity in the appropriate output unit and low activity in the other output units.

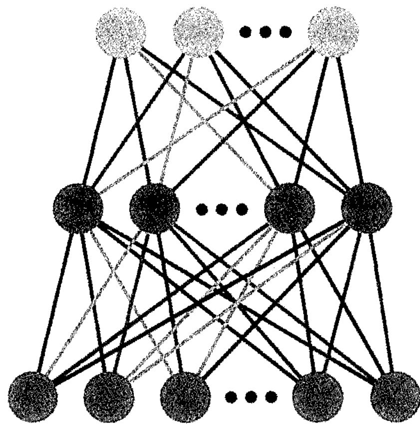


Figure 6.2
Common neural network consists of three layers of units that are fully connected. Activity passes from the input units to the hidden units and finally to the output units.

To train the network, we present an image of a digit and compare the actual activity of the 10 output units with the desired activity. We then calculate the error, which is defined as the square of the difference between the actual and the desired activities. Next we change the weight of each connection so as to reduce the error. We repeat this training process for many different images of each kind of digit until the network classifies every image correctly.

To implement this procedure, we need to change each weight by an amount that is proportional to the rate at which the error changes as the weight is changed. This quantity—called the error derivative for the weight, or simply the EW—is tricky to compute efficiently. One way to calculate the EW is to perturb a weight slightly and observe how the error changes. But that method is inefficient because it requires a separate perturbation for each of the many weights.

Around 1974, Paul J. Werbos invented a much more efficient procedure for calculating the EW while he was working toward a doctorate at Harvard University. The procedure, now known as the back-propagation algorithm, has become one of the more important tools for training neural networks.

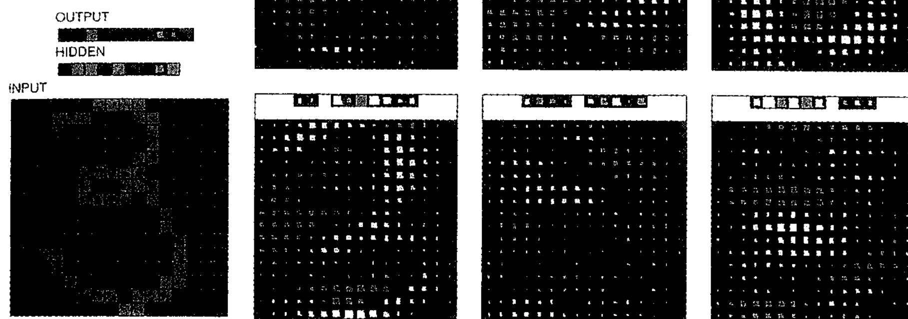
The back-propagation algorithm is easiest to understand if all the units in the network are linear. The algorithm computes each EW by first computing the EA, the rate at which the error changes as the activity level of a unit is changed. For output units, the EA is simply the difference between the actual and the desired output. To compute the EA for a hidden unit in the layer just before the output layer, we first identify all the weights between that hidden unit and the output units to which it is connected. We then multiply those weights by the EAs of those output units and add the products. This sum equals the EA for the chosen hidden unit. After calculating all the EAs in the hidden layer just before the output layer, we can compute in like fashion the EAs for other layers, moving from layer to layer in a direction opposite to the way activities propagate through the network. This is what gives back propagation its name. Once the EA has been computed for a unit, it is straightforward to compute the EW for each incoming connection of the unit. The EW is the product of the EA and the activity through the incoming connection.

For nonlinear units, the back-propagation algorithm includes an extra step. Before back-propagating, the EA must be converted into the EI, the rate at which the error changes as the total input received by a unit is changed. (The details of this calculation are given in box 6.2.)

The back-propagation algorithm was largely ignored for years after its invention, probably because its usefulness was not fully appreciated. In the early 1980s, David E. Rumelhart, then at the University of California at San Diego, and David B. Parker, then at Stanford University, independently rediscovered the algorithm. In 1986, Rumelhart, Ronald J. Williams, also at the University of California at San Diego, and I popularized the algorithm by demonstrating that

Box 6.1
How a Neural Network Represents Handwritten Digits

A neural network–consisting of 256 input units, nine hidden units and 10 output units–has been trained to recognize handwritten digits. The illustration below shows the activities of the units when the network is presented with a handwritten 3. The third output unit is most active. The nine panels at the right represent the 256 incoming weights and the 10 outgoing weights for each of the nine hidden units.



it could teach the hidden units to produce interesting representations of complex input patterns.

