CONNECTIONIST IDEAS AND ALGORITHMS

Kevin Knight

In our quest to build intelligent machines, we have but one naturally occurring model: the human brain. It follows that one natural idea for artificial intelligence (AI) is to simulate the functioning of the brain directly on a computer. Indeed, the idea of building an intelligent machine out of artificial neurons has been around for quite some time. Some early results on brain-line mechanisms were achieved by [18], and other researchers pursued this notion through the next two decades, e.g., [1, 4, 19, 21, 24]. Research in neural networks came to a virtual halt in the 1970s, however, when the networks under study were shown to be very weak computationally. Recently, there has been a resurgence of interest in neural networks. There are several reasons for this, including the appearance of faster digital computers on which to simulate larger networks, interest in building massively parallel computers, and most importantly, the discovery of powerful network learning algorithms.

The new neural network architectures have been dubbed *connectionist* architectures. For the most part, these architectures are not meant to duplicate the operation of the human brain, but rather receive inspiration from known facts about how the brain works. They are characterized by

- Large numbers of very simple neuron-like processing elements;
- Large numbers of weighted connections between the elements—the weights on the connections encode the knowledge of a network;
- Highly parallel, distributed control; and
- Emphasis on learning internal representations automatically.

Connectionist researchers conjecture that thinking about computation in terms of the brain metaphor rather than the digital computer metaphor will lead to insights into the nature of intelligent behavior.

Computers are capable of amazing feats. They can effortlessly store vast quantities of information. Their circuits operate in nanoseconds. They can perform extensive arithmetic calculations without error. Humans cannot approach these capabilities. On the other hand, humans routinely perform simple tasks such as walking, talking, and commonsense reasoning. Current AI systems cannot do any of these things better than humans. Why not? Perhaps the structure of the brain is somehow suited to these tasks, and not suited to tasks like high-speed arithmetic calculation. Working under constraints suggested by the brain may make traditional computation more difficult, but it may lead to solutions to AI problems that would otherwise be overlooked.

What constraints, then, does the brain offer us? First of all, individual neurons are extremely slow devices when compared to their counterparts in digital computers. Neurons operate in the millisecond range, an eternity to a VLSI designer. Yet, humans can perform extremely complex tasks, like interpreting a visual scene or understanding a sentence, in just a tenth of a second. In other words, we do in about a hundred steps what current computers cannot do in ten million steps. How can this be possible? Unlike a conventional computer, the brain contains a huge number of processing elements that act in parallel. This suggests that in our search for solutions, we look for massively parallel algorithms that require no more than 100 processing steps [9].

Also, neurons are failure-prone devices. They are constantly dying (you have certainly lost a few since you began reading this article), and their firing patterns are irregular. Components in digital computers, on the other hand, must operate perfectly. Why? Such components store bits of information that are available nowhere else in the computer: the failure of one component means a loss of information. Suppose that we built AI programs that were not sensitive to the failure of a few components, perhaps by using redundancy and distributing information across a wide range of components? This would open the possibility of very large-scale implementations. With current technology, it is far easier to build a *billion-component* integrated circuit in which 95 percent of the components work correctly than it is to build a perfectly functioning *million-component* machine [8].

Another thing people seem to be able to do better than computers is handle fuzzy situations. We have very large memories of visual, auditory, and problem-solving episodes, and one key operation in solving new problems is finding closest matches to old situations. Inexact matching is something brain-style models seem to be good at, because of the diffuse and fluid way in which knowledge is represented.

The idea behind connectionism, then, is that we may see significant advances in AI if we approach

problems from the point of view of brain-style computation rather than rule-based symbol manipulation. At the end of this article, we will look more closely at the relationship between connectionist and symbolic AI.

Hopfield Networks

The history of AI is curious. The first problems attacked by AI researchers were problems like chess and theorem proving, because these were thought to require the essence of intelligence. Vision and language understanding—processes easily mastered by five-year-olds—were not thought to be difficult. These days, we have expert chess programs, and expert medical diagnosis programs, but no programs that can match the basic perceptual skills of a child. Neural network researchers contend that there is a basic mismatch between standard computer information-processing technology and the technology used by the brain.

In addition to these perceptual tasks, AI is just starting to grapple with fundamental problems in memory and commonsense reasoning. Computers are notorious for their lack of common sense. Many people believe that common sense derives from our massive store of knowledge, and more importantly, our ability to access relevant knowledge quickly, effortlessly, and at the right time.

When we read the description "gray, large, mammal," we automatically think of elephants and their associated features. We access our memories by *content*. In traditional implementations, access by content involves expensive searching and matching procedures. Massively parallel networks suggest a more efficient method.

A neural network, introduced by Hopfield [12], proposed one theory of memory. A Hopfield network has the following interesting features:

- *Distributed representation.* A memory is stored as a pattern of activation across a set of processing elements. Furthermore, memories can be superimposed upon one another—different memories are represented by different patterns over the *same* set of processing elements.
- *Distributed, asynchronous control.* Each processing element makes decisions based only on its own local situation. All of these local actions add up to a global solution.
- *Content-addressable memory.* A number of patterns can be stored in a network. To retrieve a pattern, we need only specify a portion of it. The network automatically finds the closest match.
- *Fault tolerance.* If a few of the processing elements misbehave or fail completely, the network will still function properly.

How are these features achieved? A simple Hopfield net is shown in Figure 1. Processing elements, or *units*, are always in one of two states, active or inactive. Units are connected to each other with weighted, symmetric connections. A positive connection indicates that the two units tend to activate each other. A negative connection allows an active unit to deactivate a neighboring unit.

The network operates as follows. A random unit is chosen. If any of its neighbors are active, the unit computes the sum of the weights on the connections to those active neighbors. If the sum is positive, the unit becomes active, otherwise it becomes inactive. Another random unit is chosen, and the process repeats until the network reaches a stable state (i.e., until no more units can change state). This process is called *parallel relaxation*. If the network starts in the state shown in Figure 1, the unit in the lower left corner will tend to activate the unit above it. This unit, in turn, will attempt to activate the unit above it, but the inhibitory connection from the upper-right unit will foil this attempt, and so on.

This network has only four distinct stable states. They are shown in Figure 2. Given *any* initial state, the network will necessarily settle into one of these four configurations.[1] The network can be thought of as storing the patterns in Figure 2. Hopfield's major contribution was to show that given any set of weights and any initial state, his parallel relaxation algorithm would eventually steer the network into a stable state. There can be no divergence or oscillation.

The network can be used as a *content-addressable memory* by setting the activities of the units to correspond to a partial pattern. The network will then settle into the stable state that best matches the partial pattern. An example is shown in Figure 3.

Parallel relaxation is nothing more than search, albeit of a style not usually employed in AI. It is useful to think of the various states of a network as forming a search space as in Figure 4. A randomly chosen state will ultimately transform itself into one of the local minima namely the nearest stable state. This is how we get the content-addressable behavior. We also get an error-correcting behavior. Suppose we read the description, "gray, large, fish, eats plankton." We imagine a whale, even though we know that a whale is a mammal, not a fish. Even if the initial state contains inconsistencies, a Hopfield network will settle into the solution that violates the fewest constraints offered by the inputs. Traditional match-and-retrieve procedures are less forgiving.

Now, suppose a unit occasionally fails, say, by becoming active or inactive when it should not. This causes no major problem: surrounding units will quickly set it straight again. It would take the unlikely concerted effort of many errant units to push the network into the wrong stable state. In networks of thousands of more highly interconnected units, such fault tol-

[1]The stable state in which all units are inactive can only be reached if it is also the initial state.

erance is even more apparent—units and connections can disappear completely without adversely affecting the overall behavior of the network.

As we can see, parallel networks of simple elements can compute interesting things. The next important question is: What is the relationship between the weights on the network's connections and the local minima into which it settles? In other words, if the weights encode the "knowledge" of a particular network, how is that knowledge acquired? Knowledge acquisition is a difficult problem in AI, and one attractive feature of connectionist architectures is that their method of representation (namely, real-valued connection weights) lends itself very nicely to automatic learning.

In the next section, we will look closely at learning in several neural network models, including perceptrons, backpropagation networks, and Boltzmann machines.

Learning in Neural Networks

The *perceptron*, an invention of [24] was one of the earliest neural network models. A perceptron models a neuron by taking a weighted sum of its inputs and sending an output 1 if the sum is greater than some adjustable threshold value (otherwise it sends 0). Figure 5 shows the device.

