CHAPTER 10 - NEURAL NETWORKS

It has been known for more than a century that the nervous system is made of individual nerve cells or *neurons*. There are an immense number of neurons - on the order of 10^{11} - in the human nervous system. Neurons do not exist in isolation from one another, but rather are linked to form a network in which each neuron has multiple connections to other neurons. While all neurons are structurally similar, an individual neuron has one of three principal tasks within this network:

- to collect information from sensory cells
- to relay information from one neuron to another
- to deliver information (*e.g.*, stimulate a muscle cell).

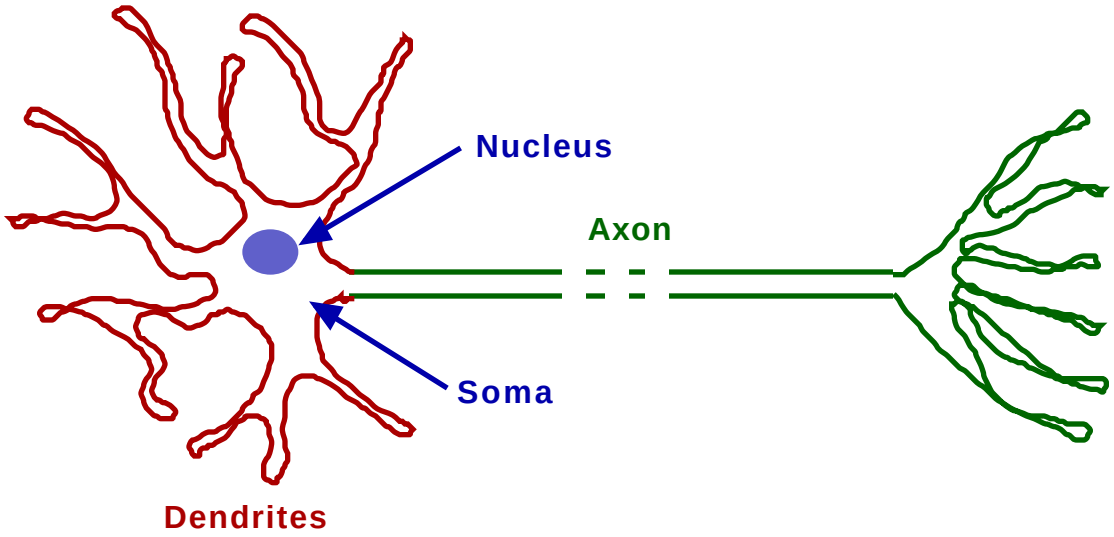


Fig. 10.1 Schematic drawing of a neuron. The soma is typically 10 to 80 μm in diameter, while the axon and dendrites have much smaller diameters of a few μm (the largest axon being about 15 μm in diameter). The length of an axon can range from a few mm in the brain, up to a meter for large motor neurons. Information travels from the dendrites through the soma, to the axon.

In Sec. 10.1, we provide a short overview of the physical characteristics of a single neuron. Although neurons are the elementary building blocks of the nervous system, they work collectively in networks, which are discussed in Sec. 10.2. Finally, a computer representation of a neural network is presented in Sec. 10.3.

10.1 A single neuron

As illustrated in Fig. 10.1, neurons are string-like in appearance, and there is a preferred direction in which they transmit information electrically. At one end of the cell is a region called the *soma* which contains the cell's nucleus and from which radiates a branched structure containing *dendrites*. The dendrites receive information from other neurons or sensory cells, a function enhanced by the branched geometry of the dendrites that allows them to have up to 10^5 inputs from other cells. The information is directed towards the soma, and from there proceeds away from the nuclear region along a single *axon*. The far end of the axon is also branched, and allows a signal to be passed on to many other cells.

