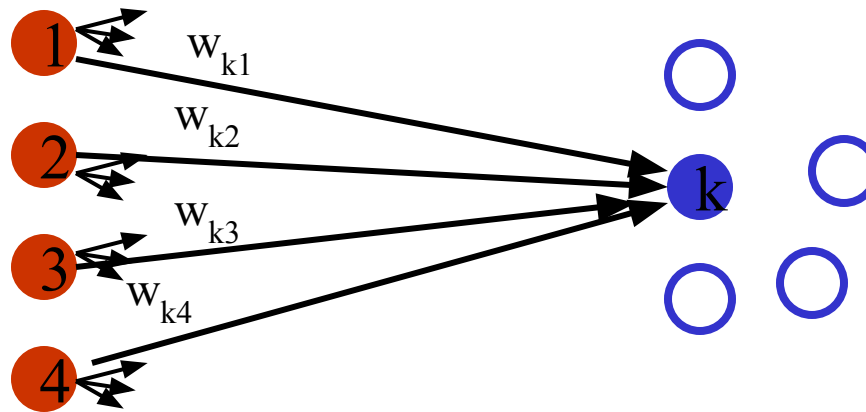


# Unsupervised Learning with Artificial Neural Networks

- The ANN is given a set of patterns,  $P$ , from space,  $S$ , but little/no information about their classification, evaluation, interesting features, etc. It must learn these by itself!
- Tasks
  - Clustering - Group patterns based on similarity (Focus of this lecture)
  - Vector Quantization - Fully divide up  $S$  into a small set of regions (defined by codebook vectors) that also helps cluster  $P$ .
  - Probability Density Approximation - Find small set of points whose distribution matches that of  $P$ .
  - Feature Extraction - Reduce dimensionality of  $S$  by removing unimportant features (i.e. those that do not help in clustering  $P$ )

# Weight Vectors in Clustering Networks

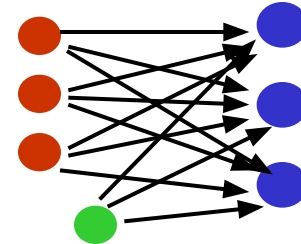


- Node  $k$  represents a particular class of input vectors, and the weights into  $k$  encode a prototype/centroid of that class.
- So if  $\text{prototype}(\text{class}(k)) = [i_{k1}, i_{k2}, i_{k3}, i_{k4}]$ , then:  
 $w_{km} = f_e(i_{km})$  for  $m = 1..4$ , where  $f_e$  is the encoding function.
- In some cases, the encoding function involves normalization. Hence  
 $w_{km} = f_e(i_{k1} \dots i_{k4})$ .
- The weight vectors are learned during the unsupervised training phase.

