Νευρο-Ασαφής Υπολογιστική
Neuro-Fuzzy Computing

Διδάσκων –
Δημήτριος Κατσαρός

@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 20η

Supervised Hebbian learning

Linear associator

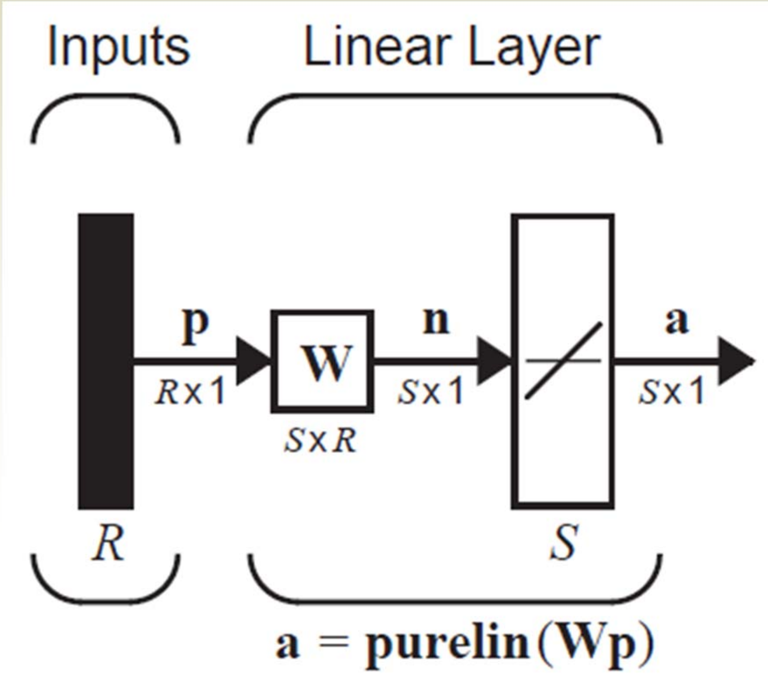
- The Hebb rule was one of the first neural network learning laws. It was proposed by Donald Hebb in 1949 as a possible mechanism for synaptic modification in the brain and since then has been used to train artificial neural networks

- The most famous idea contained in The Organization of Behavior was the postulate that came to be known as Hebbian learning:

"*When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*"

Linear associator

- Hebb's learning law can be used in combination with a variety of neural network architectures. We will use a very simple architecture for our initial presentation of Hebbian learning. In this way we can concentrate on the learning law rather than the architecture. The network we will use is the *linear associator*, which is shown below



Linear associator

- The output vector **a** is determined from the input vector **p** according to:

$$\mathbf{a} = \mathbf{W}\mathbf{p}$$

or

$$a_i = \sum_{j=1}^{R} w_{ij} p_j$$

- The linear associator is an example of a type of neural network called an *associative memory*. The task of an associative memory is to learn pairs of prototype input/output vectors:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

The Hebb rule

- How can we interpret Hebb's postulate mathematically, so that we can use it to train the weight matrix of the linear associator?
- First, let's rephrase the postulate: If two neurons on either side of a synapse are activated simultaneously, the strength of the synapse will increase
- Notice from previous equation that the connection (synapse) between input p_j and output a_i is the weight w_{ij}
- Therefore Hebb's postulate would imply that if a positive p_j produces a positive a_i then w_{ij} should increase
- This suggests that one mathematical interpretation of the postulate could

The Hebb rule

- This suggests that one mathematical interpretation of the postulate could be:

$$w_{ij}^{new} = w_{ij}^{old} + \alpha f_i(a_{iq}) g_j(p_{jq})$$

where p_{jq} is the j-th element of the input vector $\mathbf{p_q}$; a_{iq} is i-th the element of the network output when the q-th input vector is presented to the network; and α is a positive constant, called the learning rate

- This equation says that the change in the weight is proportional to a product of functions of the activities on either side of the synapse
- We will use the simplified version:

$$w_{ij}^{new} = w_{ij}^{old} + \alpha a_{iq} p_{jq}$$

The Hebb rule

- Note that this expression actually extends Hebb's postulate beyond its strict interpretation. The change in the weight is proportional to a product of the activity on either side of the synapse
- Therefore, not only do we increase the weight when both p_j and a_i are positive, but we also increase the weight when they are both negative. In addition, this implementation of the Hebb rule will decrease the weight whenever p_j and a_i have opposite sign

The Hebb rule

- The Hebb rule defined in the previous equation is an *unsupervised* learning rule. It does not require any information concerning the target output
- In this lecture we are interested in using the Hebb rule for supervised learning, in which the target output is known for each input vector
- For the supervised Hebb rule we substitute the target output for the actual output. In this way, we are telling the algorithm what the network should do, rather than what it is currently doing
- The resulting equation is

$$w_{ij}^{new} = w_{ij}^{old} + t_{iq} p_{jq}$$

The Hebb rule

- We can write the previous equation in vector notation:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q \mathbf{p}_q^T$$