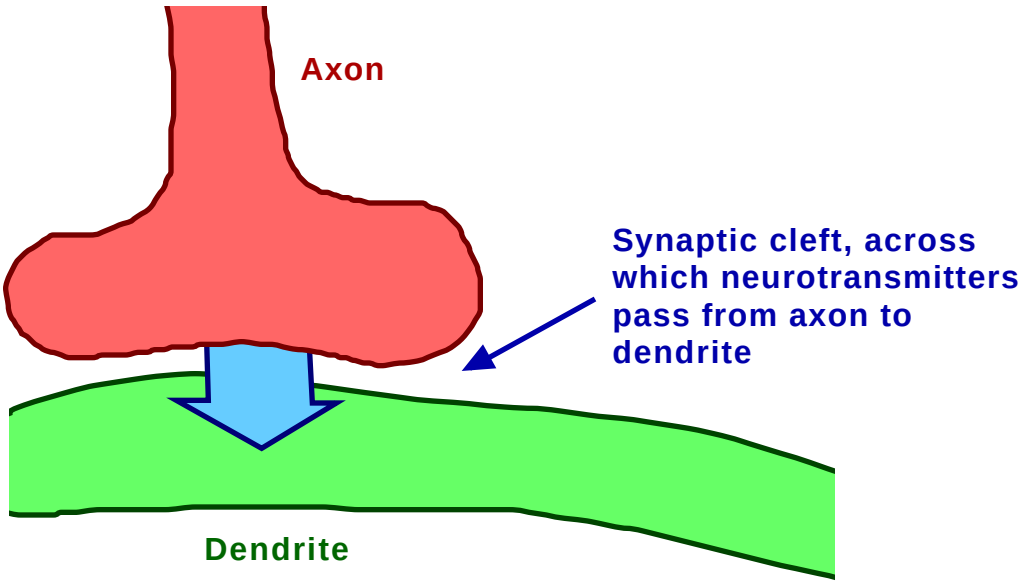


Fig. 10.2 An electrical pulse reaches the end of an axon from one neuron, where it causes the release of neurotransmitters. These chemicals traverse the synaptic gap to a dendrite of a second neuron, providing a stimulus for the neuron to generate a pulse of its own.

Within a single nerve call, information travels in the form of an electrical pulse along a continuous plasma membrane that forms the cell boundary. The speed of the pulse ranges from 0.5 to 2 m/s within the brain, but reaches 100 m/s along the motor neurons connecting the brain to muscle cells, as it must if the muscles are to have reasonable response times. The branched structure at the far end of the axon terminates in regions called *synapses*, where the axon is in close proximity with other cells, but is separated by a distance of 0.1 to 0.2 μm at the *synaptic cleft*. When the electrical pulse reaches the synapse, it causes the release of chemical *neurotransmitters* that travel across the synaptic cleft to a neighboring cell. The geometry of a synapse is illustrated in Fig. 10.2

Although the neurotransmitters from one neuron provide a stimulus for the dendrite of a second neuron, the stimulus may not be strong enough to trigger an electrical pulse to the axon of the second neuron. The second neuron contains many dendrites, whose collective action determines whether the second neuron emerges from its *rest state* of no electrical activity and becomes electrically *active*. The details of how, and under what conditions, a neuron creates a pulse are fairly complex and take us away from our principal aim of finding simple ways to describe the flow of information in a network. The reader interested in the biochemistry of neurons is directed to the references at the end of Sec. 10.5 for further reading.

10.2 A network of neurons

The neuron shows a preferred direction for the flow of information. We will represent a neuron diagramatically in Fig. 10.3, where the soma and associated dendrites are indicated by a circle, and a synapse at the end of an axon is represented by a triangle. The information flow is given by the red arrow. Although Fig. 10.3 shows only one synapse, in the following diagrams we will use multiple branches to represent the many synapses of a more conventional neuron.

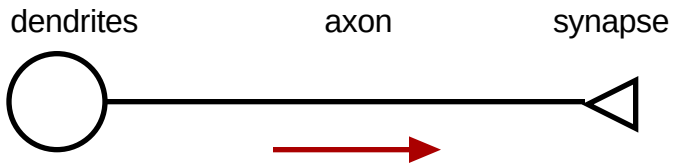


Fig. 10.3 Schematic representation of a neuron. The dendrites and soma are indicated by a circle, while a synapse at the end of the axon is shown as a triangle. The red arrow displays the direction of information flow.