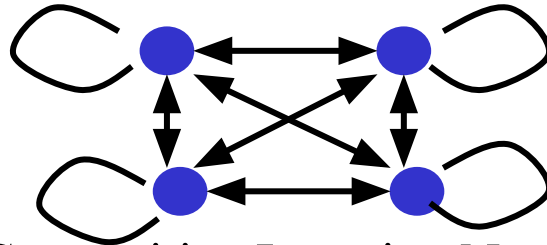
# Network Types for Clustering

- Winner-Take-All Networks

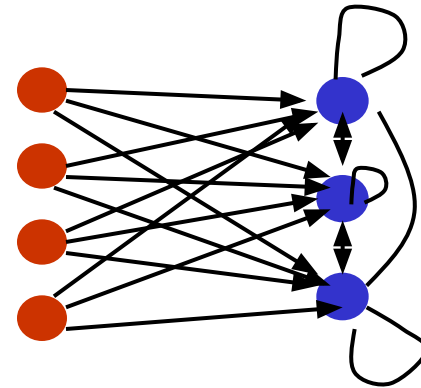
- Hamming Networks



- Maxnet

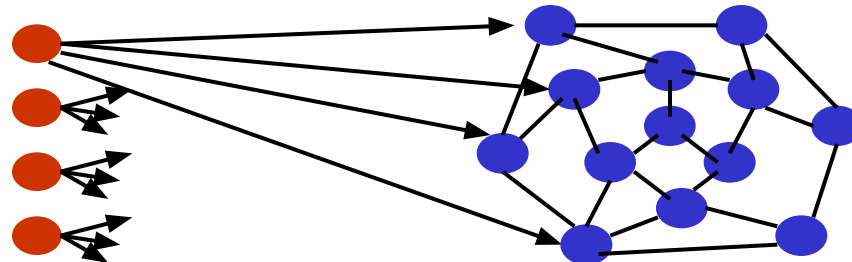


- Simple Competitive Learning Networks



- Topologically Organized Networks

- Winner & its neighbors “take some”



# Hamming Networks

Given: a set of patterns  $m$  patterns,  $P$ , from an  $n$ -dim input space,  $S$ .

Create: a network with  $n$  input nodes and  $m$  simple linear output nodes (one per pattern), where the incoming weights to the output node for pattern  $p$  is based on the  $n$  features of  $p$ .

$i_{pj}$  =  $j$ th input bit of the  $p$ th pattern.  $i_{pj} = 1$  or  $-1$

Set  $w_{pj} = i_{pj}/2$ .

Also include a threshold input of  $-n/2$  at each output node.

Testing: enter an input pattern,  $I$ , and use the network to determine which member of  $P$  that is closest to  $I$ . Closeness is based on the Hamming distance (# non-matching bits in the two patterns).

Given input  $I$ , the output value of the output node for pattern  $p =$

$$\sum_{k=1}^n w_{pk} I_k - \frac{n}{2} = \sum_{k=1}^n \frac{i_{pk}}{2} I_k - \frac{n}{2} = \frac{1}{2} \sum_{k=1}^n i_{pk} I_k - \frac{n}{2}$$

$=$  the negative of the Hamming distance between  $I$  and  $p$

# Hamming Networks (2)

Proof: (that output of output node p is the negative of the Hamming distance between p and input vector I).

Assume: k bits match.

Then: n - k bits do not match, and n-k is the Hamming distance.

And: the output value of p's output node is:

