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.

---

2-way competition

- Presented input pattern
- Neurons
- The “**winner**”
- Input presentation carries the assumption that the network is supposed to “learn” the input.
2-way competition

- Present input pattern
- Neurons

The "winner" becomes more like the input.

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

The "loser" stays as is.

Why not make the winner \textit{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

An Application

- Display an image file with “millions of colors” on a graphic display with, say, 256 colors.
- Each color in the image has to be mapped into one of the colors.
- Map each image color into the closest one of the 256.
An Application, continued

- The actual choice of the 256 might not be fixed; it is likely a limitation of some hardware table (of RGB values) rather than a limitation of the screen itself.

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

A Related Application

- Use the reduction in number of colors of the image to store a version of the image more compactly (1M color → 256 colors reduces the number of bits by a factor of 2.5),

or to transmit the version image 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: 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

- Another measure of distance, used when the values are integer, is the “Manhattan” or “city-block” distance:

\[ \| x - w_i \| = \sum_i ( |x_i - w_{ij}| ) \]

Measures of distance

- Another measure of distance, used when the values are 2-valued, is the “Hamming distance”:

\[ \sum_i ( |x_j = w_{ij}| ) \]

0 when the values are equal, 1 otherwise
A measure of similarity is given by the inner product

The inner product

\[ x \cdot w_i \]

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

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

\[ \| x \| = \| w_i \| = 1 \]
Inner product as cosine

- The normalized inner product is the cosine of the angle between $x$ and $w_i$ as vectors.

![Diagram](x, w_i)

Example for Different Metrics

- Suppose $x = [1 \ 1 \ -1 \ 1]$, $w = [1 \ -1 \ -1 \ -1]$
- Euclidean distance $= \sqrt{0^2 + 2^2 + 0^2 + 2^2} = 2.83...$
- Manhattan distance $= 0 + 2 + 0 + 2 = 4$
- Hamming distance $= 0 + 1 + 0 + 1 = 2$
- inner product $= [1 \ 1 \ -1 \ 1] [1 \ -1 \ -1 \ -1]^T = 0$

smaller is closer

larger is closer
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)

input \( x \)  

neuron weight vectors = stored patterns

inner products

max network

“1-hot” code

winner indicator

(Hamming network does not learn)
Max Sub-Network

- a recurrent neural net that cycles values through neurons, eliminating one loser each cycle until only the winner is left.
- 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 <$1/(\text{# of neurons})$.

Max Network

- Activation functions are “poslin”:
  \[
  \text{poslin}(x) = \begin{cases} 
  x & \text{if } x > 0, \\
  0 & \text{otherwise}
  \end{cases}
  \]
- The network is operated \textit{synchronously}.
- The initial outputs are forced to those of the input values.
- On each cycle, each neuron computes poslin(weighted inputs).
Max Network

- For the \(i\)th neuron
  \[ y_i := \text{poslin}(y_i - \sum_{j \neq i} y_j) \]
  \[ = (1+\epsilon) y_i - \epsilon y_j \]

- 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 \(\epsilon = 0.2 < 1/4\)
  \[ y_i := (1+\epsilon) y_i - \epsilon y_j \]

<table>
<thead>
<tr>
<th>step</th>
<th>(y_1)</th>
<th>(y_2)</th>
<th>(y_3)</th>
<th>(y_4)</th>
<th>(\text{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>
Matlab *compet* function (non-learning)

COMPET(N) takes one input argument,  
N - SxQ matrix of net input (column) vectors. 
and returns output vectors with 1 where each net input 
vector has its maximum value, and 0 elsewhere.

```
compet([-3; -1; 5; 2; -9]) ==> (3,1) 1
```

Sparse matrix notation: row 3, col 1 = 1

a = compet(n)  
= compet(Wp)
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)

\[ \Delta w(q) = \Delta w(q-1) + \lambda a_i(q)(p(q) - w(q-1)) \]

- 1 for \( i = \text{winner} \)
- 0 otherwise

learning rate

Only winner learns
Kohonen Rule
(when specialized to single winner = Instar Rule)

In the general Kohonen rule, there can be multiple “winners”.

Graphical Representation

\[ \mathbf{w}(q) = \mathbf{w}(q-1) + \Delta \mathbf{p}(q) \]

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

\[ \mathbf{w}(q) = \mathbf{w}(q-1) - \Delta \mathbf{p}(q) \]
Graphical Representation

\[ \dot{w}(q) = \dot{w}(q-1) + \beta(p(q) - \dot{w}(q-1)) \]

\[ \dot{w}(q) = (1 - \beta) \dot{w}(q-1) + \beta p(q) \]

Matlab Demos

- nnd14cl (competitive learning)
Matlab Demos

- **democ1**

  Data points (red)

  2D weights of neurons (coinciding initially)

  The neurons are now better *representatives* of the data.
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”).
“Dead” Units / Starvation

One 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 to win if a neuron has won often. This is called the “conscience” method.