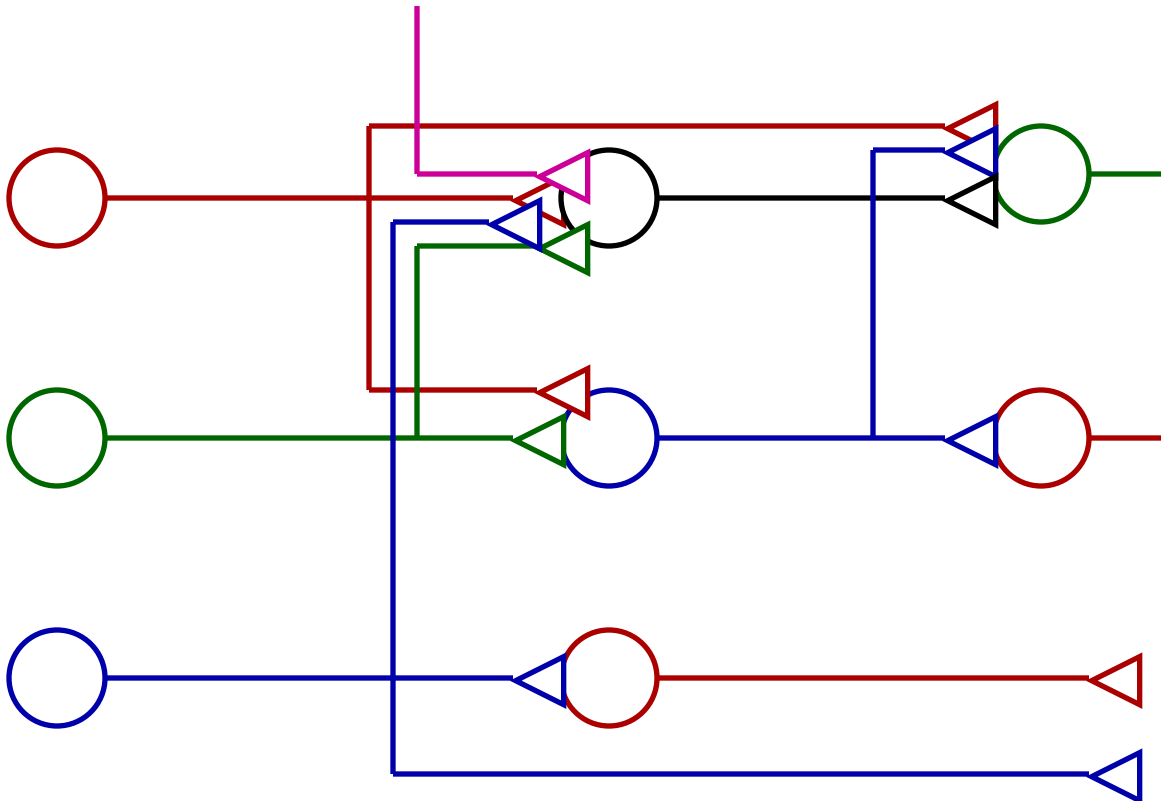


Fig. 10.4 Circuit diagram of a network with multiple connections. Dendrites are represented by circles, axons by straight lines, and synapses by triangles. Each neuron is color-coded. The information flow is from left to right.

The connectivity of a network of neurons is much more complex. In Fig. 10.4, we show a simple "circuit diagram" that might be present with several neurons, each of which has a number of synapses. As stressed in Sec. 10.1, a real neuron may have *several orders of magnitude* more synapses than are present in the figure. The figure illustrates the intricacy of the connectivity of the network: potentially, many neurons can affect many other neurons. While it might be tempting to suppose that one neuron (say the black one in the top center of the figure) may "fire" more frequently, since it has many more synapses than the other neurons, this is not necessarily the case. Some neurons provide an inhibitory signal at a synapse, that may affect the excitatory activity of the other neurons. Further, some synapses may have more influence than others: a signal travelling along a dendrite may decay with distance, so that signals from remote dendrites may have little effect on the activity of the neuron.

The summed signal from the synapses is obviously a complex function of the synapses, their geometry and chemistry. But even if the synapses provide a clear stimulus to the neuron through its dendrites, a threshold signal may be required before the neuron will fire. That is, there may be a threshold that the integrated signal from the synapses must overcome before the neuron becomes active. This is a mathematically rich problem indeed, and we turn now to a simple representation of neural networks that can be programmed on a computer.

