Competitive Learning
Some Types of Learning

- **Supervised learning**: training using desired response for given stimuli ("rote" learning)

- **Unsupervised learning**: classification by "clustering" of stimuli, without specified response

- **Hybrid**: e.g. unsupervised to form cluster, supervised to learn desired response to class
Competitive Learning

- A form of *unsupervised* learning, but *combinable with supervised* learning.

- Neurons “compete” based on proximity to input pattern.

- Neuron closest to pattern (the “*winner*”) adjusts its weight to be still closer.
Already Seen

- We’ve seen an example of competitive learning:
  Lateral inhibition networks
Input presentation carries the assumption that the network is supposed to “learn” the input.

Presented input pattern

The “winner”

Neurons
2-way competition

Presented input pattern

Input presentation carries the assumption that the network is supposed to “learn” the input.

Neurons

The “winner” becomes more like the input.

The “loser” stays as is.
Why not make the winner exactly like the input?

- There may be many more distinct input patterns than neurons.

- By “averaging” its behavior, a neuron can put a large number of distinct, but similar inputs into the same category.
Categorizing Inputs by 2 neurons

wins

wins
An Application

- Displaying an image file with “millions of colors” using a palate of, say, 256 colors.

- Each color in the image has to be mapped into one of the colors in the palate.

- Map each image color into the closest one of the 256 palate colors.

- In this case, a competitive network can learn a reasonable set of colors to use for a given image.
Related Applications

- Use the reduction in number of colors of the image to *store* a version of the image more *compactly*:

  \[(1\text{M color } \rightarrow 256 \text{ colors reduces the number of } bits \text{ by a factor of } 2.5),\]

  or to *transmit* the version over a slow channel.
A Competitive Neural Network

- When presented with patterns from the same selection of inputs repeatedly, will tend to stabilize so that its neurons are representatives of clusters of closer inputs.

- Each neuron will tend to be similar to inputs in its cluster (like a chameleon, perhaps)
Measures of similarity or closeness (opposite: dissimilarity or distance)

• Suppose $x$ is an input vector and $w_i$ the weight vector of the $i^{th}$ neuron.

• One measure of distance is the Euclidean distance:

$$
\| x - w_i \| = \sqrt{\sum_i (x_j - w_{ij})^2}
$$

$$
= \sqrt{(x_j - w_{ij})(x_j - w_{ij})^T}
$$

(vector inner product)
Measures of distance

- The discrete metric:

\[ d(x, w_i) = 0 \text{ if } x = w_i, \]
\[ 1 \text{ otherwise} \]
Another measure of distance, used when the values are integer, is the “Manhattan” “city-block”, or “taxi cab” distance:

\[ \| x - w_i \| = \sum_i ( |x_j - w_{ij}| ) \]
Another measure of distance, used when the values are \textit{2-valued}, is the “Hamming distance”:

$$\sum_i ( |x_j == w_{ij}| )$$

0 when the values are equal, 1 otherwise
Richard Hamming (1915-1998)
Properties of Distance (i.e. a Metric)

- \(d : P \times P \rightarrow R\)
  where \(P\) is the space of patterns and \(R\) is the set of real numbers

- \((\forall x, y) \; d(x, y) > 0\)

- \((\forall x) \; d(x, y) = 0 \iff x = y\)

- \((\forall x, y) \; d(x, y) = d(y, x)\)

- \((\forall x, y, z) \; d(x, z) \leq d(x, y) + d(y, z)\) (triangle inequality)
Example for Different Metrics

- Suppose \( \mathbf{x} = [1 \ 1 \ -1 \ 1] \), \( \mathbf{w} = [1 \ -1 \ -1 \ -1] \)

- Euclidean distance = \( \sqrt{0^2 + 2^2 + 0^2 + 2^2} = 2.83... \)

- Discrete metric = 1

- Manhattan distance = \( 0 + 2 + 0 + 2 = 4 \)

- Hamming distance = \( 0 + 1 + 0 + 1 = 2 \)
A measure of similarity is given by the inner product

The inner product

\[ x \cdot w_i \]

is larger when \( x \) is “closer to” \( w_i \).

Therefore inner product is not a metric.