- If we assume that the weight matrix is initialized to zero and then each of the input/output pairs are applied once, we can write:

$$\mathbf{W} = \mathbf{t}_1 \mathbf{p}_1^T + \mathbf{t}_2 \mathbf{p}_2^T + \cdots + \mathbf{t}_Q \mathbf{p}_Q^T = \sum_{q=1}^{Q} \mathbf{t}_q \mathbf{p}_q^T$$

- This can be represented in matrix form:

$$\mathbf{W} = [\mathbf{t}_1 \; \mathbf{t}_2 \; \ldots \; \mathbf{t}_Q] \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix}$$

where $\mathbf{T}=[\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_Q]$ $\mathbf{P}=[\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_Q]$

Performance analysis

- Let's analyze Hebbian learning for the linear associator. First consider the case where the $\mathbf{p_q}$ vectors are orthonormal (orthogonal and unit length). If $\mathbf{p_k}$ is input to the network, then the network output can be computed:

$$\mathbf{a} = \mathbf{W}\mathbf{p_k} = \left(\sum_{q=1}^{Q} \mathbf{t}_q \mathbf{p}_q^T\right)\mathbf{p_k} = \sum_{q=1}^{Q} \mathbf{t_q}(\mathbf{p_q^T}\mathbf{p_k})$$

- Since the $\mathbf{p_q}$ are orthogonal:

$$(\mathbf{p^T}_\mathbf{q}\mathbf{p_k}) = 1 \quad q=k$$

$$(\mathbf{p^T}_\mathbf{q}\mathbf{p_k}) = 0 \quad q\neq k$$

- Therefore, the first equation becomes: $\mathbf{a} = \mathbf{W}\mathbf{p_k} = \mathbf{t_k}$
  - The output of the network is equal to the target output
- This shows that, if the input prototype vectors are orthonormal, the Hebb rule will produce the correct output for each input

Performance analysis

- But what about non-orthogonal prototype vectors? Let's assume that each vector is unit length, but that they are not orthogonal. Then the previous equation becomes

$$\mathbf{a} = \mathbf{W}\mathbf{p_k} = \mathbf{t_k} + \boxed{\sum_{q \neq k} \mathbf{t}_q (\mathbf{p}_q^T \mathbf{p_k})} \quad \textbf{error}$$

- Because the vectors are not orthogonal, the network will not produce the correct output. The magnitude of the error will depend on the amount of correlation between the prototype input patterns

Example

- Recall the apple and orange recognition problem

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \text{ for orange} \qquad \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \text{ for apple}$$

- (Note that they are not orthogonal.) If we normalize these inputs and choose as desired outputs -1 and 1, we obtain:

$$\left\{\mathbf{p_1} = \begin{bmatrix} 0.5774 \\ -0.5774 \\ -0.5774 \end{bmatrix}, \mathbf{t_1} = [-1]\right\} \left\{\mathbf{p_2} = \begin{bmatrix} 0.5774 \\ 0.5774 \\ -0.5774 \end{bmatrix}, \mathbf{t_2} = [1]\right\}$$

- Our weight matrix becomes:

$$\mathbf{TP^T} = [-1 \quad 1] \begin{bmatrix} 0.5774 & -0.5774 & -0.5774 \\ 0.5774 & 0.5774 & -0.5774 \end{bmatrix} = [0 \quad 1.1548 \quad 0]$$

Example

- So, if we use our two prototype patterns:

$$\mathbf{W}\mathbf{p_1} = [0 \;\; 1.1548 \;\; 0]\begin{bmatrix} 0.5774 \\ \text{-}0.5774 \\ \text{-}0.5774 \end{bmatrix} = [-0.6668]$$

$$\mathbf{W}\mathbf{p_2} = [0 \;\; 1.1548 \;\; 0]\begin{bmatrix} 0.5774 \\ 0.5774 \\ \text{-}0.5774 \end{bmatrix} = [0.6668]$$

- The outputs are close, but do not quite match the target outputs

Pseudoinverse rule

- When the prototype input patterns are not orthogonal, the Hebb rule produces some errors. There are several procedures that can be used to reduce these errors
- We will discuss one of those procedures, the pseudoinverse rule
- Recall that the task of the linear associator was to produce an output $\mathbf{t_q}$ of for an input $\mathbf{p_q}$
- In other words:

$$\mathbf{W p_q} = \mathbf{t_q} \quad \text{for } q=1,2,\ldots,Q$$

- If it is not possible to choose a weight matrix so that these equations are exactly satisfied, then we want them to be approximately satisfied