10.3 Computer representation of networks

Let us introduce some symbols to both quantify, and perhaps exaggerate, our knowledge of the network. At time t_1, the neurons on the left side of Fig. 10.4 may be active or inactive. To represent their *activity*, we assign a quantity λ_i to each neuron i, where λ_i need have only two values. While the numbers 0 and 1 are appropriate values for λ_i, in fact a more convenient choice that allows simpler functional forms for network algorithm is

$$\lambda_i = \pm 1. \qquad (10.1)$$

After a short time, the signals have travelled along the axons and arrived at another set of neurons, whose state at time t_2 we denote by λ_j. The synapses may provide an excitatory signal to some neurons, but not to others, depending on the values of λ_i and their coupling strength to each neuron j. We could write the influence (but not the outcome, we'll get to that in a moment!) of the set of neurons i on the set j by as the linear combination

$$h_j = \sum_i w_{ij} \lambda_i, \qquad (10.2)$$

where h_j is the called the *synaptic potential* of neuron j, and w_{ij} are the coupling strengths or *connection weights* (frequently shortened to just *weights*) between neurons i and j. Note that both h_j and λ_i are evaluated at time t_1.

Whether neuron j will actually become active at t_2 depends upon the magnitude of h_j at time t_1. One model for the criterion of whether neuron j fires, given that it has been raised to a synaptic potential h_j, is to compare h_j against a fixed threshold ϕ_j: if h_j exceeds ϕ_j then neuron j will fire. The dependence of the values of λ_j at time step t_2 on

the synaptic potential h_j at time step t_1 can then be summarized by

$$\lambda_j(t_2) = 2 \bullet \theta(h_j(t_1) - \phi_j) - 1, \qquad (10.3)$$

where $\theta(x)$ is the step-function potential

$$\theta(x) = 0 \text{ if } x < 0 \quad \text{and} \quad \theta(x) = 1 \text{ if } x > 0. \qquad (10.4)$$

The form of Eq. (10.3) is such that $\lambda_j = \pm 1$. The θ-function is not the only model function used to obtain $\lambda_j(t_2)$ given $h_j(t_1)$; one could use probabilistic functions as an alternative. The mapping of the network onto the model systems is demonstrated in Fig. 10.5.

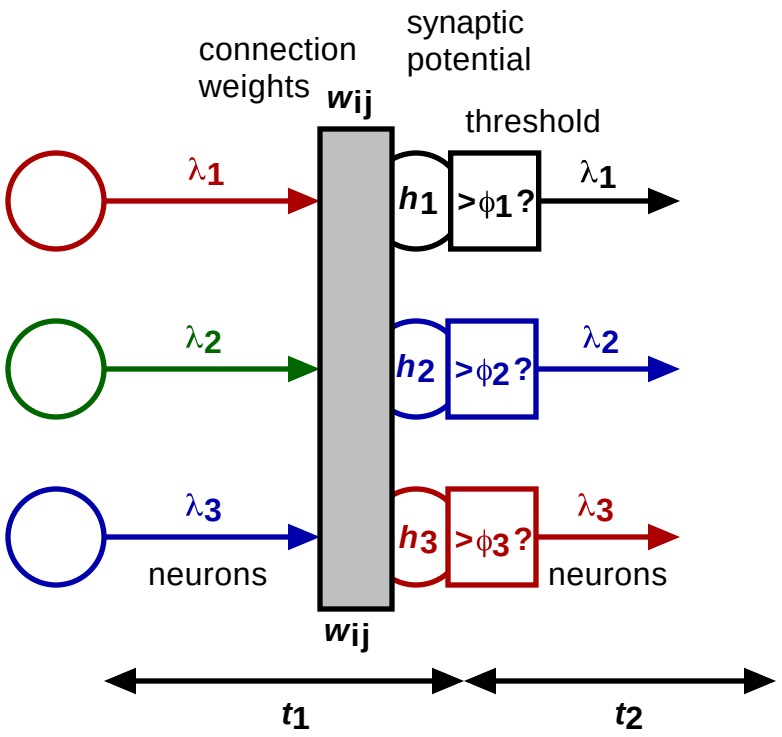


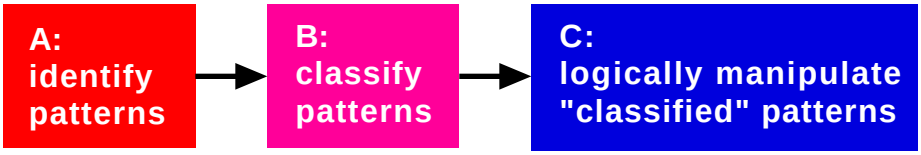
Fig. 10.5 Mapping of the network in Fig. 10.4 onto the model network represented by Eqs. (10.1) to (10.3).