$$\frac{1}{2} \sum_{k=1}^n i_{pk} I_k = n - \frac{1}{2} (k - (n - k)) = n - k$$

k matches, where each match gives (1)(1) or (-1)(-1) = 1

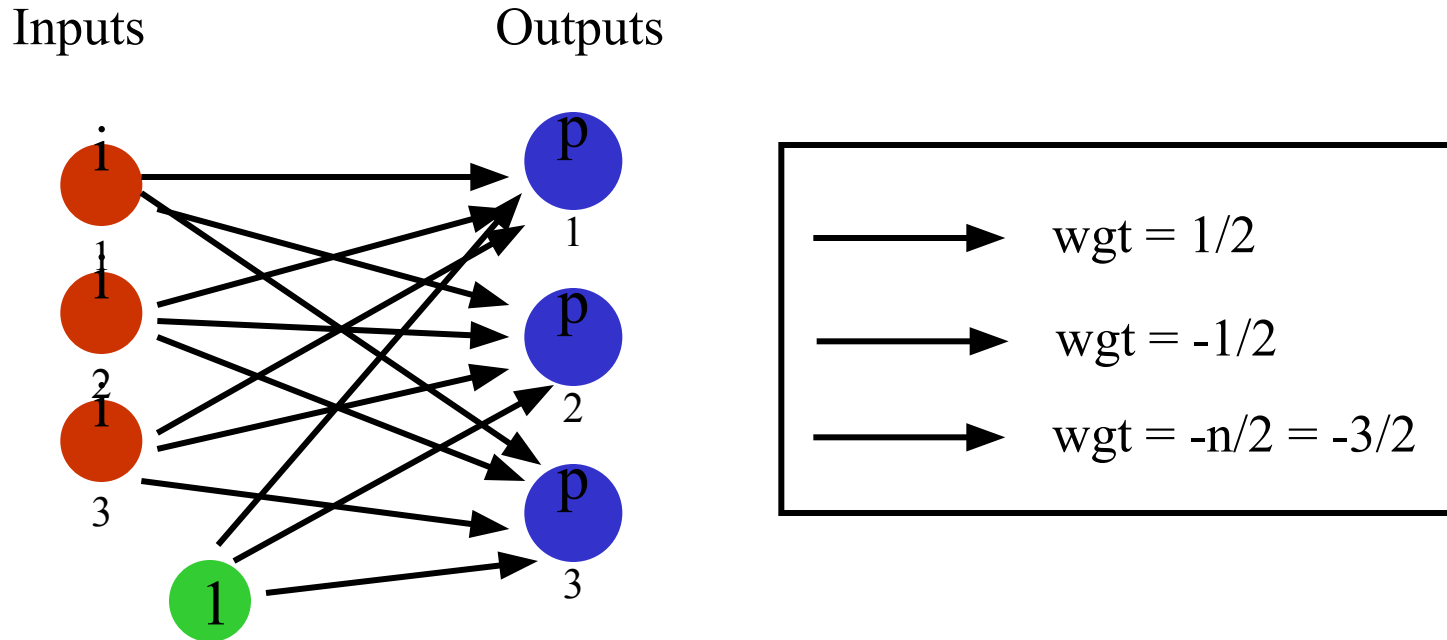
Neg. Hamming distance

n-k mismatches, where each gives (-1)(1) or (1)(-1) = -1

The pattern  $p^*$  with the largest negative Hamming distance to I is thus the pattern with the smallest Hamming distance to I (i.e. the nearest to I). Hence, the output node that represents  $p^*$  will have the highest output value of all output nodes: it wins!

# Hamming Network Example

$P = \{(1 \ 1 \ 1), (-1 \ -1 \ -1), (1 \ -1 \ 1)\} = 3$  patterns of length 3



Given: input pattern  $I = (-1 \ 1 \ 1)$

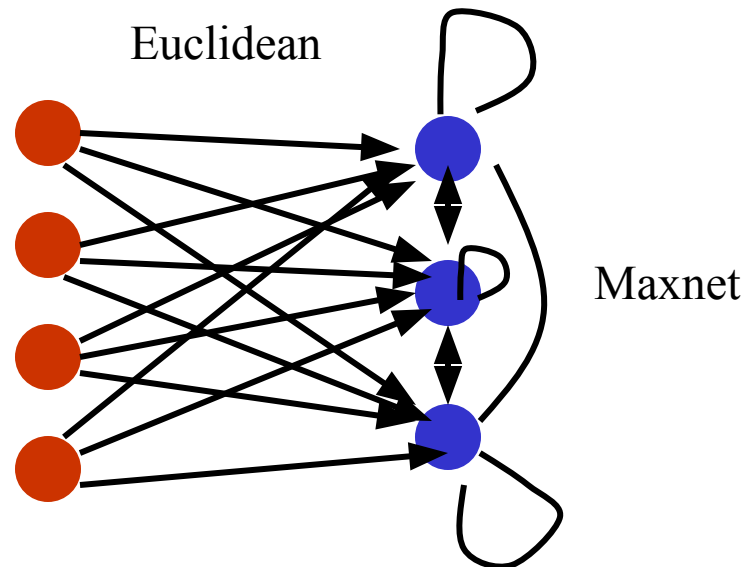
Output ( $p_1$ ) =  $-1/2 + 1/2 + 1/2 - 3/2 = -1$  (Winner)

Output ( $p_2$ ) =  $1/2 - 1/2 - 1/2 - 3/2 = -2$

Output ( $p_3$ ) =  $-1/2 - 1/2 + 1/2 - 3/2 = -2$

# Simple Competitive Learning

- Combination Hamming-like Net + Maxnet with learning of the input-to-output weights.
  - Inputs can be real valued, not just 1, -1.
    - So distance metric is actually Euclidean or Manhattan, not Hamming.
- Each output node represents a centroid for input patterns it wins on.
- Learning: winner node's incoming weights are updated to move closer to the input vector.