Usually it is best if \( x \) and \( w_i \) are normalized before using this measure:

\[ \| x \| = \| w_i \| = 1 \]

In this case, \( (1 - xw_i)/2n \) is a distance, where \( n \) is the dimension.
Inner Product is not a Metric

- $\begin{bmatrix} 1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & -1 & 1 \end{bmatrix} = 4$
- $\begin{bmatrix} 1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & -1 & 1 & -1 \end{bmatrix} = -4$
- $\begin{bmatrix} 1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -1 & -1 & -1 \end{bmatrix} = 0$
Inner product as cosine

- When the pattern space consists of vectors of numbers, the normalized inner product is the \textit{cosine} of the angle between $x$ and $w_i$ as vectors.
Determining a Winner

- The winner is the neuron with weight either:
  - the smallest distance to the input, or
  - the largest inner product with the input.

- Again, if inner products are used, it is best to **normalize** the weight and input first, or use only normalized values.
Example: Hamming Network (competitive, non-learning)

Input x connects to neuron weight vectors representing stored patterns.

Inner products are calculated for each input with the neuron weight vectors.

The max network selects the neuron with the highest inner product, indicated by the '1-hot' code output.

The winner indicator shows the selected neuron.

Neuron weight vectors = stored patterns
“Neural” Implementation of a Max Sub-Network

- It is very similar to “activity bubble” formation
- a recurrent neural net that cycles values through neurons, eliminating one loser each cycle until only the winner is left.
- (This is totally unnecessary from a strictly computational viewpoint.)
- Each neuron has as inputs the outputs of all neurons including itself.
- Self-weights are 1; Weights from other neurons are \(-\varepsilon\), where \(\varepsilon\) is any quantity < \(1/(\#\ of\ neurons)\).
Max Network

- Activation functions are “poslin”:
  \[ \text{poslin}(x) = x \text{ if } x > 0, \ 0 \text{ otherwise} \]

- The network is operated \textit{synchronously}.

- The initial outputs are forced to those of the input values.

- On each cycle, each neuron computes \text{poslin}(\text{weighted inputs}).
Max Network

- For the $i^{th}$ neuron
  $$y_i := \text{poslin}(y_i - \varepsilon \sum_{j \neq i} y_j)$$
  $$= (1+\varepsilon)y_i - \varepsilon \sum y_j$$

- These weights are designed so that:
  - all but one output is non-zero after $n$ cycles (assuming inputs were originally distinct)
  - all outputs persist at the same value after $n$ cycles
MaxNet Example

- $n = 4$ neurons, take $\varepsilon = 0.2 < 1/4$

\[ y_i := (1+\varepsilon)y_i - \varepsilon \sum y_j \]

<table>
<thead>
<tr>
<th>step</th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
<th>$y_4$</th>
<th>sum</th>
<th>epsilon</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0000</td>
<td>1.0000</td>
<td>4.0000</td>
<td>2.0000</td>
<td>10.0000</td>
<td>0.2000</td>
<td></td>
</tr>
<tr>
<td>1.6000</td>
<td>0.0000</td>
<td>2.8000</td>
<td>0.4000</td>
<td>4.8000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9600</td>
<td>0.0000</td>
<td>2.4000</td>
<td>0.0000</td>
<td>3.3600</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4800</td>
<td>0.0000</td>
<td>2.2080</td>
<td>0.0000</td>
<td>2.6880</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0384</td>
<td>0.0000</td>
<td>2.1120</td>
<td>0.0000</td>
<td>2.1504</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>2.1043</td>
<td>0.0000</td>
<td>2.1043</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>2.1043</td>
<td>0.0000</td>
<td>2.1043</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

winner
Matlab *compet* function (non-learning)

`compet(M)` takes one input argument, 
M: an matrix column vectors.
It returns output vectors with 1 where each net input 
vector has its maximum value, and 0 elsewhere.

```
>> M = [0 1; 5 2; 1 4; 2 3]
M =
 0  1
 5  2
 1  4
 2  3
>> C = compet(M)
C =
  (2,1)  1
  (3,2)  1
>> size(C)
ans =
 4  2
```

This is Matlab’s sparse matrix notation.