The inputs (x_1, x_2 . . . x_n) and connection weights (w_1, w_2 . . . w_n) in the figure are typically real values, both positive and negative. If the presence of some feature x_i tends to cause the perceptron to fire, the weight w_i will be positive; if the feature x_i inhibits the perceptron, the weight w_i will be negative. The perceptron itself consists of the weights, the summation processor, and the adjustable threshold processor. Learning is a process of modifying the values of the weights and the threshold. It is convenient to implement the threshold as just another weight w_0 (as in Figure 6). This weight can be thought of as the propensity of the perceptron to

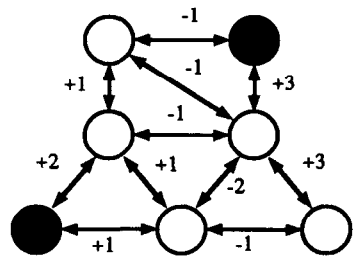


FIGURE 1. A simple Hopfield network. Units have binary states (black represents "on" and white represents "off"), and connection weights are symmetric.

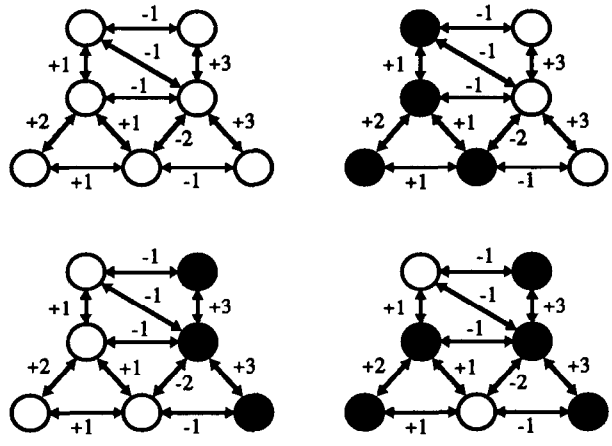


FIGURE 2. The four stable states of a particular Hopfield net.

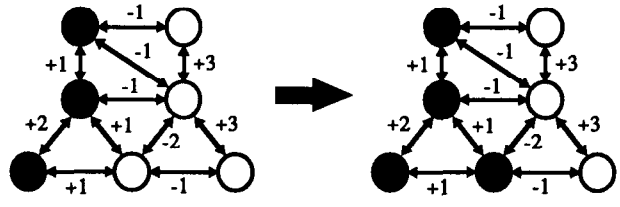


FIGURE 3. A Hopfield net as a model of content-addressable memory. To retrieve a pattern, we need only supply a portion of it.

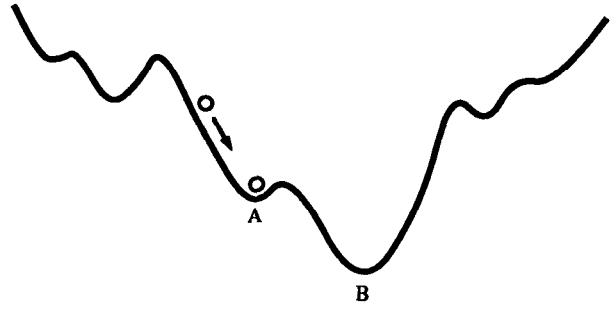


FIGURE 4. A simplified view of what a Hopfield net computes.

fire irrespective of its inputs. The perceptron of Figure 6 fires if the weighted sum is greater than zero.

A perceptron computes a binary function of its input. Multiple perceptrons can be combined to compute more complex functions, as shown in Figure 7.

Such a group of perceptrons can be trained on sample input/output pairs until it learns to compute the correct function. The amazing property of perceptron learning is this: whatever a perceptron can compute, it can *learn* to compute! We will demonstrate this in a moment. At the time perceptrons were invented, many people speculated that intelligent systems could be constructed out of perceptrons (see Figure 8).

Since the perceptrons of Figure 7 are independent of one another, they can be separately trained. Let us concentrate on what a single perceptron can learn to do. Consider the pattern classification problem shown in Figure 9. Given values for x_1 and x_2, we want to train a perceptron to output 1 if it thinks the input belongs to the class of white dots, and 0 if it thinks the input belongs to the class of black dots. We have no explicit rule to guide us; we must induce a rule from a set of training instances. We will now see how perceptrons can learn to solve such problems.

First, it is necessary to take a close look at what the perceptron computes. Let $\vec{x}$ be an input vector $(x_1, x_2 \ldots x_n)$. Notice that the weighted summation function $g(x)$ and the output function $o(x)$ can be defined as:

$$g(x) = \sum_{i=0}^{x} w_i x_i$$

$$o(x) = \begin{cases} 1 & \text{if } g(x) > 0 \\ 0 & \text{if } g(x) < 0 \end{cases}$$

Consider the case where we have only two inputs (as in Figure 9). Then:

$$g(x) = w_0 + w_1 x_1 + w_2 x_2$$

If $g(x)$ is exactly 0, the perceptron cannot decide whether to fire or not. A slight change in inputs could cause the device to go either way. If we solve the equation $g(x) = 0$, we get the equation for a line:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

The location of the line is completely determined by the weights w_0, w_1, and w_2. If an input vector lies on one side of the line, the perceptron will output 1; if it lies on the other side, the perceptron will output 0. A line that correctly separates the training instances corresponds to a perfectly functioning perceptron. Such a line is called a *decision surface.* In perceptrons with many inputs, the decision surface will be a hyperplane through the multidimensional space of possible input vectors. The problem of learning is one of locating an appropriate decision surface.

We will present a formal learning algorithm in a moment. For now, consider the informal rule:

> If the perceptron fires when it should not fire, make each w_i smaller by an amount proportional to x_i. If the perceptron fails to fire when it should fire, make each w_i larger by a similar amount.

Suppose we want to train a three-input perceptron to fire only when its first input is on. If the perceptron fails to fire in the presence of an active x_1, we will increase w_1 (and we may increase other weights). If the perceptron fires incorrectly, we will end up decreasing weights that are not w_1. In addition, w_0 will find a value based on the total number of incorrect firings versus incorrect misfirings. Soon, w_1 will become large enough to overpower w_0, while w_2 and w_3 will not be powerful enough to fire the perceptron, even in the presence of both x_2 and x_3.

Now let us return to the functions $g(x)$ and $o(x)$. While the sign of $g(x)$ is critical to determining whether the perceptron will fire, the magnitude is also important. The absolute value of $g(x)$ tells how far a given input vector $\vec{x}$ lies from the decision surface. This gives us a way of characterizing how good a set of weights is. Let $\vec{w}$ be the weight vector $(w_0, w_1 \ldots w_n)$, and let X be the subset of training instances *misclassified* by the current set of weights. Then define the *Perceptron Criterion Function*, $J(\vec{w})$, to be the sum of the distances of the misclassified input vectors from the decision surface:

$$J(\vec{w}) = \sum_{\vec{x} \in X} \left| \sum_{i=0}^{n} w_i x_i \right| = \sum_{\vec{x} \in X} |\vec{w}\vec{x}|$$

To create a better set of weights than the current set, we would like to reduce $J(\vec{w})$. Ultimately, if all inputs are classified correctly, $J(\vec{w}) = 0$.

How do we go about minimizing $J(\vec{w})$? We can use a form of local search known as *gradient descent.*[2] For our current purposes, think of $J(\vec{w})$ as defining a surface in the space of all possible weights. Such a surface might look like the one in Figure 10.

In the figure, weight w_0 should be part of the weight space, but is omitted here because it is easier to visualize J in only three dimensions. Now, some of the weight vectors constitute solutions, in that a perceptron with a solution vector will classify all of its inputs correctly. Note that there are an infinite number of solution vectors. For any solution vector $\vec{w}_s$, we know that $J(\vec{w}_s) = 0$. Suppose we begin with a random weight vector $\vec{w}$ that is not a solution vector. We want to slide down the J surface. There is a mathematical method for doing this—we compute the gradient of the function $J(\vec{w})$. Before we derive the gradient function, we will reformulate the Perceptron Criterion Function to remove the absolute value sign:

$$J(\vec{w}) = \sum_{\vec{x} \in X} \vec{w} \begin{cases} \vec{x} \text{ if } \vec{x} \text{ is misclassified} \\ \quad \text{as a negative example} \\ -\vec{x} \text{ if } \vec{x} \text{ is misclassified} \\ \quad \text{as a positive example} \end{cases}$$

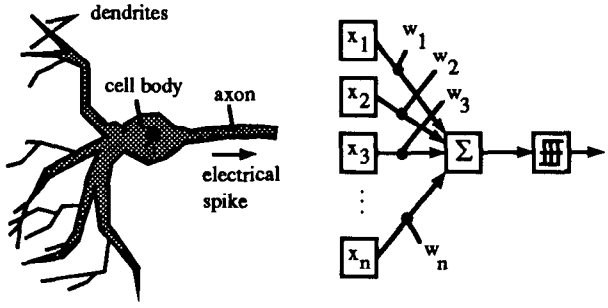


FIGURE 5. A neuron and a perceptron.