The back-propagation algorithm has proved surprisingly good at training networks with multiple layers to perform a wide variety of tasks. It is most useful in situations in which the relation between input and output is nonlinear and training data are abundant. By applying the algorithm, researchers have produced neural networks that recognize handwritten digits, predict currency exchange rates and maximize the yields of chemical processes. They have even used the algorithm to train networks that identify precancerous cells in Pap smears and that adjust the mirror of a telescope so as to cancel out atmospheric distortions.

Within the field of neuroscience, Richard Andersen of the Massachusetts Institute of Technology and David Zipser of the University of California at San Diego showed that the back-propagation algorithm is a useful tool for explain-

ing the function of some neurons in the brain's cortex. They trained a neural network to respond to visual stimuli using back propagation. They then found that the responses of the hidden units were remarkably similar to those of real neurons responsible for converting visual information from the retina into a form suitable for deeper visual areas of the brain.

Yet back propagation has had a rather mixed reception as a theory of how biological neurons learn. On the one hand, the back-propagation algorithm has made a valuable contribution at an abstract level. The algorithm is quite good at creating sensible representations in the hidden units. As a result, researchers gained confidence in learning procedures in which weights are gradually adjusted to reduce errors. Previously, many workers had assumed that such methods would be hopeless because they would inevitably lead to locally optimal but globally terrible solutions. For example, a digit-recognition network might consistently home in on a set of weights that makes the network confuse ones and sevens even though an ideal set of weights exists that would allow the network to discriminate between the digits. This fear supported a widespread belief that a learning procedure was interesting only if it were guaranteed to converge eventually on the globally optimal solution. Back propagation showed that for many tasks global convergence was not necessary to achieve good performance.

On the other hand, back propagation seems biologically implausible. The most obvious difficulty is that information must travel through the same connections in the reverse direction, from one layer to the previous layer. Clearly, this does not happen in real neurons. But this objection is actually rather superficial. The brain has many pathways from later layers back to earlier ones, and it could use these pathways in many ways to convey the information required for learning.

A more important problem is the speed of the back-propagation algorithm. Here the central issue is how the time required to learn increases as the network gets larger. The time taken to calculate the error derivatives for the weights on a given training example is proportional to the size of the network because the amount of computation is proportional to the number of weights. But bigger networks typically require more training examples, and they must update the weights more times. Hence, the learning time grows much faster than does the size of the network.

The most serious objection to back propagation as a model of real learning is that is requires a teacher to supply the desired output for each training example. In contrast, people learn most things without the help of a teacher. Nobody presents us with a detailed description of the internal representations of the world that we must learn to extract from our sensory input. We learn to understand sentences or visual scenes without any direct instructions.

How can a network learn appropriate internal representations if it starts with no knowledge and no teacher? If a network is presented with a large set of

Box 6.2
The Back-Propagation Algorithm

To train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and the actual output is reduced. This process requires that the neural network compute the error derivative of the weights (EW). In other words, it must calculate how the error changes as each weight is increased or decreased slightly. The back-propagation algorithm is the most widely used method for determining the EW.

To implement the back-propagation algorithm, we must first describe a neural network in mathematical terms. Assume that unit j is a typical unit in the output layer and unit i is a typical unit in the previous layer. A unit in the output layer determines its activity by following a two-step procedure. First, it computes the total weighted input x_j, using the formula

$$x_j = \sum_i y_i w_{ij},$$

where y_i is the activity level of the ith unit in the previous layer and w_{ij} is the weight of the connection between the ith and jth unit.