Pseudoinverse rule

- One approach would be to choose the weight matrix to minimize the following performance index:

$$F(\mathbf{W}) = \sum_{q=1}^{Q} ||\mathbf{t_q} - \mathbf{W}\mathbf{p_q}||^2$$

- If the prototype input vectors $\mathbf{p_q}$ are orthonormal and we use the Hebb rule to find **W**, then F(**W**) will be zero
- When the input vectors are not orthogonal and we use the Hebb rule, then F(**W**) will be not be zero, and it is not clear that F(**W**) will be minimized
- It turns out that the weight matrix that will minimize F(**W**) is obtained by using the pseudoinverse matrix, which we will define next

Pseudoinverse rule

- The equation two slides back can be rewritten as follows:

$$\mathbf{WP} = \mathbf{T}$$

where $\mathbf{T}=[\mathbf{t}_1,\mathbf{t}_2, \ldots, \mathbf{t}_Q]$ $\mathbf{P}=[\mathbf{p}_1,\mathbf{p}_2, \ldots, \mathbf{p}_Q]$

- Then, the equation in the previous slide can be written:

$$F(\mathbf{W}) = ||\mathbf{T}\text{-}\mathbf{WP}||^2 = ||\mathbf{E}||^2$$

where $\mathbf{E} = \mathbf{T}\text{-}\mathbf{WP}$

and $||\mathbf{E}|| \sum_i \sum_j e^2 ij$

- Note that F(**W**) can be made zero if we can solve **WP=T**. If the P matrix has an inverse, the solution is: $\mathbf{W} = \mathbf{TP}^{-1}$
  - However, this is rarely possible. Normally the vectors $\mathbf{p}_q$ (the columns of **P**) will be independent, but R (the dimension of $\mathbf{p}_q$) will be larger than Q (the number of $\mathbf{p}_q$ vectors). Therefore, **P** will not be a square matrix, and no exact inverse will exist

Pseudoinverse rule

- It is known that the weight matrix that minimizes the equation 2 slides back is given by the pseudoinverse rule:

$$\mathbf{W} = \mathbf{T}\mathbf{P}^{+}$$

where $\mathbf{P}^{+}$ is the Moore-Penrose pseudoinverse

- The pseudoinverse of a real matrix is the unique matrix :

$$\mathbf{P}\mathbf{P}^{+}\mathbf{P} = \mathbf{P}$$

$$\mathbf{P}^{+}\mathbf{P}\mathbf{P}^{+} = \mathbf{P}^{+}$$

$$\mathbf{P}^{+}\mathbf{P} = (\mathbf{P}^{+}\mathbf{P})^{T}$$

$$\mathbf{P}\mathbf{P}^{+} = (\mathbf{P}\mathbf{P}^{+})^{T}$$

- When the number, R, of rows of **P** is greater than the number of columns, Q, of **P**, and the columns **P** of are independent, then the pseudoinverse can be computed by

$$\mathbf{P}^{+} = (\mathbf{P}^{T}\mathbf{P})^{-1}\mathbf{P}^{T}$$

Example

- Consider again the apple and orange recognition problem Recall that the input/output prototype vectors are:

$$\left\{\mathbf{p_1} = \begin{bmatrix} 1 \\ \text{-1} \\ \text{-1} \end{bmatrix}, \mathbf{t_1} = [-1]\right\}\left\{\mathbf{p_2} = \begin{bmatrix} 1 \\ 1 \\ \text{-1} \end{bmatrix}, \mathbf{t_2} = [1]\right\}$$

- (Note that we do not need to normalize the input vectors when using the pseudoinverse rule.)
- The weight matrix is calculated from $\mathbf{W}=\mathbf{TP}^{+}$

$$\mathbf{W} = \mathbf{TP}^{+} = [-1\ 1]\left(\begin{bmatrix} 1 & 1 \\ \text{-1} & 1 \\ \text{-1} & \text{-1} \end{bmatrix}\right)$$

Example

where the pseudoinverse is computed from:

$$\mathbf{P}^{+} = (\mathbf{P}^{\mathbf{T}}\mathbf{P})^{-1}\mathbf{P}^{\mathbf{T}} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}^{-1} \begin{bmatrix} 1 & \text{-1} & \text{-1} \\ 1 & 1 & \text{-1} \end{bmatrix} = \begin{bmatrix} 0.25 & \text{-0.5} & \text{-0.25} \\ 0.25 & 0.5 & \text{-0.25} \end{bmatrix}$$