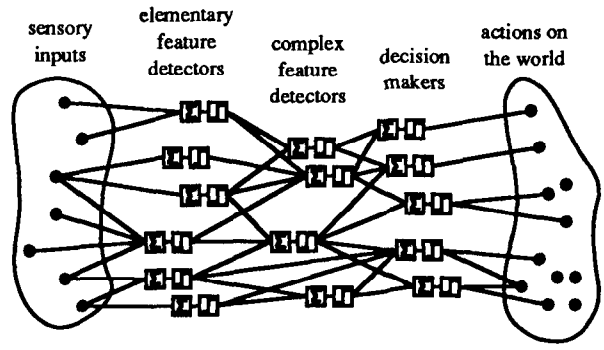


FIGURE 8. An early notion of an intelligent system built out of trainable perceptrons.

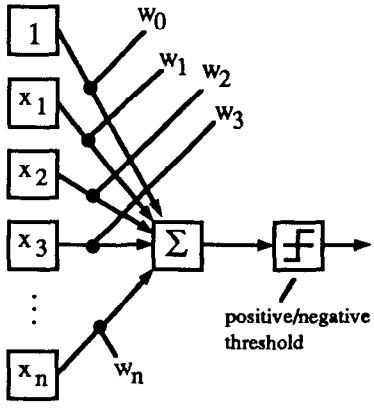


FIGURE 6. Perceptron with adjustable threshold implemented as additional weight w_0.

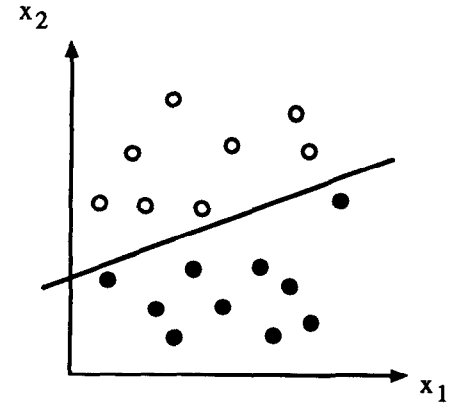


FIGURE 9. A pattern classification problem. This problem is *linearly separable*, because we can draw a line that separates one class from another.

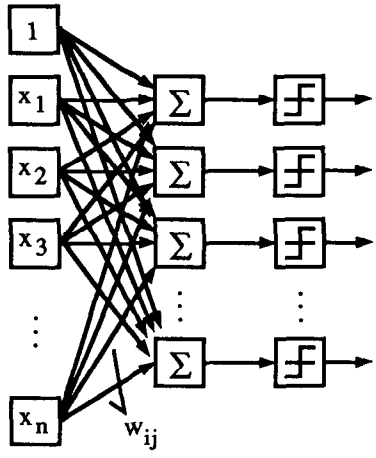


FIGURE 7. A perceptron with many inputs and many outputs.

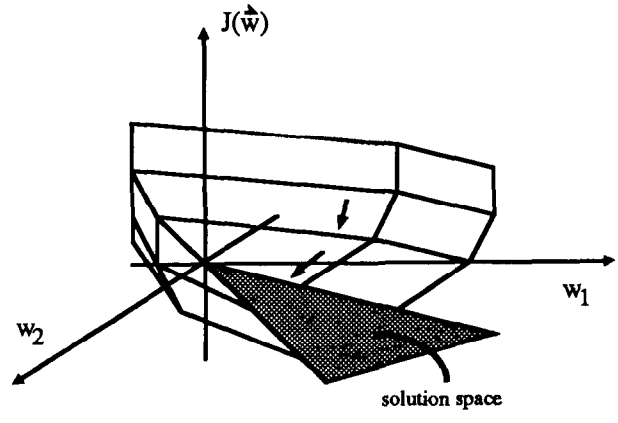


FIGURE 10. Adjusting the weights by gradient descent, minimizing $J(\vec{w})$. (Weight w_0 is omitted for clarity).

Recall that X is the set of misclassified input vectors.

Now, here is ∇J, the gradient of $J(\vec{w})$ with respect to the weight space:

$$\nabla J(\vec{w}) = \sum_{\vec{x} \in X} \begin{cases} \vec{x} \text{ if } \vec{x} \text{ is misclassified as a negative example} \\ -\vec{x} \text{ if } \vec{x} \text{ is misclassified as a positive example} \end{cases}$$

The gradient is a vector that tells us the direction to move in weight space in order to reduce $J(\vec{w})$. In order to find a solution weight vector, we simply change the weights in the direction of the gradient, recompute $J(\vec{w})$ recompute the new gradient, and iterate until $J(\vec{w}) = 0$. The rule for updating the weights at time $t + 1$ is:

$$\vec{w}_{t+1} = \vec{w}_t + \eta \nabla J$$

Or in expanded form:

$$\vec{w}_{t+1} = \vec{w}_t + \eta \sum_{\vec{x} \in X} \begin{cases} \vec{x} \text{ if } \vec{x} \text{ is misclassified as a negative example} \\ -\vec{x} \text{ if } \vec{x} \text{ is misclassified as a positive example} \end{cases}$$

η is a scale factor that tells us how far to move in the direction of the gradient. A small η will lead to slower learning, but a large η may cause a move through weight space that "overshoots" the solution vector. Taking η to be a constant gives us what is usually called the "fixed-increment perceptron learning algorithm":

Algorithm: Fixed-Increment Perceptron Learning

Given: a classification problem with n input features $(x_1, x_2 \ldots x_n)$ and two output classes.

Compute: a set of weights $w_0, w_1, w_2 \ldots w_n)$ that will cause a perceptron to fire whenever the input falls into the first output class.

1. Create a perceptron with $n + 1$ inputs and $n + 1$ weights, where the extra input x_0 is always set to 1.
2. Initialize the weights $(w_0, w_1 \ldots w_n)$ to random real values.
3. Iterate through the training set, collecting all of the examples *misclassified* by the current set of weights.
4. If all examples are classified correctly, output the weights and quit.
5. Otherwise, compute the vector sum S of the misclassified input vectors, where each vector has the form $(x_0, x_1 \ldots x_n)$. In creating the sum, add to S a vector $\vec{x}$ if $\vec{x}$ is an input for which the perceptron incorrectly *fails to fire*, but add vector $-\vec{x}$ if $\vec{x}$ is an input for which the perceptron incorrectly *fires*. Multiply the sum by a scale factor η.
6. Modify the weights $(w_0, w_1 \ldots w_n)$ by adding the elements of the vector S to them. Go to step 3.

The perceptron learning algorithm is a search algorithm. It begins in a random initial state and finds a solution state. The search space is simply all of the possible assignments of real values to the weights of the perceptron, and the search strategy is gradient descent.

So far, we have seen two search methods employed by neural networks: *gradient descent* in perceptrons and *parallel relaxation* in Hopfield networks. It is important to understand the relation between the two. Parallel relaxation is a problem-solving strategy, analogous to state space search in symbolic AI. Gradient descent is a learning strategy, analogous to inductive techniques in symbolic AI. In both symbolic and connectionist AI, learning is viewed as a type of problem solving, and this is why search is useful in learning. But the ultimate goal of learning is to get a system into a position where it can solve problems better. Do not confuse learning algorithms with others.

The *Perceptron Convergence Theorem*, due to Rosenblatt [24], guarantees that the perceptron will find a solution state (i.e., it will learn to classify any linearly separable set of inputs). Figure 11 shows a perceptron learning to classify the instances of Figure 9. Remember that every set of weights specifies some decision surface—in this case some two-dimensional line.

The introduction of perceptrons in the late 1950s created a great deal of excitement in the research community. Here was a device that strongly resembled a neuron and for which well-defined learning algorithms were available. There was much speculation about how intelligent systems could be constructed from perceptron building blocks. The book, *Perceptrons*, [20] put an end to such speculation by analyzing the computational capabilities of the devices. The authors, Minsky and Papert, noticed that while the Convergence Theorem guaranteed correct classification of linearly separable data, most problems do not supply such nice data. Indeed, the perceptron is incapable of learning to solve some very simple problems. One example given in the book is the exclusive-or (XOR) problem: Given two binary inputs, output 1 if *exactly* one of the inputs is on, and output 0 otherwise. We can view XOR as a pattern-classification problem in which there are four patterns and two possible outputs (see Figure 12).

The perceptron cannot learn a linear decision surface to separate these different outputs, *because no such decision surface exists.* No single line can separate the "1" outputs from the "0" outputs. Minsky and Papert gave a number of problems with this property: telling whether a line drawing is connected, separating figure from ground in a picture, etc. Notice that the deficiency here is not in the perceptron learning algorithm, but in the way the perceptron represents knowledge.