# Winning & Learning

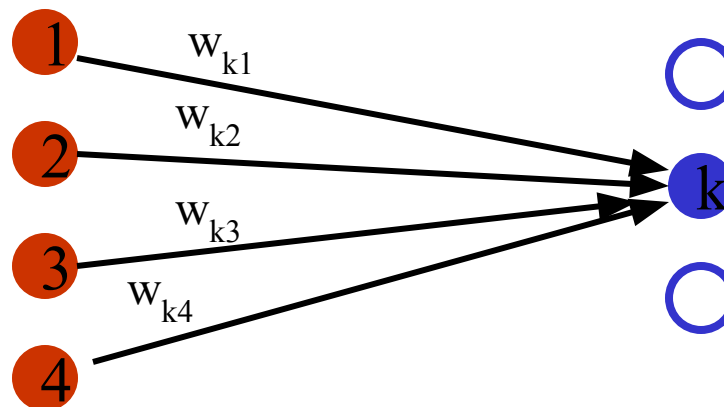
“Winning isn’t everything...it’s the *ONLY* thing” - Vince Lombardi

- Only the incoming weights of the winner node are modified.
- Winner = output node whose incoming weights are the shortest Euclidean distance from the input vector.

$$\sqrt{\sum_{i=1}^n I_i^2 w_{ki}^2} = \text{Euclidean distance from input vector } I \text{ to the vector represented by output node } k \text{'s incoming weights}$$

- Update formula: If  $j$  is the winning output node:

$$w_{ji}(\text{new}) = w_{ji}(\text{old}) + \alpha (I_i - w_{ji}(\text{old}))$$



Note: The use of real-valued inputs & Euclidean distance means that the simple product of weights and inputs does not correlate with “closeness” as in binary networks using Hamming distance.



# SCL Examples (1)

## 6 Cases:

(0 1 1)      (1 1 0.5)  
(0.2 0.2 0.2)    (0.5 0.5 0.5)  
(0.4 0.6 0.5)    (0 0 0)

Learning Rate: 0.5

## Initial Randomly-Generated Weight Vectors:

[ 0.14   0.75   0.71 ]  
[ 0.99   0.51   0.37 ]    Hence, there are 3 classes to be learned  
[ 0.73   0.81   0.87 ]

## Training on Input Vectors

Input vector # 1:    [ 0.00   1.00   1.00 ]  
  Winning weight vector # 1: [ 0.14   0.75   0.71 ] Distance:   0.41  
  Updated weight vector: [ 0.07   0.87   0.85 ]  
Input vector # 2:    [ 1.00   1.00   0.50 ]  
  Winning weight vector # 3: [ 0.73   0.81   0.87 ] Distance:   0.50  
  Updated weight vector: [ 0.87   0.90   0.69 ]

# SCL Examples (2)

Input vector # 3: [ 0.20 0.20 0.20 ]

Winning weight vector # 2: [ 0.99 0.51 0.37 ] Distance: 0.86

Updated weight vector: [ 0.59 0.36 0.29 ]

Input vector # 4: [ 0.50 0.50 0.50 ]

Winning weight vector # 2: [ 0.59 0.36 0.29 ] Distance: 0.27

Updated weight vector: [ 0.55 0.43 0.39 ]

Input vector # 5: [ 0.40 0.60 0.50 ]

Winning weight vector # 2: [ 0.55 0.43 0.39 ] Distance: 0.25

Updated weight vector: [ 0.47 0.51 0.45 ]

Input vector # 6: [ 0.00 0.00 0.00 ]

Winning weight vector # 2: [ 0.47 0.51 0.45 ] Distance: 0.83

Updated weight vector: [ 0.24 0.26 0.22 ]

Weight Vectors after epoch 1:

[ 0.07 0.87 0.85 ]

[ 0.24 0.26 0.22 ]

[ 0.87 0.90 0.69 ]

# SCL Examples (3)

## Clusters after epoch 1:

Weight vector # 1: [ 0.07 0.87 0.85 ]