Caution:
If more than 1 winner in a column, 
only the first is indicated.

```
>> M = [0 1; 5 2; 1 4; 2 3]
M =
 0  1
 5  2
 1  4
 2  3
>> compet(M)
ans =
  (2,1)  1
  (3,2)  1
```
Using Competition in Conjunction with Learning

- Input presented
- Winner selected
- The winner learns

- Others “close to” winner may learn as well.
Instar Rule (seen before)

**Instar Rule** (Stephen Grossberg)

\[ i \mathbf{w}(q) = i \mathbf{w}(q - 1) + \alpha a_i(q)(\mathbf{p}(q) - i \mathbf{w}(q - 1)) \]

- pattern - weight
- \( 1 \) for \( i = \) winner
- \( 0 \) otherwise

Only winner learns

learning rate
Graphical Representation

\[ i^* \mathbf{w}(q) = i^* \mathbf{w}(q - 1) + \alpha (\mathbf{p}(q) - i^* \mathbf{w}(q - 1)) \]

\[ i^* \mathbf{w}(q) = (1 - \alpha) i^* \mathbf{w}(q - 1) + \alpha \mathbf{p}(q) \]
Graphical Representation

\[ i^*w(q) = (1 - \alpha) i^*w(q - 1) + \alpha p(q) \]

\[ i^*w(q) = i^*w(q - 1) + \alpha (p(q) - i^*w(q - 1)) \]
Matlab Demos

- nnd14cl (competitive learning)
- A vector input is chosen on the circumference of a circle.
- A competition is held among the neurons.
- The closest neuron learns by moving closer to the input.
- Other neurons stay put.
Competitive Learning vs. Clustering

- The preceding example shows that competitive learning is one way to achieve clustering.

- We discussed the k-means clustering algorithm earlier.

- In k-means, the computation of new centers can be viewed as a kind of competition, since the new centers are computed by summing the inputs closest to a given old center.
Possible Instability

If the input vectors don’t fall into nice clusters, then for **large learning rates** the presentation of each input vector may modify the configuration so that the system will undergo continual evolution.
Possible Instability

If the input vectors don’t fall into nice clusters, then for large learning rates the presentation of each input vector may modify the configuration so that the system will undergo continual evolution.

**Solution:** Gradually **decrease the learning rate** (“annealing”).
Another problem with competitive learning is that neurons with initial weights far from any input vector may never win and thus become useless.
Have a Heart

**Solution:** Add a negative bias to each neuron, and increase the magnitude of the bias as the neuron wins. This will make it harder for a neuron to win if it has won often.

This is called the “**conscience**” method.
Competitive Learning with Explicit Output Classification

Learning Vector-Quantization

Counterpropagation (see later)
VQ is a technique used to **reduce the dimensionality of data**.

The original data is a set of vectors, of dimension $n$, say.

The vectors are mapped into a set of smaller dimension, $m$, of **codebook values**.

The codebook values are used for storage or transmission (some information content is lost).

The codebook values, upon receipt or retrieval, are re-translated into values close to the original data values (will not be exact).
Many-one codebook values are close to a subset of original values.
inverse-image = some kind of clustering
codebook values
close to a subset of original

many-one

one-one
Other Clustering Methods

- There are many.
- For a survey, see this tutorial:
  http://www.pitt.edu/~csna/reports/janowitz.pdf
  which was listed on the website of the:

  Classification Society of North America

- (k-means clustering is mentioned on page 30).
Learning Vector-Quantization
(LVQ)
Learning Vector Quantization (LVQ)

- Combine **competitive** learning with **supervision**.

- Competitive learning achieves clusters.

- Assign a **class** (or output value) to each cluster.

- Reinforce cluster representative (a neuron) when it **classifies** input in the **desired** class:
  - **Positive reinforcement**: pull the neuron weights toward the input.
  - **Negative reinforcement**: push the weights away.
Learning Vector Quantization

Training examples:
\[ \{ \mathbf{p}_1, \mathbf{t}_1 \}, \{ \mathbf{p}_2, \mathbf{t}_2 \}, \ldots, \{ \mathbf{p}_Q, \mathbf{t}_Q \} \]

If the input pattern is classified **correctly**, then move the *winning* weight *toward* the input vector according to the rule:

\[
_i \mathbf{w}^1(q) = _i \mathbf{w}^1(q - 1) + \alpha (\mathbf{p}(q) - _i \mathbf{w}^1(q - 1))
\]

\[ a_{k*} = t_{k*} = 1 \]

If the input pattern is classified **incorrectly**, then move the *winning* weight *away* from the input vector:

\[
_i \mathbf{w}^1(q) = _i \mathbf{w}^1(q - 1) - \alpha (\mathbf{p}(q) - _i \mathbf{w}^1(q - 1))
\]

\[ a_{k*} = 1 \neq t_{k*} = 0 \]
LVQ Rules

Input pattern correctly classified

Input pattern incorrectly classified

neurons

input

input
Learning Vector Quantization: Second Layer

- The classification mapping from selection of cluster representative to **output** can be accomplished by a second layer, following the competitive layer.