If we could draw an elliptical decision surface, we could encircle the two "1" outputs in the XOR space. However, perceptrons are incapable of modeling such surfaces. Another idea is to employ *two* separate line-drawing stages. We

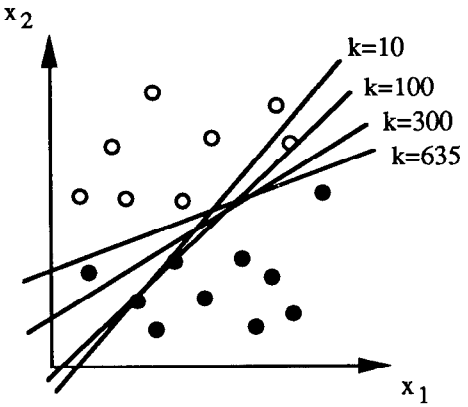
could draw one line to isolate the point ($x_1 = 1, x_2 = 1$) and then another line to divide the remaining three points into two categories. Using this idea, we can construct a multilayer perceptron (a series of perceptrons) to solve the problem. Such a device is shown in Figure 13.

Note how the output of the first perceptron serves as one of the inputs to the second perceptron, with a large, negatively weighted connection. If the first perceptron sees the input ($x_1 = 1, x_2 = 1$) it will send a massive inhibitory pulse to the second perceptron, causing that unit to output 0 regardless of its other inputs. If either of inputs is 0, the second perceptron gets no inhibition from the first perceptron, and it outputs 1 if either of the inputs is 1.

The use of multilayer perceptrons, then, solves our knowledge representation problem. However, it introduces a serious learning problem: the Convergence Theorem does not extend to multilayer perceptrons. The perceptron learning algorithm can correctly adjust weights between inputs and outputs, but it cannot adjust weights *between* perceptrons. In Figure 13, the inhibitory weight "−9.0" was hand-coded, not learned. At the time *Perceptrons* was published, no one knew how multilayer perceptrons could be made to learn. In fact, Minsky and Papert speculated:

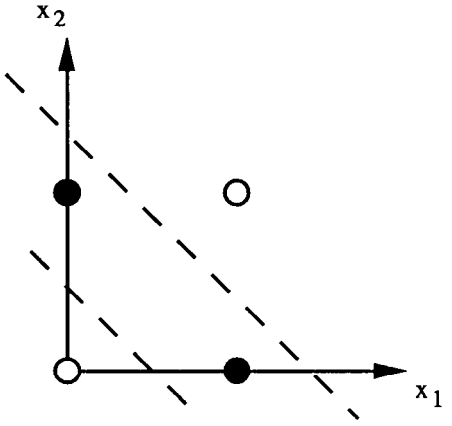
> The perceptron . . . has many features that attract attention: its linearity, its intriguing learning theorem . . . there is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile.

Despite the identification of this important research problem, actual research in perceptron learning came to a halt in the 1970s. The field saw little interest until the 1980s, when several learning pro-



k	w_0	w_1	w_2
10	0.41	-0.17	0.14
100	0.22	-0.14	0.11
300	-0.10	-0.08	0.07
635	-0.49	-0.10	0.14

FIGURE 11. A perceptron learning to solve a classification problem. k is the number of passes through the training data (i.e., the number of iterations of steps 3 through 6 of the fixed-increment perceptron learning algorithm).



x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 12. A classification problem, XOR, that is *not* linearly separable.

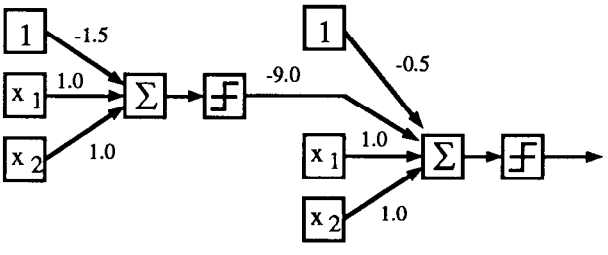


FIGURE 13. A multilayer perceptron that solves the XOR problem.

cedures for multilayer perceptrons—also called multilayer networks—were proposed. The next few sections are devoted to such learning procedures.

Backpropagation Networks

As suggested by Figure 8 and the *Perceptrons* critique, the ability to train multilayer networks is an important step in the direction of building intelligent machines out of neuron-like components. Let us reflect for a moment on why this is so. Our goal is to take a relatively amorphous mass of neuron-like elements and teach it to perform useful tasks. We would like it to be fast and resistant to damage. We would like it to generalize from the inputs it sees. We would like to build these neural masses on a very large scale, and we would like them to be able to learn efficiently. Perceptrons got us part of the way there, but we say that they were too weak computationally. So we turn to more complex, multilayer networks.

What can a multilayer network compute? The simple answer is: *anything*! Given a set of inputs, we can use summation/threshold units as simple AND, OR, and NOT gates by appropriately setting the threshold and connection weights. We can build any arbitrary combinational circuit out of such units. In fact, if we are allowed to use feedback loops, we can build a general-purpose computer with them.

The major problem is learning. The knowledge representation system employed by neural nets is quite opaque: they *must* learn their own representations because programming them by hand is impossible. Perceptrons had the nice property that whatever they could compute, they could learn to compute. Does this property extend to multilayer networks? The answer is yes, sort of. Backpropagation is a step in that direction.

It will be useful to deal first with a subclass of multilayer networks, namely fully connected, layered, feedforward networks. A sample of such a network is shown in Figure 14. This network has three layers, although it is possible and sometimes useful to have more. Activations flow from the input layer through a hidden layer, then on to the output layer. Each unit in one layer is connected in the forward direction to every unit in the next layer. As usual, the knowledge of the network is encoded in the weights on connections between units. In contrast to the parallel relaxation method used by Hopfield nets, backpropagation networks perform a simpler computation. Because activations flow in only one direction, there is no need for an iterative relaxation process. The activation levels of the units in the output layer determine the output of the network.

The existence of hidden units allows the network to develop complex feature detectors, or *internal representations*. Figure 15 shows the application of a three-layer network to the problem of recognizing digits. The two-dimensional grid containing the numeral "7" forms the input layer. A single hidden unit might be strongly activated by a horizontal line in the input, or perhaps a diagonal. The important thing to note is that the behavior of these hidden units is automatically learned, not preprogrammed. In Figure 15, the input grid appears to be laid out in two dimensions, but the fully connected network is unaware of this 2-D structure. Because this structure can be important, many networks permit their hidden units to maintain only local connections to the input layer (e.g., a different 4-by-4 sub-grid for each hidden unit).

The hope in attacking problems like handwritten character recognition is that the neural network will not only learn to classify the inputs it is trained on, but will *generalize* and be able to classify inputs that it has not yet seen. We will return to generalization in the next section.

It seems reasonable at this point to express the following: "All neural nets seem to be able to do is classification. Hard AI problems like planning, natural language parsing, and theorem proving are not simply classification tasks, so how do connectionist models address these problems?" Most of the problems we will see in this article are indeed classification problems, because these are the problems that neural networks are best suited to handle at present. A major limitation of current network formalisms is their way of dealing with phenomena that involve time. This limitation is lifted to some degree in work on recurrent networks (e.g. [14]), but for now, we will concentrate on classification problems.

Let us now return to backpropagation networks. The unit in a backpropagation network requires a slightly different activation function from the perceptron. Both functions are shown in Figure 16. A backpropagation unit still sums up its weighted inputs, but unlike the perceptron, it produces a real value between 0 and 1 as output, based on a sigmoid (or S-shaped) function. Let *sum* be the weighted sum of the inputs to a unit. The equation for the unit's output is given by:

$$output = \frac{1}{1 + e^{-sum}}$$

Like a perceptron, a backpropagation network typically starts out with a random set of weights. The network adjusts its weights each time it sees an input/output pair. Each pair requires two stages: a forward pass and a backward pass. The forward pass involves presenting a sample input to the network and letting activations flow until they reach the output layer. During the backward pass, the network's actual output is compared to the target output, and error estimates are computed for the output units. The weights connected to the output units can be adjusted in order to reduce those errors. We can then use the error estimates of the output units to derive error estimates for the units in

the hidden layers. Finally, errors are propagated back to the connections stemming from the input units.

Unlike the perceptron learning algorithm of the last section, the backpropagation algorithm usually updates its weights incrementally, after seeing each input/output pair. After it has seen all of the input/output pairs (and adjusted its weights that many times), we say that one *epoch* has been completed. Training a backpropagation network usually requires many epochs.

Refer back to Figure 14 for the basic structure on which the following algorithm is based.

Algorithm: Backpropagation

Given: A set of input/output vector pairs.

Compute: A set of weights for a three-layer network that maps inputs onto corresponding outputs.