Input vector # 1: [ 0.00 1.00 1.00 ]

Weight vector # 2: [ 0.24 0.26 0.22 ]

Input vector # 3: [ 0.20 0.20 0.20 ]

Input vector # 4: [ 0.50 0.50 0.50 ]

Input vector # 5: [ 0.40 0.60 0.50 ]

Input vector # 6: [ 0.00 0.00 0.00 ]

Weight vector # 3: [ 0.87 0.90 0.69 ]

Input vector # 2: [ 1.00 1.00 0.50 ]

## Weight Vectors after epoch 2:

[ 0.03 0.94 0.93 ]

[ 0.19 0.24 0.21 ]

[ 0.93 0.95 0.59 ]

## Clusters after epoch 2:

unchanged.

# SCL Examples (4)

## 6 Cases

(0.9 0.9 0.9)	(0.8 0.9 0.8)
(1 0.9 0.8)	(1 1 1)
(0.9 1 1.1)	(1.1 1 0.7)

## Other parameters:

Initial Weights from Set: {0.8 1.0 1.2}

Learning rate: 0.5

# Epochs: 10

- Run same case twice, but with different initial randomly-generated weight vectors.
- The clusters formed are highly sensitive to the initial weight vectors.

# SCL Examples (5)

## Initial Weight Vectors:

[ 1.20 1.00 1.00 ] \* All weights are medium to high  
[ 1.20 1.00 1.20 ]  
[ 1.00 1.00 1.00 ]

## Clusters after 10 epochs:

Weight vector # 1: [ 1.07 0.97 0.73 ]  
Input vector # 3: [ 1.00 0.90 0.80 ]  
Input vector # 6: [ 1.10 1.00 0.70 ]  
Weight vector # 2: [ 1.20 1.00 1.20 ]  
Weight vector # 3: [ 0.91 0.98 1.02 ]  
Input vector # 1: [ 0.90 0.90 0.90 ]  
Input vector # 2: [ 0.80 0.90 0.80 ]  
Input vector # 4: [ 1.00 1.00 1.00 ]  
Input vector # 5: [ 0.90 1.00 1.10 ]

\*\* Weight vector #3 is the big winner & #2 loses completely!!

# SCL Examples (6)

## Initial Weight Vectors:

```
[ 1.00  0.80  1.00 ]    * Better balance of initial weights
[ 0.80  1.00  1.20 ]
[ 1.00  1.00  0.80 ]
```

## Clusters after 10 epochs:

```
Weight vector # 1: [ 0.83  0.90  0.83 ]
  Input vector # 1:  [ 0.90  0.90  0.90 ]
  Input vector # 2:  [ 0.80  0.90  0.80 ]
Weight vector # 2: [ 0.93  1.00  1.07 ]
  Input vector # 4:  [ 1.00  1.00  1.00 ]
  Input vector # 5:  [ 0.90  1.00  1.10 ]
Weight vector # 3: [ 1.07  0.97  0.73 ]
  Input vector # 3:  [ 1.00  0.90  0.80 ]
  Input vector # 6:  [ 1.10  1.00  0.70 ]
** 3 clusters of equal size!!
```

Note: All of these SCL examples were run by a simple piece of code that had NO neural-net model, but merely a list of weight vectors that was continually updated.

# SCL Variations

- Normalized Weight & Input Vectors

$$\text{Distance} = \sqrt{\sum_{i=1}^n I_i^2} - 2 \sum_{i=1}^n I_i w_{ki} + \sqrt{\sum_{i=1}^n w_{ki}^2}$$

- To minimize distance, maximize:

$$\sum_{i=1}^n I_i w_{ki} \quad \sum_{i=1}^n I_i^2 = 1 \quad \sum_{i=1}^n w_{ki}^2 = 1 \quad \text{Due to Normalization}$$