Eqs. (10.2) and (10.3) give us a mathematical means of describing and propagating a neural network. There are many assumptions in this simple model - linear dependence for coupling, no backward flow of information - that may not be valid for real networks. Indeed, there are logical operations that this model network is incapable of performing if there is no intermediate layer of neurons between the input and output layers. Nevertheless, it provides an appealing starting point for attacking the problem of *pattern association*, which is the project of Sec. 10.6.

10.4 The task of pattern association

Pattern association is an important and growing area of computing. Applications include automated letter-sorting at the post office, optical character reading for digital conversion of documents, landscape recognition for aviation *etc*. Pattern association is also an important function of the brain: we frequently reconstruct past events having only limited clues to guide us. Our brain selects information from its immense memory by associating fragmentary clues with different aspects of the information that we are trying to recall. In this section, we describe an algorithm for recognizing which pattern of a specific known set is closest to an "unknown" pattern that may have wrong or incomplete data. In Sec. 10.6, the algorithm is used to compare patterns with the letters X, Y and Z.

This section deals with only one application of neural networks, namely the associative recall of information. In Chap. 11, we look at the classification of patterns (*e.g.*, recognizing which alphanumeric patterns are letters and which are numbers) and the use of networks to execute logical operations. Our strategy in discussing neural networks is:



In terms of the number of numerical steps in a given network algorithm, operation A generally involves the most steps, and operation C the least. However, we choose to present the material in the order A to C because of its intuitive basis in problem-solving. We emphasize that the number of computational steps in a neural network may be much less than what is required to solve a problem using **if** statements.

Pattern recognition involves matching an "unknown" pattern with one of several known patterns. In some sense, the process is like a "best fit" for a parameter: one tries to find which known pattern has the minimum mismatch with the unknown pattern.

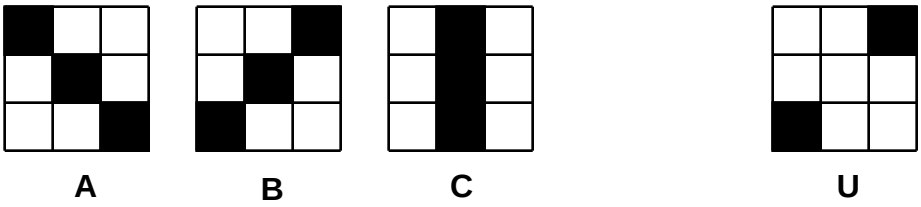


Fig. 10.6 Patterns A - C are "known" patterns: pixelized versions of straight lines, while U is a pattern purportedly based on A - C, but whose identity has been slightly altered. Our task is to find from which pattern U was generated.

To put the question on a more quantitative footing, consider the three known patterns A, B and C in Fig. 10.6. The patterns are pixelized lines, one vertical line and two diagonal lines. The unknown pattern is U, which obviously is not an exact match with any of the patterns A - C. Suppose we are told that U originally looked like one of A - C, but the version in our hands went through a xerox machine too many times and has become distorted. How do we determine which pattern is closest to U?

First, let us change the form of the patterns from a 2-index array to a vector. While this procedure is not necessary mathematically, it makes the notation somewhat less confusing. The result is shown in Fig. 10.7. The convention that we adopt for generating the vector is to read off the pattern from left to right, row-by-row, starting with the top row. This convention is obviously not unique, and other conventions could