1. Let A be the number of units in the input layer, as determined by the length of the training input vectors. Let C be the number of units in the output layer. Now choose B, the number of units in the hidden layer.[3] As shown in Figure 14, the input and hidden layers each have an extra unit used for thresholding; therefore, the units in these layers will sometimes be indexed by the ranges $(0 \ldots A)$ and $(0 \ldots B)$. We denote the activation levels of the units in the input layers by x_i, in the hidden layer by h_j, and in the output layer by o_j. Weights connecting the input layer to the hidden layer are denoted by $w1_{ij}$, where subscript i indexes the input units, and j indexes the hidden units. Likewise, weights connecting

[2]Gradient descent is the same thing as *hill climbing*, modulo a change in sign. Hill climbing is one of the *weak methods* often used in symbolic AI.

[3]Successful large-scale networks have used input-hidden-output topologies like 960-9-45 [23], 203-80-26 [28], and 459-24-24-1 [29]. A larger hidden layer results in a more powerful network, but too much power may be undesirable, as we will see later.

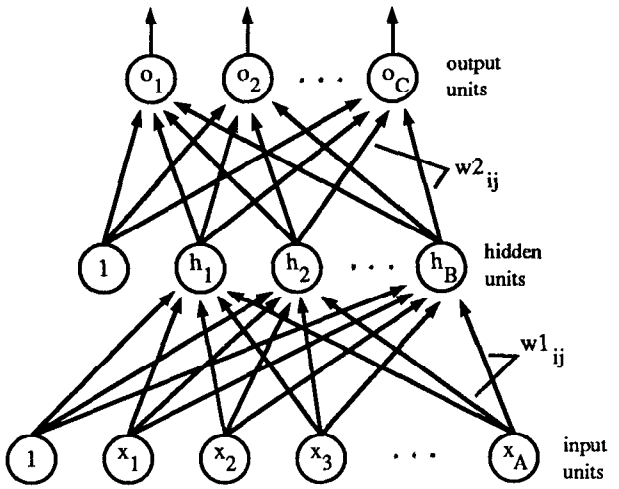


FIGURE 14. A multilayer network. In this diagram x_i, h_i, and o_i represent unit activation levels of input, hidden, and output units. Weights on connections between the input and hidden layers are denoted here by $w1_{ij}$, while weights on connections between the hidden and output layers are denoted by $w2_{ij}$.

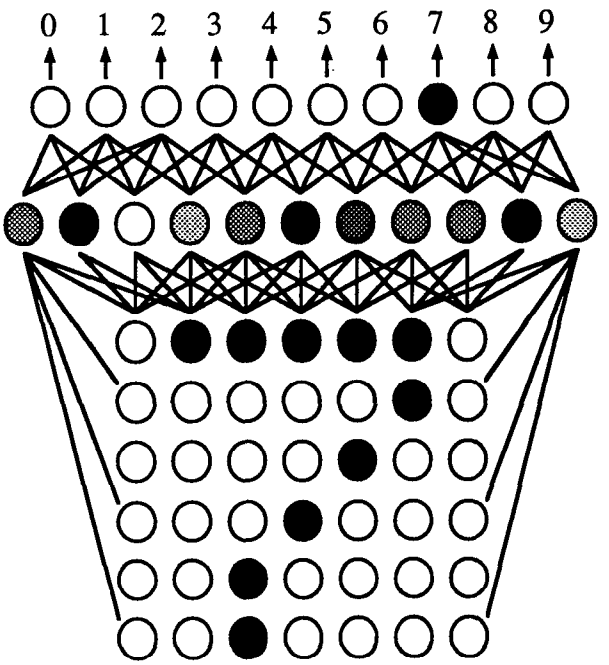


FIGURE 15. Using a multilayer network to learn to classify handwritten digits. The hidden units learn to recognize important features in the input.

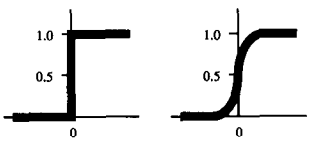


FIGURE 16. The stepwise activation function of the perceptron (left), and the sigmoid activation function of the backpropagation unit (right). The sigmoid function is continuous and differentiable, features required by the backpropagation algorithm.

the hidden layer to the output layer are denoted by $w2_{ij}$ with i indexing to hidden units and j indexing output units.

2. Initialize the weights in the network. Each weight should be set randomly to number between -0.1 and 0.1.

$$w1_{ij} = random(-0.1,0.1)$$
$$\text{for all } i = 0 \ldots A, j = 1 \ldots B$$

$$w2_{ij} = random(-0.1,0.1)$$
$$\text{for all } i = 0 \ldots B, j = 1 \ldots C$$

3. Initialize the activations of the thresholding units. These should never change their values.

$$x_0 = 1.0$$
$$h_0 = 1.0$$

4. Choose an input/output pair. Suppose the input vector is x_i and the target output vector is y_i. Assign activation levels to the input units.
5. Propagate the activations from the units in the input layer to the units in the hidden layer, using the activation function of Figure 16:

$$h_j = \frac{1}{1 + e^{-\sum_{i=0}^{A} w1_{ij} x_i}}$$

$$\text{for all } j = 1 \ldots B$$

Note that i ranges from 0 to A. $w1_{0j}$ is the thresholding weight for hidden unit j (its propensity to fire irrespective of its inputs). x_0 is always 1.0.

6. Propagate the activations from the units in the hidden layer to the units in the output layer.

$$o_j = \frac{1}{1 + e^{-\sum_{i=0}^{B} w2_{ij} h_i}}$$
$$\text{for all } j = 1 \ldots C$$

Again, the thresholding weight $w2_{0j}$ for output unit j plays a role in the weighted summation. h_0 is always 1.0.

7. Compute the errors[4] of the units in the output layer, denoted $\delta 2_j$. Errors are based on the network's actual output (o_j) and the target output (y_j).

$$\delta 2_j = o_j(1 - o_j)(y_j - o_j)$$
$$\text{for all } j = 1 \ldots C$$

8. Compute the errors of the units in the hidden layer, denoted $\delta 1_j$.

$$\delta 1_j = h_j(1 - h_j)\sum_{i=1}^{C} \delta 2_i \cdot w2_{ji}$$

$$\text{for all } j = 1 \ldots B$$

9. Adjust the weights between the hidden layer and output layer.[5] The learning rate is denoted η; its function is the same as in perceptron learning. A reasonable value of η is 0.35.

$$\Delta w2_{ij} = \eta \cdot \delta 2_j \cdot h_i$$
$$\text{for all } i = 0 \ldots B, j = 1 \ldots C$$

10. Adjust the weights between the input layer and the hidden layer.

$$\Delta w1_{ij} = \eta \cdot \delta 1_j \cdot x_i$$
$$\text{for all } i = 0 \ldots A, j = 1 \ldots B$$

11. Go to step 4 and repeat. When all of the input/output pairs have been presented to the network, one epoch has been completed. Repeat steps 4 to 10 for as many epochs as desired.

The algorithm generalizes straightforwardly to networks of more than three layers.[6] For each extra hidden layer, insert a forward propagation step between steps 6 and 7; an error computation step between steps 8 and 9; and a weight adjustment step between steps 10 and 11. Error computation for hidden units should use the equation in step 8, but with i ranging over the units in the next layer, not necessarily the output layer.

The speed of learning can be increased by modifying the weight modification steps 9 and 10 to include a momentum term α. The weight update formulas become:

$$\Delta w2_{ij}(t + 1) = \eta \cdot \delta 2_j \cdot h_i + \alpha \Delta w2_{ij}(t)$$

$$\Delta w1_{ij}(t + 1) = \eta \cdot \delta 1_j \cdot x_i + \alpha \Delta w1_{ij}(t)$$

where h_i, x_i, $\delta 1_j$ and $\delta 2_j$ are measured at time $t + 1$. $\Delta w_{ij}(t)$ is the change the weight saw during the previous forward-backward pass. If α is set to 0.9 or so, learning speed is improved.[7]

Recall that the activation function has a sigmoid shape. Since infinite weights would be required for the actual outputs of the network to reach 0.0 and 1.0, binary target outputs (the y_j's of steps 4 and 7) are usually given as 0.1 and 0.9 instead. The sigmoid is required by backpropagation because the derivation of the weight update rule requires that the activation function be both continuous and differentiable.

The derivation of the weight update rule is more complex than the derivation of the fixed-increment update rule for perceptrons, but the idea is much the same. There is an error function that defines a surface over weight space, and the weights are modified in the direction of the gradient of the surface. See [25–27] for details. Interestingly, the error surface for multilayer nets is more complex than

[4] The error formula is related to the derivative of the activation function. The mathematical derivation behind the backpropagation learning algorithm is beyond the scope of this article.

[5] Again, we omit the details of the derivation. The basic idea is that each hidden unit tries to minimize the errors of output units to which it connects.