Next, the unit calculates the activity y_j using some function of the total weighted input. Typically, we use the sigmoid function:

$$y_j = \frac{1}{1 + e_j^{-x}}.$$

Once the activities of all the output units have been determined, the network computes the error E, which is defined by the expression

$$E = \frac{1}{2} \sum_j (y_j - d_j)^2,$$

where y_j is the activity level of the jth unit in the top layer and d_j is the desired output of the jth unit.

The back-propagation algorithm consists of four steps:

1. Compute how fast the error changes as the activity of an output unit is changed. This error derivative $\partial E/\partial y$ is the difference between the actual and the desired activity.

$$\frac{\partial E}{\partial y_j} = y_j - d_j$$

2. Compute how fast the error changes as the total input received by an output unit is changed. This quantity $\partial E/\partial x$ is the answer from step 1 multiplied by the rate at which the output of a unit changes as its total input is changed.

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \frac{dy_j}{dx_j} = y_j - d_j y_j (1 - y_j)$$

3. Compute how fast the error changes as a weight on the connection into an output unit is changed. This quantity $\partial E/\partial w$ is the answer from step 2 multiplied by the activity level of the unit from which the connection emanates.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial x_j} \frac{\partial x_j}{\partial w_{ij}} = \frac{\partial E}{\partial x_j} y_i$$

Box 6.2 (continued)

4. Compute how fast the error changes as the activity of a unit in the previous layer is changed. This crucial step allows back propagation to be applied to multilayer networks. When the activity of a unit in the previous layer changes, it affects the activities of all the output units to which it is connected. So to compute the overall effect on the error, we add together all these separate effects on output units. But each effect is simple to calculate. It is the answer in step 2 multiplied by the weight on the connection to that output unit.

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \frac{\partial x_j}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} w_{ij}$$

By using steps 2 and 4, we can convert the $\partial E/\partial y$ of one layer of units into $\partial E/\partial y$ for the previous layer. This procedure can be repeated to get the $\partial E/\partial y$ for as many previous layers as desired. Once we know the $\partial E/\partial y$ of a unit, we can use steps 2 and 3 to compute the $\partial E/\partial w$ on its incoming connections.

patterns but is given no information about what to do with them, it apparently does not have a well-defined problem to solve. Nevertheless, researchers have developed several general-purpose, unsupervised procedures that can adjust the weights in the network appropriately.

All these procedures share two characteristics: they appeal, implicitly or explicitly, to some notion of the quality of a representation, and they work by changing the weights to improve the quality of the representation extracted by the hidden units.

In general, a good representation is one that can be described very economically but nonetheless contains enough information to allow a close approximation of the raw input to be reconstructed. For example, consider an image consisting of several ellipses. Suppose a device translates the image into an array of a million tiny squares, each of which is either light or dark. The image could be represented simply by the positions of the dark squares. But other, more efficient representations are also possible. Ellipses differ in only five ways: orientation, vertical position, horizontal position, length and width. The image can therefore be described using only five parameters per ellipse.

Although describing an ellipse by five parameters requires more bits than describing a single dark square by two coordinates, we get an overall savings because far fewer parameters than coordinates are needed. Furthermore, we do not lose any information by describing the ellipses in terms of their parameters: given the parameters of the ellipse, we could reconstruct the original image if we so desired.

Almost all the unsupervised learning procedures can be viewed as methods of minimizing the sum of two terms, a code cost and a reconstruction cost. The

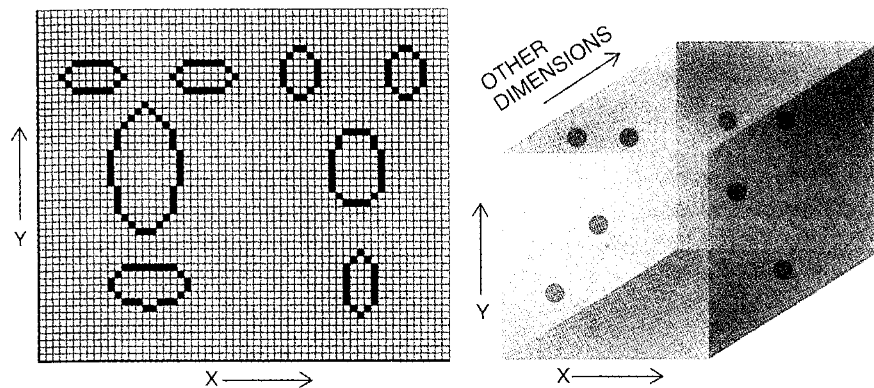


Figure 6.3
Two faces composed of eight ellipses can be represented as many points in two dimensions. Alternatively, because the ellipses differ in only five ways—orientation, vertical position, horizontal position, length and width—the two faces can be represented as eight points in a five-dimensional space.