When I and W are normal vectors (i.e. Length = 1) and  $\theta$  = IW angle

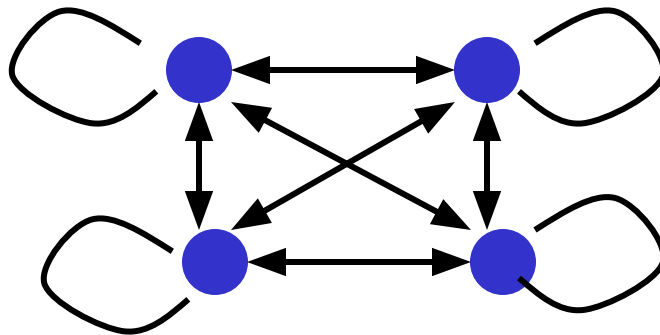
$$\cos \theta = I \cdot W$$

Normalization of  $I = \langle i_1 \ i_2 \dots i_n \rangle$ .  $W = \langle w_1 \ w_2 \dots w_n \rangle$  normalized similarly.

$$L = \sqrt{i_1^2 + i_2^2 + \dots + i_n^2} \quad \tilde{I} = \left\langle \frac{i_1}{L}, \frac{i_2}{L}, \dots, \frac{i_n}{L} \right\rangle$$

So by keeping all vectors normalized, the dot product of the input and weight vector is the cosine of the angle between the two vectors, and a high value means a small angle (i.e., the vectors are nearly the same). Thus, the node with the highest net value is the nearest neighbor to the input.

# Maxnet



$\longleftrightarrow$   $wgt \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$   
 $\longrightarrow$   $wgt \begin{bmatrix} & \\ & \end{bmatrix}$   
 e.g.  $\begin{bmatrix} & & 1 & \\ & & & \end{bmatrix} \begin{bmatrix} & \\ & \end{bmatrix} \frac{1}{n}$

Simple network to find node with largest initial input value.

Topology: clique with self-arcs, where all self-arcs have a small positive (excitatory) weight, and all other arcs have a small negative (inhibitory) weight.

Nodes: have transfer function  $f_T = \max(\text{sum}, 0)$

Algorithm:

Load initial values into the clique

Repeat:

Synchronously update all node values via  $f_T$

Until: all but one node has a value of 0

Winner = the non-zero node



# Maxnet Examples

- Input values: (1, 2, 5, 4, 3) with epsilon = 1/5 and theta = 1

0.000	0.000	3.000	1.800	0.600
0.000	0.000	2.520	1.080	0.000
0.000	0.000	2.304	0.576	0.000
0.000	0.000	2.189	0.115	0.000
0.000	0.000	2.166	0.000	0.000
0.000	0.000	2.166	0.000	0.000

$$= (1)5 - (0.2)(1+2+4+3)$$

- Input values: (1, 2, 5, 4.5, 4.7) with epsilon = 1/5 and theta = 1

Stable attractor

0.000	0.000	2.560	1.960	2.200
0.000	0.000	1.728	1.008	1.296
0.000	0.000	1.267	0.403	0.749
0.000	0.000	1.037	0.000	0.415
0.000	0.000	0.954	0.000	0.207
0.000	0.000	0.912	0.000	0.017
0.000	0.000	0.909	0.000	0.000
0.000	0.000	0.909	0.000	0.000

Stable attractor

# Maxnet Examples (2)

- Input values: (1, 2, 5, 4, 3) with epsilon = 1/10 and theta = 1

0.000	0.700	4.000	2.900	1.800
0.000	0.000	3.460	2.250	1.040
0.000	0.000	3.131	1.800	0.469
0.000	0.000	2.904	1.440	0.000
0.000	0.000	2.760	1.150	0.000
0.000	0.000	2.645	0.874	0.000
0.000	0.000	2.558	0.609	0.000
0.000	0.000	2.497	0.353	0.000
0.000	0.000	2.462	0.104	0.000
0.000	0.000	2.451	0.000	0.000
0.000	0.000	2.451	0.000	0.000

- Input values: (1, 2, 5, 4, 3) with epsilon = 1/2 and theta = 1

0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000

Stable attractor

\* Epsilon > 1/n with theta = 1 => too much inhibition

Stable attractor