[6] A network with one hidden layer can compute any function that a network with many hidden layers can compute: with an exponential number of hidden units, one unit could be assigned to every possible input pattern. However, learning is sometimes faster with multiple hidden layers, especially if the input is highly nonlinear (i.e., hard to separate with a series of straight lines).

[7] Empirically, best results have come from letting α be *zero* for the first few training passes, then increasing it to 0.9 for the rest of training. This process first gives the algorithm some time to find a good general direction, then moves it in that direction with some extra speed.

the error surface for perceptrons. One notable difference is the existence of local minima. Recall the bowl-shaped space we used to explain perceptron learning (Figure 10). As we modified weights, we moved in the direction of the bottom of the bowl; eventually, we reached it. A backpropagation network, however, may slide down the error surface into a set of weights that does not solve the problem it is being trained on. If that set of weights is at a local minimum, the network will never reach the optimal set of weights. Thus, we have no analogue of the Perceptron Convergence Theorem for backpropagation networks.

There are several methods of combating the problem of local minima. The momentum factor α, which tends to keep the weight changes moving in the same direction, allows the algorithm to skip over small minima. Simulated annealing, to be discussed later, is also useful. Finally, adjusting the shape of a unit's activation function can have an effect on the network's susceptibility to local minima.

Fortunately, backpropagation networks rarely slip into local minima. It turns out that, especially in larger networks, the high-dimensional weight space provides plenty of degrees of freedom for the algorithm. The lack of a convergence theorem is not a problem in practice. However, this pleasant feature of backpropagation was not discovered until recently, when digital computers became fast enough to support large-scale simulations of neural networks. The backpropagation algorithm was actually derived independently by a number of researchers in the past, but it was discarded as many times because of the potential problems with local minima. In the days before fast digital computers, researchers could only judge their idea by proving theorems about them, and they had no idea that local minima would turn out to be rare in practice. The modern form of backpropagation is often credited to [16, 22, 25, 31].

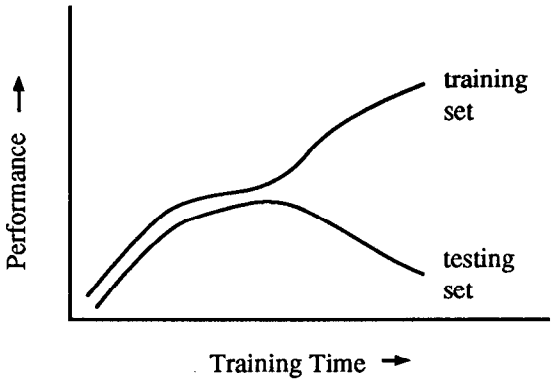


FIGURE 17. A common generalization effect in neural network learning.

Backpropagation networks are not without real problems, however—the most serious being the slow speed of learning. Even simple tasks require extensive training periods. The XOR problem, for example, involves only five units and nine weights, but it can require many passes through the four training cases before the weights converge, especially if the learning parameters are not carefully tuned. Also, simple backpropagation does not scale up very well. The number of training examples required is superlinear in the size of the network.

Since backpropagation is inherently a parallel, distributed algorithm, the idea of improving speed by building special-purpose backpropagation hardware is attractive. However, fast new variations of backpropagation and other learning algorithms appear frequently in the literature, e.g., [7]. By the time an algorithm is transformed into hardware and embedded in a computer system, the algorithm is likely to be obsolete.

Generalization

If all possible inputs and outputs are shown to a backpropagation network, it will (probably, eventually) find a set of weights that maps the inputs onto the outputs. For many AI problems, however, it is impossible to give all possible inputs. Consider face recognition and character recognition. There are an infinite number of orientations and expressions to a face, and an infinite number of fonts and sizes for a character, yet humans learn to classify these objects easily from only a few examples. We would hope that our networks would do the same. And in fact, backpropagation shows promise as a generalization mechanism. If we work in a domain where similar inputs get mapped onto similar outputs, backpropagation will interpolate when given inputs it has never seen before.

There are some pitfalls, however. Figure 17 shows the common generalization effect during a long training period. During the first part of the training, performance on the training set improves as the network adjusts its weight through backpropagation. Performance on the test set (examples that the network is *not* allowed to learn on) also improves, although it is never quite as good as the training set. After a while, network performance reaches a plateau as the weights shift around, looking for a path to further improvement. Ultimately, such a path is found, and performance on the training set improves again. But performance on the test set *gets worse*. Why? The network has begun to memorize the individual input/output pairs rather than settling for weights that generally describe the mapping for all cases. With thousands of real-valued weights at its disposal, back-

propagation is theoretically capable of storing entire training sets; with enough hidden units, the algorithm could learn to assign a hidden unit to every distinct input pattern in the training set. It is a testament to the power of backpropagation that this actually happens in practice.

Of course it is undesirable for backpropagation to have that much power. There are several ways to prevent it from resorting to a table-lookup scheme. One way is to stop training when a plateau has been reached, on the assumption that any other improvement will come through cheating. Another way is to add deliberately small amounts of noise to the training inputs. The noise should be enough to prevent memorization, but it should not be great enough to confuse the classifier. A third way to help generalization is to reduce the number of hidden units in the network, creating a bottleneck between the input and output layers. Confronted with a bottleneck, the network will be forced to come up with compact internal representations of its inputs.

Finally, there is the issue of exceptions. In many domains, there are general rules, but also exceptions to the rules. For example, we can generally make the past tense of an English verb by adding "-ed" to it, but this is not true of verbs like "sing," "think," and "eat." When we show a network many present/past tense pairs, we would like it to generalize in spite of the exceptions—but not to generalize so far that the exceptions are lost. Backpropagation performs fairly well in this regard, as do simple perceptrons, as reported in [26].

Boltzmann Machines

A Boltzmann machine is a variation on the idea of a Hopfield network. Recall that pairs of units in a Hopfield net are connected by symmetric weights. Units update their states asynchronously by looking at their local connections to other units.

In addition to serving as content-addressable memories, Hopfield networks can solve a wide variety of constraint satisfaction problems. Each unit can be viewed as a hypothesis. Mutually supporting hypotheses are connected with positive weights, and incompatible hypotheses are connected with negative weights.

A major limitation of Hopfield networks is that they settle into local minima. In constraint satisfaction tasks we need to find the globally optimal state of the network. This state corresponds to an interpretation that satisfies as many interacting constraints as possible. Unfortunately, Hopfield networks cannot find global solutions because they settle into stable states via a completely distributed algorithm. If a network reaches a stable state like state A in Figure 4, that means no single unit is willing to change its state in order to move uphill; thus the network will never reach globally optimal state B. If several units decided to change state simultaneously, the network might be able to scale the hill and slip into state B. We need a way to push networks into globally optimal states while maintaining our distributed approach.

Boltzmann machines solve this problem by employing a search technique called *simulated annealing* [15]. Space limitations preclude a full discussion of Boltzmann machines; for details, see [11]. Briefly, units in a Boltzmann machine update their individual binary states using stochastic rather than deterministic rules. At first, units switch on and off randomly, but as the network "cools down," they approximate a Hopfield network. If the cooling procedure is slow enough, a Boltzmann machine is guaranteed to avoid local minima. As in backpropagation networks, the weights of a Boltzmann machine are usually acquired via a learning algorithm.

Unsupervised Learning

Some networks, e.g. [3], do not receive target output values from a teacher, but instead only receive a real-valued signal indicating punishment or reward. These networks adjust their behavior to avoid future punishment.

What if a neural network is given *no* feedback for its outputs, not even a reinforcement signal? Can the network learn anything useful? The unintuitive answer is: yes. This form of learning is called *unsupervised learning* because no teacher is required. Given a set of input data, the network is allowed to play with it to try to discover regularities and relationships between the different parts of the input.

Learning is often made possible through some notion of which features in the input set are important. But often we do not know in advance which features are important, and asking a learning system to deal with raw input data can be computationally expensive. Unsupervised learning can be used as a "feature discovery" module that precedes supervised learning.

Consider the data in Figure 18. The group of 10 animals, each described by its own set of features, breaks down naturally into three groups: mammals, reptiles, and birds. We would like to build a network that can *learn* which group a particular animal belongs to, and to generalize so that it can identify animals it has not yet seen. We can easily accomplish this with a six-input, three-output backpropagation network. We simply present the network with an input, observe its output, and update its weights based on the errors it makes. Since without a teacher, however, the error cannot be computed, we must seek other methods.