- This produces the following weight matrix:

$$\mathbf{W} = \mathbf{T}\mathbf{P}^{+} = [-1 \;\; 1] \begin{bmatrix} 0.25 & \text{-0.5} & \text{-0.25} \\ 0.25 & 0.5 & \text{-0.25} \end{bmatrix} = [0 \;\; 1 \;\; 0]$$

- Let's try this matrix on our two prototype patterns.

$$\mathbf{W}\mathbf{p_1} = [0 \;\; 1 \;\; 0] \begin{bmatrix} 1 \\ \text{-1} \\ \text{-1} \end{bmatrix} = [-1]$$

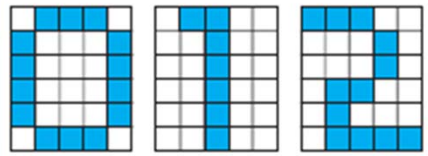
$$\mathbf{W}\mathbf{p_2} = [0 \;\; 1 \;\; 0] \begin{bmatrix} 1 \\ 1 \\ \text{-1} \end{bmatrix} = [1]$$

Application

- Now let's see how we might use the Hebb rule on a practical, although greatly oversimplified, pattern recognition problem. For this problem we will use a special type of associative memory — the autoassociative memory
- In an *autoassociative memory* the desired output vector is equal to the input vector (i.e., $\mathbf{t_q}=\mathbf{p_q}$). We will use an autoassociative memory to store a set of patterns and then to recall these patterns, even when corrupted patterns are provided as input

Application

- The patterns we want to store are:



$\mathbf{p}_1,\mathbf{t}_1$ $\mathbf{p}_2,\mathbf{t}_2$ $\mathbf{p}_3,\mathbf{t}_3$

  - Since we are designing an autoassociative memory, these patterns represent the input vectors and the targets
- They represent the digits {0, 1, 2} displayed in a 6X5 grid. We need to convert these digits to vectors, which will become the prototype patterns for our network. Each white square will be represented by a "-1", and each dark square will be represented by a "1". Then, to create the input vectors, we will scan each 6X5 grid one column at a time
- For example, the first prototype pattern will be:

$$\mathbf{p}_1 = [-1\ \ 1\ \ 1\ \ 1\ \ 1\ \ -1\ \ 1\ \ -1\ \ -1\ \ -1\ \ -1\ \ 1\ \ 1\ \ -1\ \ldots\ 1\ \ -1]^T$$

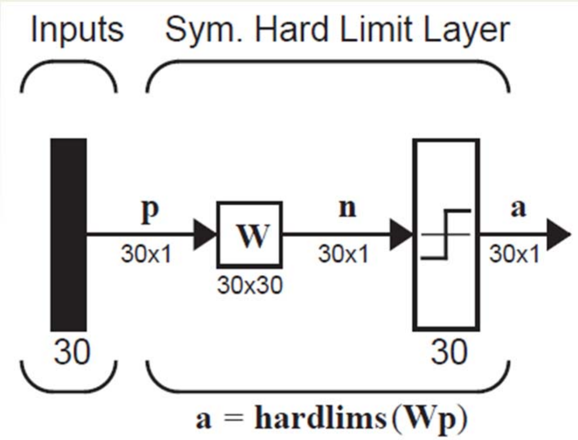
Application

- The vector $\mathbf{p}_1$ corresponds to the digit "0", $\mathbf{p}_2$ to the digit "1", and $\mathbf{p}_3$ to the digit "2". Using the Hebb rule, the weight matrix is computed:

$$\mathbf{W} = \mathbf{p}_1\mathbf{p}_1^T + \mathbf{p}_2\mathbf{p}_2^T + \mathbf{p}_3\mathbf{p}_3^T$$

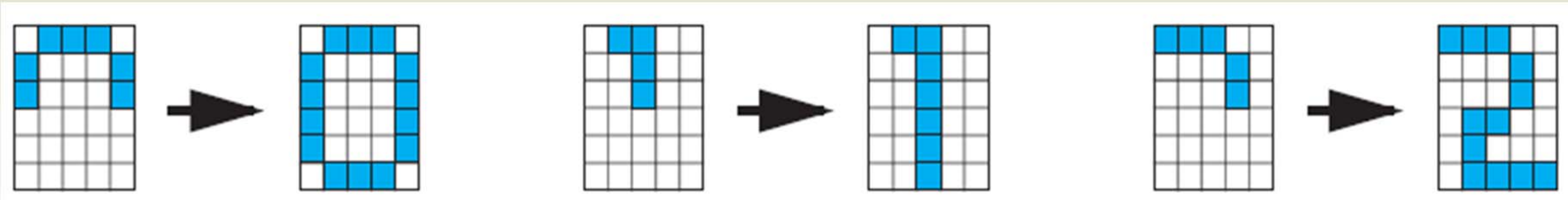
  - Note that $\mathbf{p}_q$ replaces $\mathbf{t}_q$ in the respective equation, since this is autoassociative memory

