The Perceptron Model

\[ y = 1 \text{ for } \sum w_i x_i \geq \theta \]
\[ = 0 \text{ otherwise} \]
Perceptron Training Algorithm

1. Start with a random value of $w$ ex: $<0,0,0...>$
2. Test for $wx_i > 0$
   If the test succeeds for $i=1,2,...n$
   then return $w$
3. Modify $w$, $w_{next} = w_{prev} + x_{fail}$
Feedforward Network
Example - XOR

$\theta = 0.5$
$w_1 = 1$
$w_2 = 1$

Calculation of XOR

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Calculation of $x_1x_2$

$\theta = 1$
$w_1 = -1$
$w_2 = 1.5$

$0 < \Theta$
$w_2 \geq \Theta$
$w_1 < \Theta$
$w_1 + w_2 < \Theta$
Example - XOR

\[w_1 = 1\]

\[w_2 = 1\]

\[\theta = 0.5\]
Can Linear Neurons Work?

\[ y = m_3x + c_3 \]

\[ y = m_2x + c_2 \]

\[ y = m_1x + c_1 \]

\[ h_1 = m_1(w_1x_1 + w_2x_2) + c_1 \]

\[ h_1 = m_1(w_1x_1 + w_2x_2) + c_1 \]

\[ \text{Out} = (w_5h_1 + w_6h_2) + c_3 \]

\[ = k_1x_1 + k_2x_2 + k_3 \]
**Note:** The whole structure shown in earlier slide is reducible to a single neuron with given behavior

\[ Out = k_1x_1 + k_2x_2 + k_3 \]

**Claim:** A neuron with linear I-O behavior can’t compute X-OR.

**Proof:** Considering all possible cases:

- [assuming 0.1 and 0.9 as the lower and upper thresholds]
  \[ m(w_1.0 + w_2.0 - \theta) + c < 0.1 \]
  For (0,0), Zero class: \[ \Rightarrow c - m.\theta < 0.1 \]

- \[ m(w_2.1 + w_1.0 - \theta) + c > 0.9 \]
  For (0,1), One class: \[ \Rightarrow m.w_1 - m.\theta + c > 0.9 \]
For (1,0), One class: \[ m.w_1 - m.\theta + c > 0.9 \]

For (1,1), Zero class: \[ m.w_1 - m.\theta + c > 0.9 \]

These equations are inconsistent. Hence X-OR can’t be computed.

**Observations:**

1. A linear neuron can’t compute X-OR.
2. A multilayer FFN with linear neurons is collapsible to a single linear neuron, hence no a additional power due to hidden layer.
3. Non-linearity is essential for power.
Multilayer Perceptron
Gradient Descent Technique

- Let $E$ be the error at the output layer

$$E = \frac{1}{2} \sum_{j=1}^{p} \sum_{i=1}^{n} (t_i - o_i)^2_j$$

- $t_i = \text{target output}; \ o_i = \text{observed output}$

- $i$ is the index going over $n$ neurons in the outermost layer
- $j$ is the index going over the $p$ patterns (1 to $p$)
- Ex: XOR: $p=4$ and $n=1$
Weights in a FF NN

- \( w_{mn} \) is the weight of the connection from the \( n^{th} \) neuron to the \( m^{th} \) neuron.
- \( E \) vs \( \overline{W} \) surface is a complex surface in the space defined by the weights \( w_{ij} \).
- \( -\frac{\delta E}{\delta w_{mn}} \) gives the direction in which a movement of the operating point in the \( w_{mn} \) coordinate space will result in maximum decrease in error.

\[
\Delta w_{mn} \propto -\frac{\delta E}{\delta w_{mn}}
\]
Sigmoid neurons

- Gradient Descent needs a derivative computation - not possible in perceptron due to the discontinuous step function used!
  → Sigmoid neurons with easy-to-compute derivatives used!

- Computing power comes from non-linearity of sigmoid function.

\[ y \to 1 \text{ as } x \to \infty \]
\[ y \to 0 \text{ as } x \to -\infty \]
Derivative of Sigmoid function

\[ y = \frac{1}{1 + e^{-x}} \]

\[ \frac{dy}{dx} = -\frac{1}{(1 + e^{-x})^2} (-e^{-x}) = \frac{e^{-x}}{(1 + e^{-x})^2} \]

\[ = \frac{1}{1 + e^{-x}} \left( 1 - \frac{1}{1 + e^{-x}} \right) = y(1 - y) \]
Training algorithm

- Initialize weights to random values.
- For input \(x = \langle x_n, x_{n-1}, \ldots, x_0 \rangle\), modify weights as follows:
  
  Target output = \(t\), Observed output = \(o\)

  \[
  \Delta w_i \propto -\frac{\delta E}{\delta w_i}
  \]

  \[
  E = \frac{1}{2} (t - o)^2
  \]

- Iterate until \(E < \delta\) (threshold)
Calculation of $\Delta w_i$

$$\frac{\delta E}{\delta w_i} = \frac{\delta E}{\delta net} \times \frac{\delta net}{\delta w_i} \left( \text{where: net} = \sum_{i=0}^{n-1} w_i x_i \right)$$

$$= \frac{\delta E}{\delta o} \times \frac{\delta o}{\delta net} \times \frac{\delta net}{\delta w_i}$$

$$= -(t - o) o (1 - o) x_i$$

$$\Delta w_i = -\eta \frac{\delta E}{\delta w_i} \left( \eta = \text{learning constant}, \ 0 \leq \eta \leq 1 \right)$$

$$\Delta w_i = \eta (t - o) o (1 - o) x_i$$
Observations

*Does the training technique support our intuition?*

- The larger the $x_i$, larger is $\Delta w_i$
  - Error burden is borne by the weight values corresponding to large input values