Our first problem is to ensure that only *one* of the three output units becomes active for any given input. One solution to this problem is to let the network settle, find the output unit with the highest level of activation, set that unit to 1, and set all other output units to 0. In other words, the output unit with the highest activation is the only one we consider to be active. A more

neural-like solution is to have the output units fight among themselves for control of an input vector. The scheme is shown in Figure 19. The input units are directly connected to the output units, as in the perceptron, but the output units are also connected to each other, via prewired negative, or inhibitory, connections. The output unit with the most activation along its input lines initially will most strongly dampen its competitors. As a result, the competitors will become weaker, losing their power of inhibition over the stronger output unit. The stronger unit then becomes even stronger, and its inhibiting effect on the other output units becomes overwhelming. Soon the other output units are all completely inactive. This type of mutual inhibition is called *winner-take-all* behavior. One popular unsupervised learning scheme based on this behavior is known as *competitive learning*.

In competitive learning, output units fight for control over portions of the input space. A simple competitive learning algorithm is the following:

1. Present an input vector.
2. Calculate the initial activation for each output unit.
3. Let the output units fight until only one is active.
4. Increase the weights on connections between the active output unit and *active* input units. This makes it more likely that the output unit will be active next time the pattern is repeated.

A problem with this algorithm is that one output unit may learn to be active all the time—it may claim all of the space of inputs for itself. For example, if all the weights on a unit's input lines are large, it will tend to bully the other output units into submission. Learning will only further increase those weights.

The solution, originally due to Rosenblatt (and described in [27]), is to ration the weights. The sum of the weights on a unit's input lines is limited to 1. Increasing the weight of one connection requires that we decrease the weight of some other connection. Here is the learning algorithm:

	has-hair?	has-scales?	has-feathers?	flies?	lives in water?	lays eggs?
Dog	1	0	0	0	0	0
Cat	1	0	0	0	0	0
Bat	1	0	0	1	0	0
Whale	1	0	0	0	1	0
Canary	0	0	1	1	0	1
Robin	0	0	1	1	0	1
Ostrich	0	0	1	1	0	1
Snake	0	1	0	0	0	1
Lizard	0	1	0	0	0	1
Alligator	0	1	0	0	1	1

FIGURE 18. Data for unsupervised learning.

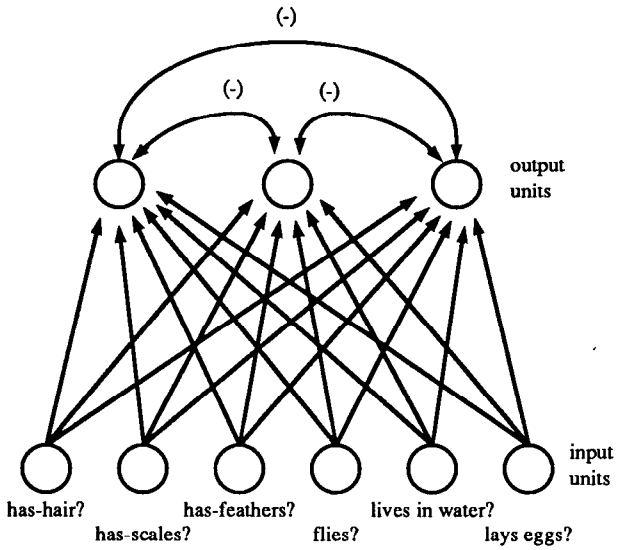


FIGURE 19. A competitive learning network. Input units are connected directly to output units. Through the use of inhibitory connections, output units fight for control of input.

Algorithm: Competitive Learning

Given: A network consisting of n binary-valued input units directly connected to any number of output units.

Produce: A set of weights such that the output unit become active according to some natural division of the inputs.

1. Present an input vector, denoted $(x_1, x_2 \ldots x_n)$.
2. Calculate the initial activation for each output unit by computing a weighted sum of its inputs.[8]
3. Let the output units fight until only one is active.[9]
4. Adjust the weights on the input

lines that lead to the single active output unit:

$$\Delta w_j = \eta \frac{x_j}{m} - \eta w_j$$
$$\text{for all } j = 1 \ldots n$$

where w_j is the weight on the connection from input unit j to the active output unit, x_j is the value of the jth input bit, m is the number of input units that are active in the input vector that was chosen in step (1), and η is the learning rate (some small constant). It can be shown that if the weights on the connections feeding into an output unit total 1 before the weight change, they will still total 1 afterwards.

5. Repeat steps 1-4 for all input patterns, for many epochs.

The weight update rule in step 4 makes the output unit more prone to fire when it sees the same input again. If the same input is presented over and over, the output unit will eventually adjust its weights for maximum activation on that input. Because input vectors arrive in a mixed fashion, however, output units never settle on a perfect set of weights. The hope is that each will find a natural group of input vectors and gravitate toward it, that is, toward high activations when presented with those inputs. The algorithm halts when the weight changes become very small.

The competitive learning algorithm works well in many cases, but it has some problems. Sometimes, one output unit will always win, despite the existence of more than one cluster of input vectors. If two clusters are close together, one output unit may learn weights that give it a high level of activation when presented with an input from *either* cluster. In other words, it may oscillate between the two clusters. Normally, another output unit will win occasionally, and move to claim one of the two clusters. However, if the other output units are completely unexcitable by the input vectors, they may never win the competition. One solution, called *leaky learning*, is to change the weights belonging to relatively inactive output units as well as the most active one. The weight update rule for losing output units is the same as in the algorithm above, except that they move their weights with a much smaller η (learning rate). An alternative solution is to adjust the sensitivity of an output unit through the use of a bias, or adjustable threshold. Recall that this bias mechanism was used in perceptrons, and corresponded to the propensity of a unit to fire irrespective of its inputs. Output units that seldom win in the competitive learning process can be given larger biases. In effect, they are given control over a larger portion of the input space. In this way, units that consistently lose are eventually given a chance to win and adjust their weights in the direction of a particular cluster.

[8]There is no reason to pass the weighted sum through a sigmoid function, as we did with backpropagation, because we only calculate activation levels for the purpose of singling out the most highly activated output unit.

[9]As mentioned earlier, any method for determining the most highly activated output unit is sufficient. Simulators written in a serial programming language may dispense with the neural circuitry and simply compare activations levels to find the maximum.

Applications of Neural Networks

The study of neural networks has yielded a number of techniques that have been used to approach difficult problems with some success. For example, Figure 20 shows how a backpropagation network can be trained to discriminate among different vowel sounds, given a pair of frequencies taken from a speech waveform. A good deal of connectionist research is also directed toward the problem of machine vision. Neural networks provide a framework for integrating the numerous constraint sources necessary for vision, in a highly parallel fashion [2]. Connectionist systems have been applied in many other areas, including speech generation [28], combinatorial problems [13], game playing [29], signal processing [10], image compression [5], and road following [23].

Since all of these systems rely heavily on automatic learning, we can think of them as exercises in "extensional programming" [5]. There exists some complex relationship between input and output, and we program that relationship into the computer by showing its examples from the real world. Contrast this with traditional, "intensional programming," in which we write rules or specialized algorithms without reference to any particular examples. In the former case, we hope that the network generalizes to handle new cases correctly; in the latter case, we hope that the algorithm is general enough to handle whatever cases it receives. Extensional programming is a powerful technique because it drastically cuts down on knowledge acquisition time, a major bottleneck in the construction of AI systems. However, current learning methods are not adequate for the extensional programming of very complex tasks, such as the translation of English sentences into Japanese.

Connectionist AI and Symbolic AI

The connectionist approach to AI is quite different from the traditional symbolic approach. Both approaches are joined at the problem, as both try to address difficult issues in search, knowledge representation, and learning. Let us list some of the methods they employ:

Connectionist

- Search—Parallel relaxation.
- Knowledge Representation—Large number of real-valued connection strengths (Structures often stored as distributed patterns of activation).
- Learning—Backpropagation, Boltzmann machines, reinforcement learning, unsupervised learning.

Symbolic

- Search—State space traversal.
- Knowledge Representation—Predicate logic, semantic networks, frames, scripts.
- Learning—Macro-operators, *version spaces*, explanation-based learning, discovery.

The approaches have different strengths and weaknesses. One major allure of connectionist systems is that they employ knowledge representations that seem to be more *learnable* than their symbolic counterparts. Nearly all connectionist systems have a strong learning component. However, neural network learning algorithms usually involve a large number of training examples and long training periods, compared to their symbolic cousins. Also, after a network has learned to perform a difficult task, its knowledge is usually quite opaque—an impenetrable mass of connection weights. Getting the network to explain its reasoning, then, is difficult. Of course, this may not be a bad thing. Humans, for example, appear to have little access to the procedures they use for many tasks like speech recognition and vision. It is no accident that the most promising uses for neural networks are in these areas of low-level perception.

It is difficult to see how connectionist systems will tackle difficult problems that symbolic, state-space search addresses (e.g., chess, theorem-proving, and planning). Parallel relaxation search, however, does have some advantages over symbolic search. First of all, it maps naturally onto highly parallel hardware. When such hardware becomes widely available, parallel relaxation methods will be extremely efficient. More importantly, parallel relaxation search may prove very efficient because it can make use of states that have no analogues in symbolic search. If we freeze a network while it is still settling, we may not be able to make sense out of the pattern of activity, but eventually, a consistent solution state falls out of the relaxation process. In contrast, a symbolic system can only expand new search nodes that correspond to valid, possible states of the world.