- Because there are only two allowable values for the elements of the prototype vectors, we will modify the linear associator so that its output elements can only take on values of "-1" or "1". We can do this by replacing the linear transfer function with a symmetrical hard limit transfer function. The resulting network is:



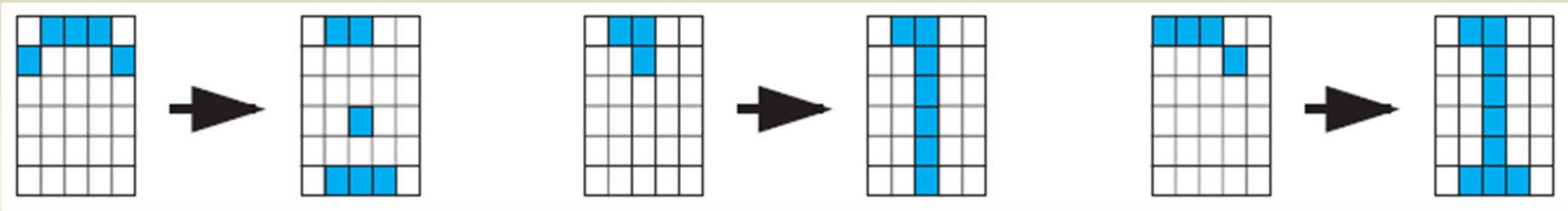
Application

- Now let's investigate the operation of this network. We will provide the network with corrupted versions of the prototype patterns and then check the network output. In the first test, the network is presented with a prototype pattern in which the lower half of the pattern is occluded. In each case the correct pattern is produced by the network



Application

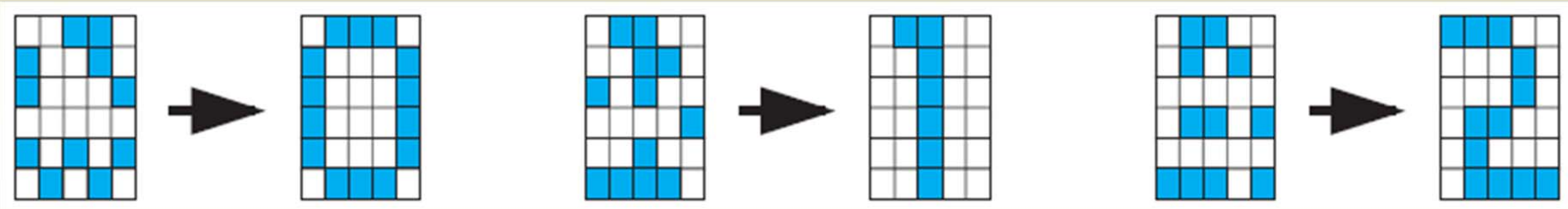
- In the next test we remove even more of the prototype patterns. The next figure illustrates the result of removing the lower two-thirds of each pattern. In this case only the digit "1" is recovered correctly. The other two patterns produce results that do not correspond to any of the prototype patterns



- This is a common problem in associative memories. We would like to design networks so that the number of such spurious patterns would be minimized. The solution are the recurrent associative memories

Application

- In our final test we will present the autoassociative network with noisy versions of the prototype pattern. To create the noisy patterns we will randomly change seven elements of each pattern. The results are shown below. For these examples all of the patterns were correctly recovered



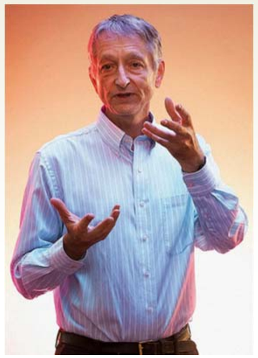
Hopfield network

Recall: Nobel Prize in Physics 2024

https://www.nobelprize.org/prizes/physics/2024/summary/

“for foundational discoveries and inventions that enable machine learning with artificial neural networks”