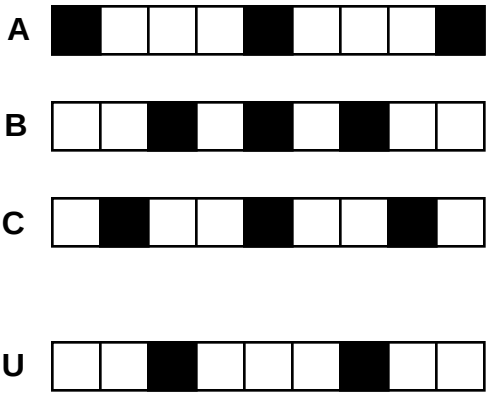


Fig. 10.7 Patterns A, B, C and U displayed as vectors. The convention for generating the vector in this figure is to read off the pattern from left to right, row-by-row, starting with the top row.

just as easily have been employed. Next we assign a +1 or -1 to each pixel according to whether the pixel is black or white. Again, one could just as easily assign 1 or 0 and then do a translation, but the +1 / -1 notation lends itself to compact algorithms. In other words, each pattern is a 9-component vector whose elements have the value ±1. In general, one must be very cautious in using representations that have zeroes for elements. As described in Chap. 11, zeroes may make the functional form of the learning rules cumbersome, or may wipe them out entirely.

We denote the 9-component vector of the known pattern A by λ_i^A (i = 1 ... 9), and similarly with the other known patterns. For the sake of notational clarity, a slightly different symbol will be used for the unknown pattern, namely L_i. The statistic that we use to measure the similarity of the patterns is nothing more than a normalized chi-square, but now is called the Hamming distance H. Just to make sure that there is no ambiguity in which Hamming distance applies to which pattern, we add a subscript, H_A. The Hamming distance from pattern A to the unknown pattern is then

$$H_A = (1/4) \sum_i (\lambda_i^A - L_i)^2, \qquad (10.5)$$

where the normalization factor of 1/4 is added so that each mismatch adds 1 to the Hamming distance. Thus, we have

$$H_A = 5 \qquad H_B = 1 \qquad H_C = 5. \qquad (10.6)$$

In Chap. 8, we introduced a best-fit procedure that minimizes the chi-square statistic; here, the best fit minimizes the Hamming distance.

When we ourselves look at the patterns, and ask which one is U closest to, we try to find ways in which the unknown pattern can be made to look like one of the known patterns. We tend to think that the fewer changes needed to make U look like one of the other patterns, the more closely U resembles that pattern. For example, if we change the central square of U from white to black, then we have pattern B. In comparison, to change U to either of patterns A or C requires 5 changes to the pixels. We conclude that U is closest to pattern B.

Is there a computational equivalent to this process of changing a trial pattern to make it evolve dynamically towards one of the known patterns? In Chap. 8, we saw how to evolve a trial set of variables towards a solution set that minimizes chi-square. Here, neural networks provide the basis for an algorithm that evolves a pattern towards a solution. This approach is different from what might be called the "logical"

determination of a pattern:
•"logical": step-by-step identification; lots of **if** statements and comparisons
•neural network: one or two step application of the network to the pattern; implicit parallel processing.

10.5 Hebb's rule

In Sec. 10.3, we described a simple model for the temporal evolution of a set of neurons with activation λ_i from time t_1 to t_2:

$$\lambda_j(t_2) = 2 \bullet \theta(h_j(t_1) - \phi_j) - 1, \qquad (10.7)$$

where the thresholds are denoted by ϕ_j, and the synaptic potentials h_j are generated by the weights w_{ij}:

$$h_j = \sum_i w_{ij} \lambda_i. \qquad (10.8)$$

While this is a useful propagation formalism, it does not of itself give expressions for w_{ij} and ϕ_j. To find these quantities, we must specify how we expect the network to behave. For example, the network must leave exact matches untouched. That is, if the unknown pattern is identical to a known pattern, say $L_i = \lambda_i^B$, then the network must propagate L_i as λ_i^B without change.

An algorithm that correctly propagates these patterns is called Hebb's rule (after Donald Hebb, a psychologist at McGill, who proposed a set of learning rules in 1949). A generalized version of Hebb's rule is used in Chap. 11 for pattern classification; here, we present a more restricted version appropriate to pattern association. First of all, the thresholds vanish:

$$\phi_j = 0 \text{ for all } j. \qquad (10.9)$$

Since there are no thresholds at the synapses, then Eq. (10.7) becomes

$$\lambda_j(t_2) = \text{sgn}(h_j(t_1)), \qquad (10.10)$$

which obviously yields ±1 for the values of λ_j. In addition, the form for the connection weights is taken to be

$$w_{ij} = N^{-1} \sum_p \lambda_i^p \lambda_j^p, \qquad (10.11)$$

where N is the number of elements in the pattern vector (9 in the example) and the summation is over all of the known patterns λ_i^p.