code cost is the number of bits required to describe the activities of the hidden units. The reconstruction cost is the number of bits required to describe the misfit between the raw input and the best approximation to it that could be reconstructed from the activities of the hidden units. The reconstruction cost is proportional to the squared difference between the raw input and its reconstruction.

Two simple methods for discovering economical codes allow fairly accurate reconstruction of the input: principal-components learning and competitive learning. In both approaches, we first decide how economical the code should be and then modify the weights in the network to minimize the reconstruction error.

A principal-components learning strategy is based on the idea that if the activities of pairs of input units are correlated in some way, it is a waste of bits to describe each input activity separately. A more efficient approach is to extract and describe the principal components—that is, the components of variation shared by many input units. If we wish to discover, say, 10 of the principal components, then we need only a single layer of 10 hidden units.

Because such networks represent the input using only a small number of components, the code cost is low. And because the input can be reconstructed quite well from the principal components, the reconstruction cost is small.

One way to train this type of network is to force it to reconstruct an approximation to the input on a set of output units. Then back propagation can be used to minimize the difference between the actual output and the desired output. This process resembles supervised learning, but because the desired output is exactly the same as the input, no teacher is required.

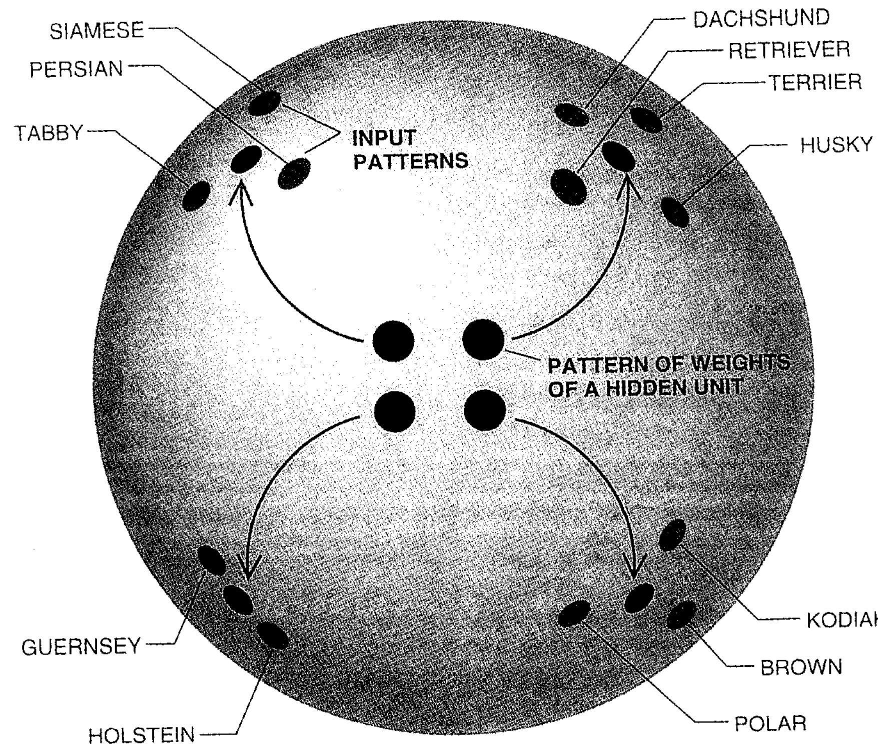


Figure 6.4
Competitive learning can be envisioned as a process in which each input pattern attracts the weight pattern of the closest hidden unit. Each input pattern represents a set of distinguishing features. The weight patterns of hidden units are adjusted so that they migrate slowly toward the closest set of input patterns. In this way, each hidden unit learns to represent a cluster of similar input patterns.