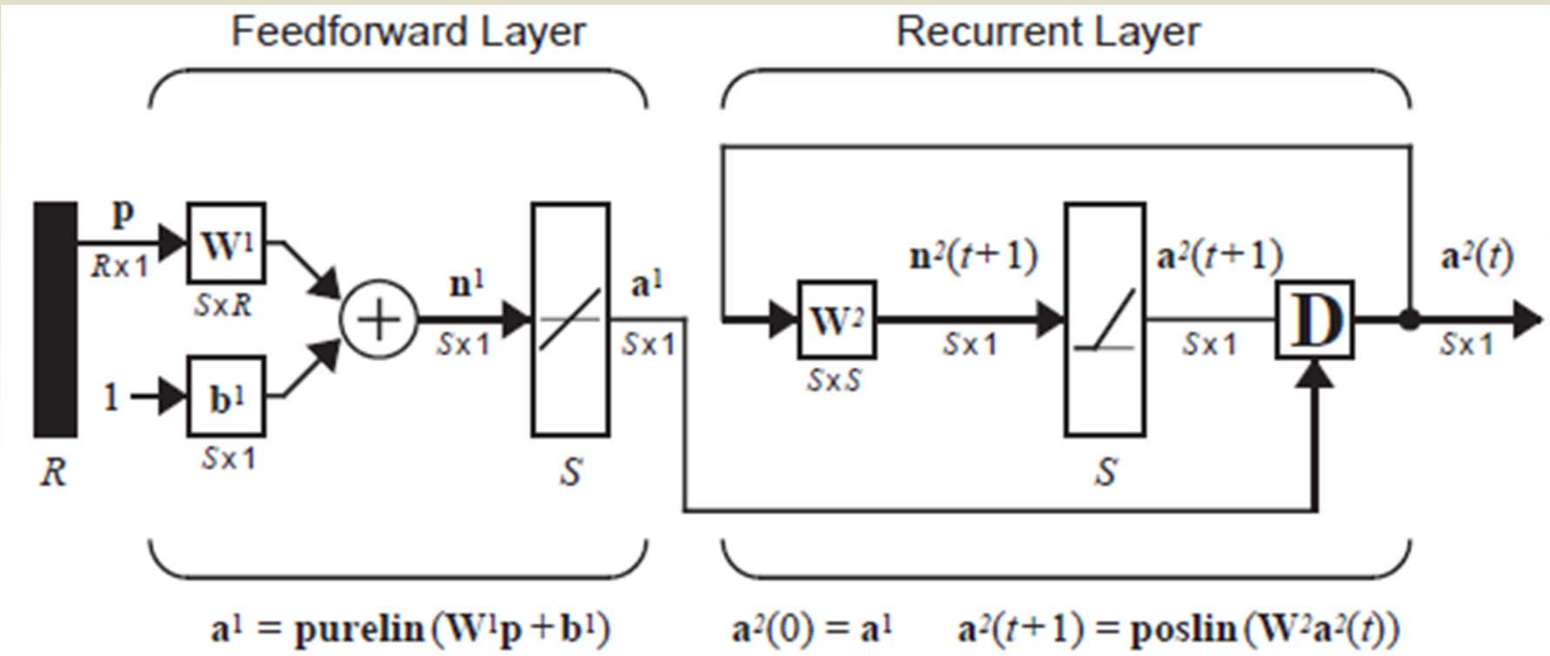
John J. Hopfield
Born: 15 July 1933, Chicago, IL, USA
Affiliation at the time of the award: Princeton University, Princeton, NJ, USA



Geoffrey E. Hinton
Born: 1947, London, UK.
PhD 1978 from The University of Edinburgh, UK.
Affiliation at the time of the award: Professor at University of Toronto, Canada