- This second layer is essentially a matrix multiply, so can be represented by neural weights as usual. These weights may be pre-assigned and fixed.
LVQ xor Example

Example patterns with targets:

\[
\begin{align*}
\{ & p_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, t_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \} \\
\{ & p_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, t_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \} \\
\{ & p_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \} \\
\{ & p_4 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, t_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \}
\end{align*}
\]

Competitive layer weights (4 neurons):

\[
W^1(0) = \begin{bmatrix}
(1w^1)^T \\
(2w^1)^T \\
(3w^1)^T \\
(4w^1)^T
\end{bmatrix} = \begin{bmatrix}
0.25 & 0.75 \\
0.75 & 0.75 \\
1 & 0.25 \\
0.5 & 0.25
\end{bmatrix}
\]

Classification layer weight (pre-assigned):

\[
W^2 = \begin{bmatrix}
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1
\end{bmatrix}
\]
Iteration 1, first layer

\[ a^1 = \text{compet}(n^1) = \text{compet} \left( \begin{bmatrix} -\|w^1_1 - p_1\| \\ -\|w^1_2 - p_1\| \\ -\|w^1_3 - p_1\| \\ -\|w^1_4 - p_1\| \end{bmatrix} \right) \]

\[ a^1 = \text{compet} \left( \begin{bmatrix} -\left\| \begin{bmatrix} 0.25 & 0.75 \end{bmatrix}^T - \begin{bmatrix} 0 & 1 \end{bmatrix}^T \right\| \\ -\left\| \begin{bmatrix} 0.75 & 0.75 \end{bmatrix}^T - \begin{bmatrix} 0 & 1 \end{bmatrix}^T \right\| \\ -\left\| \begin{bmatrix} 1.00 & 0.25 \end{bmatrix}^T - \begin{bmatrix} 0 & 1 \end{bmatrix}^T \right\| \\ -\left\| \begin{bmatrix} 0.50 & 0.25 \end{bmatrix}^T - \begin{bmatrix} 0 & 1 \end{bmatrix}^T \right\| \end{bmatrix} \right) = \text{compet} \left( \begin{bmatrix} -0.354 \\ -0.791 \\ -1.25 \\ -0.901 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \]
Iteration 1, second layer

\[ \mathbf{a}^2 = \mathbf{W}^2 \mathbf{a}^1 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \]

This is the correct class, therefore the weight vector is moved toward the input vector.

\[ \mathbf{w}^1(1) = \mathbf{w}^1(0) + \alpha(\mathbf{p}_1 - \mathbf{w}^1(0)) \]

\[ \mathbf{w}^1(1) = \begin{bmatrix} 0.25 \\ 0.75 \end{bmatrix} + 0.5 \left( \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.25 \\ 0.75 \end{bmatrix} \right) = \begin{bmatrix} 0.125 \\ 0.875 \end{bmatrix} \]
Blue vs. black represent class of neuron, desired class of input sample.

Selected input sample changes to green if correctly classified, red if not; weights adjusted toward or away, depending on class.
Variant: LVQ2

- If the winning neuron in the hidden layer **correctly** classifies the current input, the response is the same as in LVQ1.

- If the winning neuron in the hidden layer **incorrectly** classifies the current input, we move its weight vector **away** from the input vector, as in LVQ,

  and

- move the weights of the **closest** neuron to the input vector that **would have correctly classified** the input **toward** the input vector.
Like LV1, except in addition to pulling or pushing the closest, if the closest is not in the correct class, then in addition, the closest that would be correct is moved closer.
LVQ2 Rules

Input pattern correctly classified

Input pattern incorrectly classified

neurons

input

neurons

input