Many researchers, including Ralph Linsker of the IBM Thomas J. Watson Research Center and Erkki Oja of Lappeenranta University of Technology in Finland, have discovered alternative algorithms for learning principal components. These algorithms are more biologically plausible because they do not require output units or back propagation. Instead they use the correlation between the activity of a hidden unit and the activity of an input unit to determine the change in the weight.

When a neural network uses principal-components learning, a small number of hidden units cooperate in representing the input pattern. In contrast, in competitive learning, a large number of hidden units compete so that a single hidden unit is used to represent any particular input pattern. The selected hidden unit is the one whose incoming weights are most similar to the input pattern.

Now suppose we had to reconstruct the input pattern solely from our knowledge of which hidden unit was chosen. Our best bet would be to copy the pattern of incoming weights of the chosen hidden unit. To minimize the reconstruction error, we should move the pattern of weights of the winning hidden unit even closer to the input pattern. This is what competitive learning does. If the network is presented with training data that can be grouped into clusters of similar input patterns, each hidden unit learns to represent a different cluster, and its incoming weights converge on the center of the cluster.

Like the principal-components algorithm, competitive learning minimizes the reconstruction cost while keeping the code cost low. We can afford to use many hidden units because even with a million units it takes only 20 bits to say which one won.

In the early 1980s Teuvo Kohonen of Helsinki University introduced an important modification of the competitive learning algorithm. Kohonen showed how to make physically adjacent hidden units learn to represent similar input patterns. Kohonen's algorithm adapts not only the weights of the winning hidden unit but also the weights of the winner's neighbors. The algorithm's ability to map similar input patterns to nearby hidden units suggests that a procedure of this type may be what the brain uses to create the topographic maps found in the visual cortex.

Unsupervised learning algorithms can be classified according to the type of representation they create. In principal-components methods, the hidden units cooperate, and the representation of each input pattern is distributed across all of them. In competitive methods, the hidden units compete, and the representation of the input pattern is localized in the single hidden unit that is selected. Until recently, most work on unsupervised learning focused on one or another of these two techniques, probably because they lead to simple rules for changing the weights. But the most interesting and powerful algorithms probably lie somewhere between the extremes of purely distributed and purely localized representations.

Horace B. Barlow of the University of Cambridge has proposed a model in which each hidden unit is rarely active and the representation of each input pattern is distributed across a small number of selected hidden units. He and his co-workers have shown that this type of code can be learned by forcing hidden units to be uncorrelated while also ensuring that the hidden code allows good reconstruction of the input.

Unfortunately, most current methods of minimizing the code cost tend to eliminate all the redundancy among the activities of the hidden units. As a result, the network is very sensitive to the malfunction of a single hidden unit. This feature is uncharacteristic of the brain, which is generally not affected greatly by the loss of a few neurons.

The brain seems to use what are known as population codes, in which information is represented by a whole population of active neurons. That point was

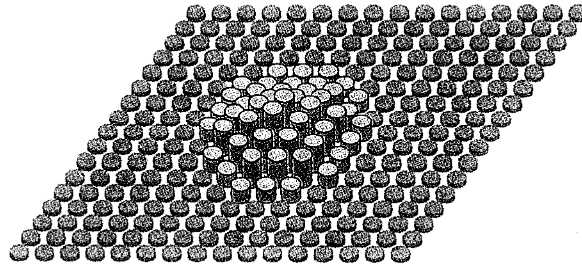


Figure 6.5
Population coding represents a multiparameter object as a bump of activity spread over many hidden units. Each disk represents an inactive hidden unit. Each cylinder indicates an active unit, and its height depicts the level of activity.

beautifully demonstrated in the experiments of David L. Sparks and his coworkers at the University of Alabama. While investigating how the brain of a monkey instructs its eyes where to move, they found that the required movement is encoded by the activities of a whole population of cells, each of which represents a somewhat different movement. The eye movement that is actually made corresponds to the average of all the movements encoded by the active cells. If some brain cells are anesthetized, the eye moves to the point associated with the average of the remaining active cells. Population codes may be used to encode not only eye movements but also faces, as shown by Malcolm P. Young and Shigeru Yamane at the RIKEN Institute in Japan in recent experiments on the inferior temporal cortex of monkeys.