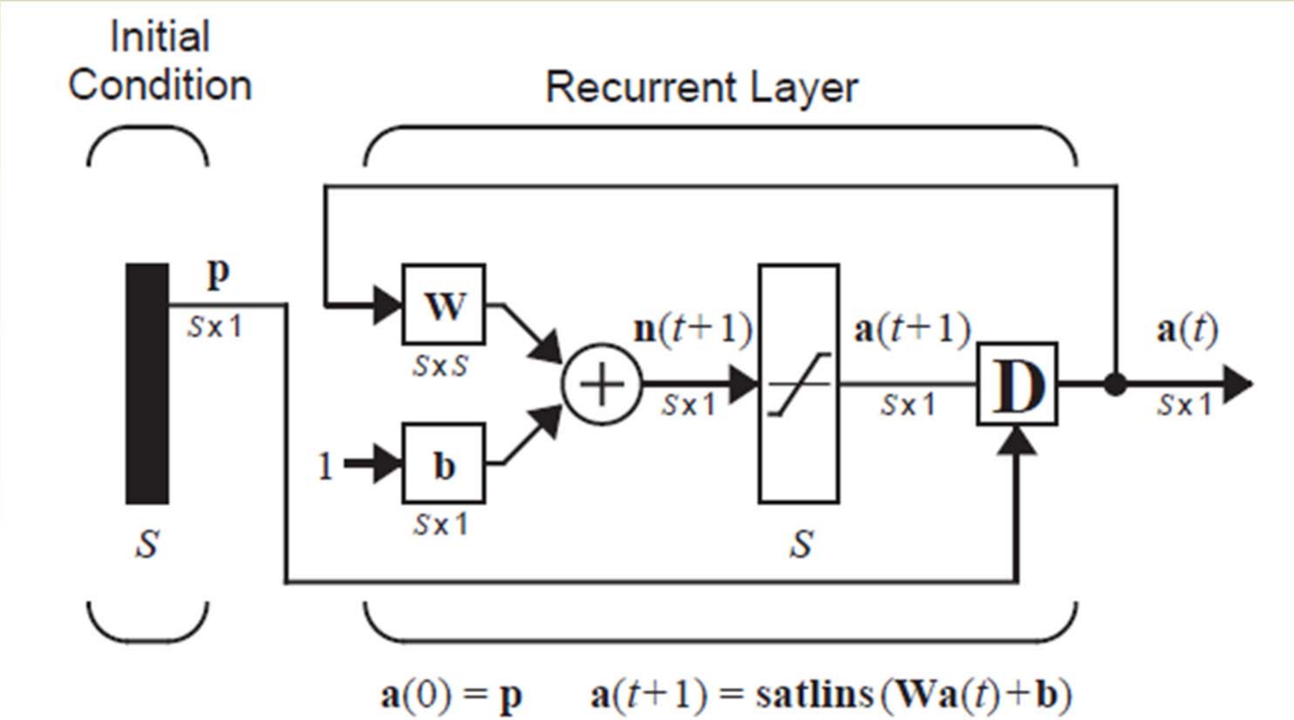
Recall: Hamming network

- The Hamming network was designed explicitly to solve binary pattern recognition problems (where each element of the input vector has only two possible values)
- It uses both feedforward and recurrent (feedback) layers
- The number of neurons in the first layer is the same as the number of neurons in the second layer



Hopfield network

- This is a recurrent network that is similar to the recurrent layer of the Hamming network, but which can effectively perform the operations of both layers of the Hamming network. Its diagram is shown below (The figure is actually a slight variation of the standard Hopfield network. We use this variation because it is somewhat simpler to describe and yet demonstrates the basic concepts.)



Hopfield network

- The neurons in this network are initialized with the input vector, then the network iterates until the output converges
- When the network is operating correctly, the resulting output should be one of the prototype vectors
  - Therefore, whereas in the Hamming network the nonzero neuron indicates which prototype pattern is chosen, the Hopfield network actually produces the selected prototype pattern at its output

Hopfield network

- The equations that describe the network operation are

$$\mathbf{a}(0) = \mathbf{p}$$

- and

$$\mathbf{a}(t + 1) = \mathit{satlins}(\mathbf{Wa}(t) + b)$$

  where *satlins* is the transfer function that is linear in the range [-1, 1] and saturates at 1 for inputs greater than 1 and at -1 for inputs less than -1
- The design of the weight matrix and the bias vector for the Hopfield network is a more complex procedure than it is for the Hamming network, where the weights in the feedforward layer are the prototype patterns

Hopfield network

- To illustrate the operation of the network, we have determined a weight matrix and a bias vector that can solve our orange and apple pattern recognition
- problem. They are given below:

$$\mathbf{W} = \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 1.2 & 0 \\ 0 & 0 & 0.2 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 0.9 \\ 0 \\ -0.9 \end{bmatrix}$$

- We want the network output to converge to either the orange pattern, $\mathbf{p}_1$, or the apple pattern, $\mathbf{p}_2$. In both patterns, the first element is 1, and the third element is -1. The difference between the patterns occurs in the second element. Therefore, no matter what pattern is input to the network, we want the first element of the output pattern to converge to 1, the third element to converge to -1, and the second element to go to either 1 or -1, whichever is closer to the second element of the input vector

Hopfield network

- The equations of operation of the Hopfield network, using the parameters **W**, **b** given in the previous slide, are:

$$a_1(t+1) = \mathit{satlins}(0.2a_1(t) + 0.9)$$
$$a_2(t+1) = \mathit{satlins}(1.2a_2(t))$$
$$a_3(t+1) = \mathit{satlins}(0.2a_3(t) - 0.9)$$

- Regardless of the initial values, $a_i(0)$, the first element will be increased until it saturates at 1, and the third element will be decreased until it saturates at -1. The second element is multiplied by a number larger than 1. Therefore, if it is initially negative, it will eventually saturate at -1; if it is initially positive it will saturate at 1