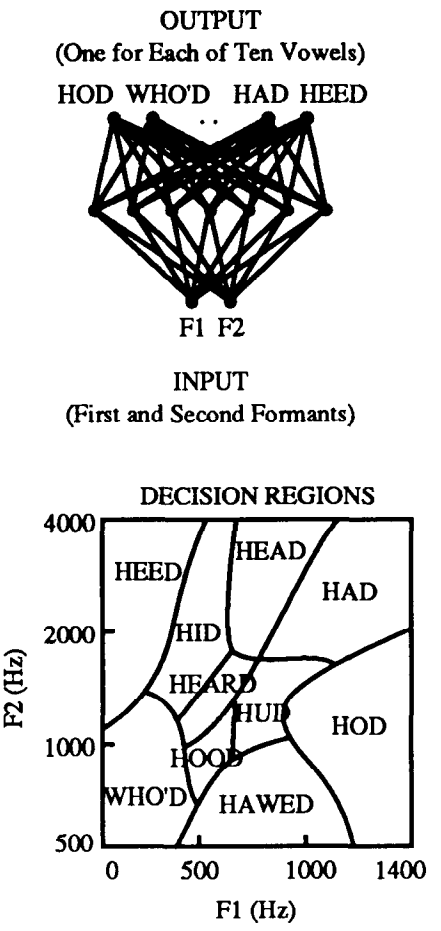


FIGURE 20. A network that learns to distinguish vowel sounds (adapted from [17]).

A good deal of connectionist research concerns itself with modeling human mental processes. Neural networks seem to display many psychologically and biologically plausible features such as content-addressable memory, fault tolerance, distributed representations, automatic generalization. Can we integrate these desirable properties into symbolic AI systems? Certainly, high-level theories of cognition can incorporate such features as new psychological primitives. Practically speaking, we may want to use connectionist architectures for low-level tasks such as vision, speech recognition, and memory, feeding results from these modules into symbolic AI programs. Another idea is to take a symbolic notion, and implement it in a connectionist framework. A connectionist production system is described in [30] and a connectionist semantic network is described in [6]. Ultimately, connectionists would like to see symbolic structures emerge naturally from complex interactions among simple units, in the same way that wetness emerges from the combination of hydrogen and oxygen, although it is an intrinsic property of neither.

Most of the promising advantages of connectionist systems described in this article are just that: *promising*. A great deal of work remains to be done to turn these

promises into results. Only time will tell how influential connectionist models will be in the evolution of AI research. In any case, connectionists can at least point to the brain's existence as proof that neural networks, in some form, are capable of exhibiting intelligent behavior.

Acknowledgments.
I would like to thank Yolanda Gil, Dave Touretzky, and Marco Zagha for their useful comments and suggestions.

References

1. Ashby, W. R. *Design for a Brain.* Wiley, New York. 1952.
2. Ballard, D. H. Parameter nets. *Artif. Intell. 22*, 3 (1984), 235–267.
3. Barto, A. G. Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology 4*, 4 (1985).
4. Block, H. D. The perceptron: A model for brain functioning. *Rev. Mod. Phy. 34*, 1 (1962), 123–135.
5. Cottrell, G. W., Munro P., and Zipser D. Learning internal representations from gray-scale images: An example of extensional programming. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (1987), pp. 461–473.
6. Derthick, M. *Mudane Reasoning by Parallel Constraint Satisfaction.* Ph.D dissertation, Carnegie Mellon University, Pittsburgh, PA., 1988.
7. Fahlman, S. E. Faster-learning variations on back-propagation: An empirical study. In *Proceedings of the 1988 Connectionist Models Summer School.* Morgan Kaufmann Publishers, San Mateo, Calif. 1988, pp. 38–51.
8. Fahlman, S. E. and Hinton, G. E. Connectionist architectures for artificial intelligence. *IEEE Comput. 20*, 1 (1987), 100–109.
9. Feldman, J. A. and Ballard, D. H. Connectionist models and their properties. *Cogn. Sci. 6*, 3 (1985), 205–254.
10. Gorman, R. and Sejnowski, T. J. Analysis of hidden units in a layered network to classify sonar targets. *Neural Networks 1*, 1 (1988), 75–89.
11. Hinton, G. E. and Sejnowski, T. J. 1986. Learning and relearning in Boltzmann Machines. In *Parallel Distributed Processing*, D. E. Rumelhart, J. L. McClelland, and the PDP Research Group Eds., MIT Press, Cambridge, Mass., pp. 282–317.
12. Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Acadamy of Sciences USA 79*, 8 (1982), pp. 2554–2558.
13. Hopfield, J. J. and Tank, D. W. 'Neural' computation of decisions in optimization problems. *Biol. Cybern. 52*, 3 (1985), 141–152.
14. Jordan, M. I. Supervised learning and systems with excess degrees of freedom. In *Proceedings of the 1988 Connectionist Models Summer School.* Morgan Kaufmann Publishers, San Mateo, Calif., 1988, pp. 62–75.
15. Kirkpatrick, S., Gelatt, Jr., C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science 220*, 4598 (1983).
16. LeCun, Y. Une procedure d'apprentissage pour reseau a seauil assymetrique (a learning procedure for asymmetric threshold networks). In *Proceedings of Cognitiva 85*, (Paris, 1985), pp. 599–604.
17. Lippmann, R. P. Review of research on neural nets for speech. *Neural Comput. 1*, 1 (1989).
18. McCulloch, W. S. and Pitts, W. A logical calculus of the ideas immanent in neural nets. *Bulletin of Math. Biophys. 5* (1943), 115–137.
19. Minsky, M. *Neural Nets and the Brain-Model Problem.* Ph.D dissertation, Princeton University, Princeton, New Jersey, 1954.
20. Minsky, M. and Papert, S. *Perceptrons.* MIT Press, Cambridge, Mass 1969. Expanded also published in edition, MIT Press, 1988.
21. Minsky, M. and Selfridge, O. G. Learning in neural nets. In *Proceedings of the Fourth London Symposium on Information Theory* (August 29 to September 2, London). Academic Press, New York 1961.
22. Parker, D. B. *Learning Logic.* Tech. Rep. TR-47, MIT Center for Computational Research in Economics and Management Science, 1985.
23. Pomerleau, D. ALVINN: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems 1*, D. Touretzky, Ed. Morgan Kaufmann, San Mateo, Calif., pp. 305–313.
24. Rosenblatt, F. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan Books, Washington, D.C. 1962.
25. Rumelhart, D. E., Hinton, G. E. and Williams, R. J. Learning internal representations by error propagation. In *Parallel Distributed Processing*, D. E. Rumelhart, J. L. McClelland, and the PDP Research Group Eds. MIT Press, Cambridge, Mass., pp. 318–362.
26. Rumelhart, D. E. and McClelland, J. L. 1986. One learning the past tenses of English verbs. In *Parallel Distributed Processing*, D. E. Rumelhart, J. L. McClelland, and the PDP Research Group Eds. MIT Press, Cambridge, Mass. pp. 216–271.
27. Rumelhart, D. E. and Zipser, D. Feature discovery by competitive learning. In *Parallel Distributed Processing*, D. E. Rumelhart, J. L. McClelland, and the PDP Research Group Eds. MIT Press, Cambridge, Mass., pp. 151–193.
28. Sejnowski, T. J. and Rosenberg, C. R. Parallel networks that learn to pronounce English text. *Complex Systems 1*, 1 (1987), 145–168.
29. Tesauro, G. and Sejnowski, T. J. A parallel network that learns to play backgammon. *Artif. Intell. 39*, (1989).
30. Touretzky, D. and Hinton, G. E. A distributed connectionist production system. *Cog. Sci. 12*, 3 (1988), 423–466.
31. Werbos, P. J. *Beyond Regression:* New Tools for Prediction and Analysis in the Behavioral Sciences. Ph.D. dissertation, Harvard University, Cambridge, Mass., 1974.

CR Categories and Subject Descriptors: A.1 **[General Literature]:** Introductory and Survey; I.2.O **[Artificial Intelligence]:** General

General Terms: Algorithms

Additional Key Words and Phrases: Backpropagation, Boltzmann machines, connectionism, Hopfield networks, learning, neural networks, perceptrons.

About the Author:

KEVIN KNIGHT is a Ph.D. candidate in computer science at Carnegie Mellon University. He is also a regular consultant to the Artificial Intelligence Laboratory at MCC in Austin, Texas. His research interests include natural language processing, unification, and neural networks.

Author's Present Address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.