For both eye movements and faces, the brain must represent entities that vary along many different dimensions. In the case of an eye movement, there are just two dimensions, but for something like a face, there are dimensions such as happiness, hairiness or familiarity, as well as spatial parameters such as position, size and orientation. If we associate with each face-sensitive cell the parameters of the face that make it most active, we can average these parameters over a population of active cells to discover the parameters of the face being represented by that population code. In abstract terms, each face cell represents a particular point in a multidimensional space of possible faces, and any face can then be represented by activating all the cells that encode very similar faces, so that a bump of activity appears in the multidimensional space of possible faces.

Population coding is attractive because it works even if some of the neurons are damaged. It can do so because the loss of a random subset of neurons has little effect on the population average. The same reasoning applies if some neurons are overlooked when the system is in a hurry. Neurons communicate by sending discrete spikes called action potentials, and in a very short time in-

terval, many of the "active" neurons may not have time to send a spike. Nevertheless, even in such a short interval, a population code in one part of the brain can still give rise to an approximately correct population code in another part of the brain.

At first sight, the redundancy in population codes seems incompatible with the idea of constructing internal representations that minimize the code cost. Fortunately, we can overcome this difficulty by using a less direct measure of code cost. If the activity that encodes a particular entity is a smooth bump in which activity falls off in a standard way as we move away from the center, we can describe the bump of activity completely merely by specifying its center. So a fairer measure of code cost is the cost of describing the center of the bump of activity plus the cost of describing how the actual activities of the units depart from the desired smooth bump of activity.

Using this measure of the code cost, we find that population codes are a convenient way of extracting a hierarchy of progressively more efficient encodings of the sensory input. This point is best illustrated by a simple example. Consider a neural network that is presented with an image of a face. Suppose the network already contains one set of units dedicated to representing noses, another set for mouths and another set for eyes. When it is shown a particular face, there will be one bump of activity in the nose units, one in the mouth units and two in the eye units. The location of each of these activity bumps represents the spatial parameters of the feature encoded by the bump. Describing the four activity bumps is cheaper than describing the raw image, but it would obviously be cheaper still to describe a single bump of activity in a set of face units, assuming of course that the nose, mouth and eyes are in the correct spatial relations to form a face.

This raises an interesting issue: How can the network check that the parts are correctly related to one another to make a face? Some time ago Dana H. Ballard of the University of Rochester introduced a clever technique for solving this type of problem that works nicely with population codes.

If we know the position, size and orientation of a nose, we can predict the position, size and orientation of the face to which it belongs because the spatial relation between noses and faces is roughly fixed. We therefore set the weights in the neural network so that a bump of activity in the nose units tries to cause an appropriately related bump of activity in the face units. But we also set the thresholds of the face units so that the nose units alone are insufficient to activate the face units. If, however, the bump of activity in the mouth units also tries to cause a bump in the same place in the face units, then the thresholds can be overcome. In effect, we have checked that the nose and mouth are correctly related to each other by checking that they both predict the same spatial parameters for the whole face.

This method of checking spatial relations is intriguing because it makes use of the kind of redundancy between different parts of an image that un-