Hopfield network

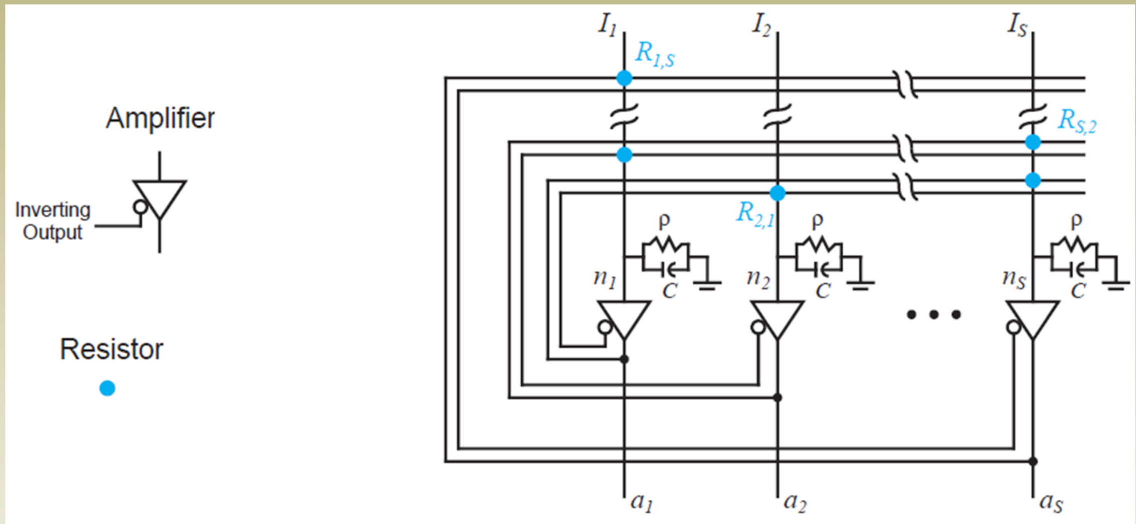
- Let's again take our oblong orange to test the Hopfield network
- The outputs of the Hopfield network for the first three iterations would be

$$\mathbf{a}(0) = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}, \ \mathbf{a}(1) = \begin{bmatrix} 0.7 \\ -1 \\ -1 \end{bmatrix}, \ \mathbf{a}(2) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \ \mathbf{a}(3) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

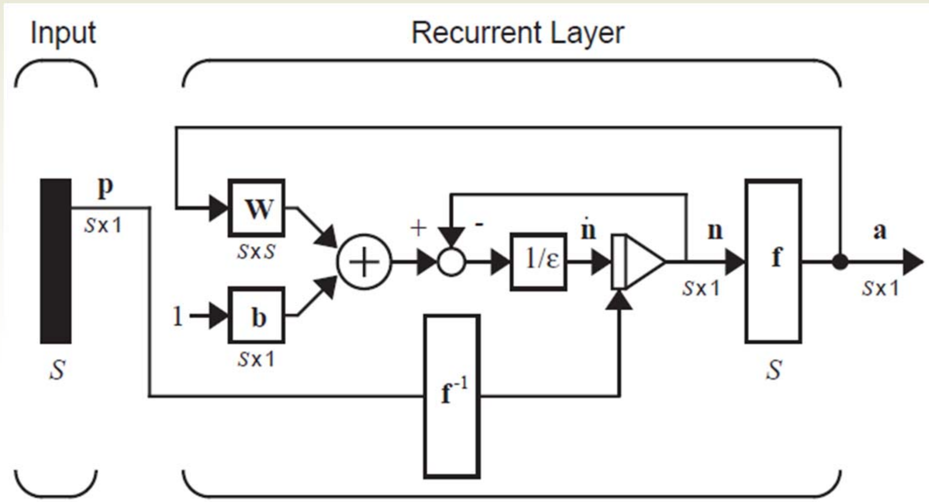
- The network has converged to the *orange* pattern, as did the Hamming network, although each network operated in a different way
  - In the Hamming network the single nonzero neuron indicated which prototype pattern had the closest match. If the first neuron was nonzero, that indicated *orange*, and if the second neuron was nonzero, that indicated *apple*. In the Hopfield network the prototype pattern itself appears at the output of the network

The original Hopfield model & network

- Hopfield presented his model as an electrical circuit



Each neuron is represented by an operational amplifier and its associated resistor/capacitor network.
There are two sets of inputs to the neurons. The first set, represented by the currents $I_1, I_2, \ldots$, are constant external inputs. The other set consists of feedback connections from other op-amps. For instance, the second output, a_2, is fed to resistor $R_{S,2}$, which is connected, in turn, to the input of amplifier S. Resistors are, of course, only positive, but a negative input to a neuron can be obtained by selecting the inverted output of a particular amplifier



$$\mathbf{n}(0) = \mathbf{f}^{-1}(\mathbf{p}), \ (\mathbf{a}(0) = \mathbf{p}) \qquad \varepsilon\, d\mathbf{n}/dt = -\mathbf{n} + \mathbf{W}\,\mathbf{f}(\mathbf{n}) + \mathbf{b}$$