To show how this algorithm works, consider the case where there is only one known pattern, λ_i. Then Eq. (10.11) is simply

$$w_{ij} = N^{-1} \lambda_i \lambda_j. \qquad (10.12)$$

It is easy to show that this form leaves the known pattern unchanged during propagation:

$$h_j = \sum_i w_{ij} \lambda_i$$

$$= N^{-1} \sum_i \lambda_i \lambda_j \lambda_i$$

$$= N^{-1} N \lambda_j = \lambda_j, \qquad (10.13)$$

where the last line follows from $(\lambda_i)^2 = 1$. Substituting Eq. (10.13) into Eq. (10.10) yields

$$\lambda_j(t_2) = \mathrm{sgn}(h_j(t_1)) = \mathrm{sgn}(\lambda_j(t_1)) = \lambda_j(t_1). \qquad (10.14)$$

Thus, a known solution is propagated without change.

To find the effect of propagating the unknown pattern L_i, we continue to work with the situation in which there is only one unknown pattern. The synaptic potentials are

$$h_j = \sum_i w_{ij} L_i = N^{-1} \sum_i \lambda_i \lambda_j L_i$$

$$= N^{-1} \lambda_j \left[\sum_{i,\,\mathrm{like}} \lambda_i L_i + \sum_{i,\,\mathrm{unlike}} \lambda_i L_i \right]. \qquad (10.15)$$

The sum over i has been broken into 2 parts, one in which the elements λ_i and L_i are

the same ("like" $\lambda_i L_i = 1$) and one in which they are different ("unlike" $\lambda_i L_i = -1$). Suppose that n elements are different. Then

$$h_j = N^{-1} \lambda_j \left[\sum_{i,\ like} (1) + \sum_{i,\ unlike} (-1) \right] = N^{-1} \lambda_j [(N - n) - n]$$

or

$$h_j = (1 - 2n/N) \lambda_j. \qquad (10.16)$$

As long as $n < N/2$ (*i.e.*, as long as half of the elements agree), then

$$\lambda_j(t_2) = \mathrm{sgn}(h_j (t_1)) = \mathrm{sgn}(\lambda_j(t_1)) = \lambda_j(t_1). \qquad (10.17)$$

Thus, Hebb's rule maps the unknown pattern onto the (one) correct pattern in a single step, if there is less than 50% initial disagreement.

One short comment before we go on to apply Hebb's rule to a specific project. The summations for the weights are unrestricted, and $i = j$ is allowed in Eq. (10.11). If $w_{ii} \neq 0$, then the network contains self-coupling, and neuron i may fire at time t_2 in part because it fired at t_1. If one wishes to avoid this attribute, then one can force $w_{ii} = 0$, and change the normalization of Eq. (10.11) from N^{-1} to $(N - 1)^{-1}$.

References

R. M. Berne and M. N. Levy, eds., *Physiology* (Mosby - Year Book, St. Louis, 1993), Sec. II.

L. Fausett, *Fundamentals of Neural Networks* (Prentice-Hall, Englewood Cliffs, NJ, 1994) Chap. 1.

E. R. Krandel and J. H. Schwartz, eds., *Principles of Neural Science* (Elsevier, New York, 1985), Chaps. 1-5.

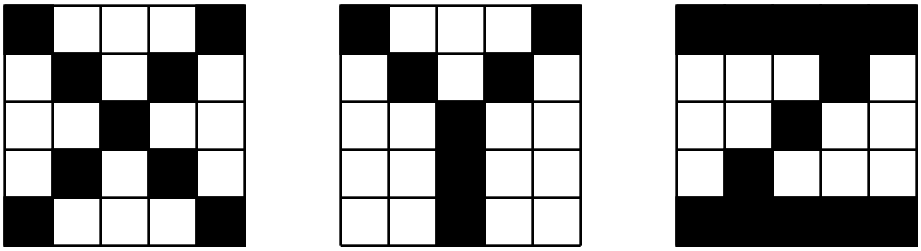
B. Muller and J. Reinhardt, *Neural Networks: an Introduction* (Springer-Verlag, Berlin, 1990), Chaps. 1-4.