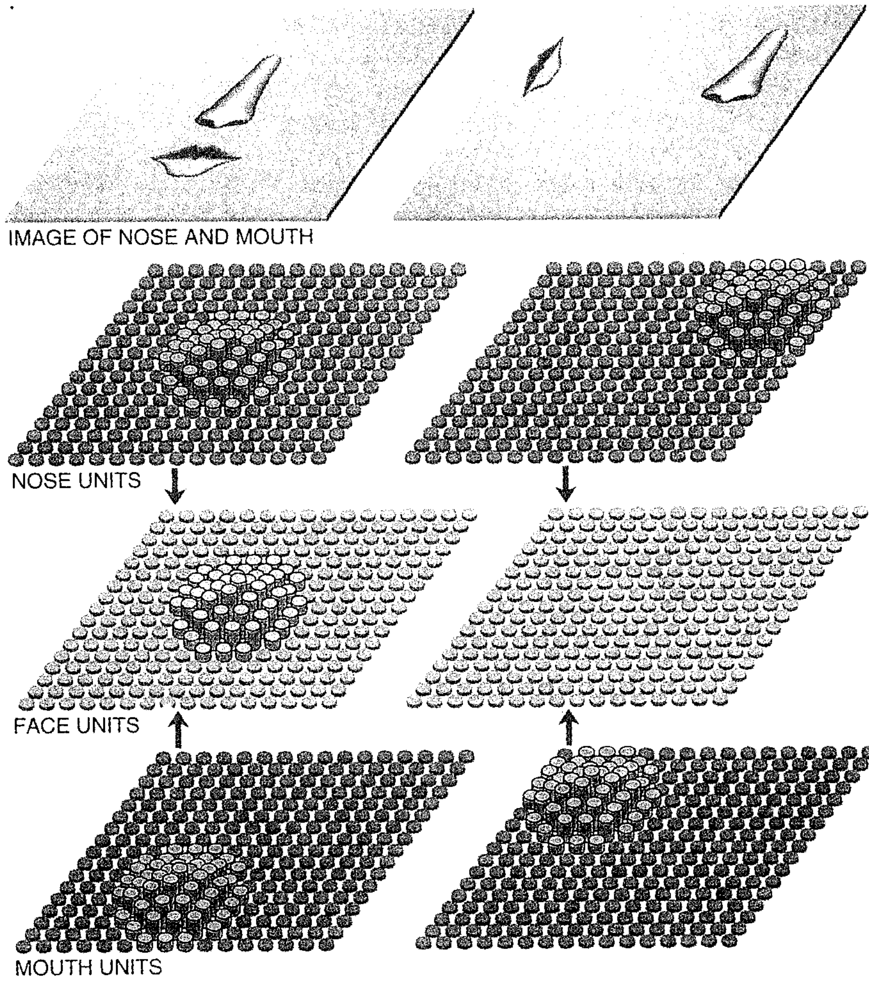


Figure 6.6
Bumps of activity in sets of hidden units represent the image of a nose and a mouth. The population codes will cause a bump in the face units if the nose and mouth have the correct spatial relation (left). If not, the active nose units will try to create a bump in the face units at one location while the active mouth units will do the same at a different location. As a result, the input activity to the face units does not exceed a threshold value, and no bump is formed in the face units (right).

supervised learning should be good at finding. It therefore seems natural to try to use unsupervised learning to discover hierarchical population codes for extracting complex shapes. In 1986, Eric Saund of M.I.T. demonstrated one method of learning simple population codes for shapes. It seems likely that with a clear definition of the code cost, an unsupervised network will be able to discover more complex hierarchies by trying to minimize the cost of coding the image. Richard Zemel and I at the University of Toronto are now investigating this possibility.

By using unsupervised learning to extract a hierarchy of successively more economical representations, it should be possible to improve greatly the speed of learning in large multilayer networks. Each layer of the network adapts its incoming weights to make its representation better than the representation in the previous layer, so weights in one layer can be learned without reference to weights in subsequent layers. This strategy eliminates many of the interactions between weights that make back-propagation learning very slow in deep multilayer networks.

All the learning procedures discussed thus far are implemented in neural networks in which activity flows only in the forward direction from input to output even though error derivatives may flow in the backward direction. Another important possibility to consider is networks in which activity flows around closed loops. Such recurrent networks may settle down to stable states, or they may exhibit complex temporal dynamics that can be used to produce sequential behavior. If they settle to stable states, error derivatives can be computed using methods much simpler than back propagation.

Although investigators have devised some powerful learning algorithms that are of great practical value, we still do not know which representations and learning procedures are actually used by the brain. But sooner or later, computational studies of learning in artificial neural networks will converge on the methods discovered by evolution. When that happens, a lot of diverse empirical data about the brain will suddenly make sense, and many new applications of artificial neural networks will become feasible.