10.5 Project 10 - Three pattern choice

As an introductory project in neural networks, we use Hebb's rule to distinguish among three patterns: the capital letters X, Y and Z. The actual computational part of the project is trivial, although some effort should be spent on I/O to make the code user-friendly. The interested student can easily become absorbed in designing graphical interfaces for this project, although we will content ourselves with typing 0's and 1's to enter the pattern to be identified.

Simulation parameters

The patterns of the code are pixelized versions of the capital letters X, Y and Z. To provide a sufficient number of neurons for the network, each letter is represented by a 5x5 array of bits.



Admittedly, the letters are a little grainy, but they will do for a first attempt. Since we are using 25 neurons to describe three patterns, Hebb's rule should be both simple to implement and rapid to execute.

To avoid becoming lost in indices, it may be easiest to describe each pattern by a vector with 25 elements. For example, writing the letter X as a string of 0's and 1's:

1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1

Of course, one does not need to type out this vector as data. In fact, for larger patterns, it may be more efficient to use a few lines of code instead. For example, the letter X in 0 / 1 notation is:

```
// empty the array
for(i=0; i<5; i++) {for(j=0; j<5; j++) a[i][j]=0;}

// add the diagonals
for(i=0; i<5; i++) {a[i][i] =  a[i][4-i] = 1;}
```

It is then straightforward to convert the array to a vector. Finally, shift the values from 0 / 1 notation to -1 / 1 notation by

v[i] = 2*v[i] - 1;

In considering your I/O routines, it is probably easiest for the user of the code to enter 5 lines of numbers, rather than a single vector. Your code could read the entry as a 2-index array (and then convert it to a vector) or as a vector (by placing a **scanf** statement in a loop). Also, it is probably easiest for the user to enter the trial pattern in 0 / 1 convention than in the -1 / 1 convention.

Code

This is a very simple code to write, since the expression for the weights is known *a priori*. Make sure that λ_i and L_i are in the -1 / 1 convention when performing steps 2 and 4, since the learning rule will not work in the 0 / 1 convention.

1. Set up the known values for the neurons λ_i^p (p different vectors, each with N components) for the three unknown patterns ($p = 3$).

2. Use λ_i^p to determine the weights

$$w_{ij} = N^{-1} \sum_p \lambda_i^p \lambda_j^p .$$

3. Read in the unknown pattern L_i (N-component vector after translation) from the screen.

4. Evaluate the synaptic potentials h_i using the weights w_{ij}:

$$h_i = \sum_j w_{ij} L_j .$$

5. Determine the updated value of L_i**'** from

$$L_i' = \mathrm{sgn}(h_i).$$

Analysis

First, enter a few trial configurations through the keyboard to gain some intuition as to how your code identifies patterns with errors. For example, a distorted Z might be

(i) few errors

1 1 1 1 1
0 0 0 1 1
0 0 1 1 0
0 1 0 0 0
1 1 1 0 1

(ii) many errors

1 0 0 0 1
0 0 1 1 1
0 0 1 0 0
0 0 0 0 0
0 1 0 0 0

Evaluate the Hamming distance H_p of your unknown pattern for each of the three letters.

Next, as a numerical test of the network, try the following:

- start with a known pattern
- introduce E errors, beginning with $E = 1$, at random locations
- evaluate the accuracy of each E, by choosing sufficiently many configurations with the same E to generate a good ensemble average
- repeat the above procedure for each pattern

Report

Your report should contain the following elements:

- a statement of the pattern-association problem, and the solution (Hebb's rule) that you have used to solve it
- an outline of your code
- report the analysis of the network's accuracy as a function of the number of errors; are all three patterns equally robust, or can one tolerate more errors?
- a copy of your code.

Demo code

The demonstration code allows you to interactively adjust the trial pattern and see immediately the response of the network. The initial pattern has been set to X, but it can be changed by clicking on the desired neuron in the central configuration. Although the demo code does not display the Hamming distance, you can use it to verify patterns that you use to test your own code.