**Neuro-fuzzy Systems & Supervised Learning**

Khurshid Ahmad,
Professor of Computer Science,
Department of Computer Science
Trinity College,
Dublin-2, IRELAND
December 1st, 2009.
https://www.cs.tcd.ie/Khurshid.Ahmad/Teaching.html

---

**Neuro-fuzzy models**

A fuzzy inference system can be shown to be functionally equivalent to a class of adaptive networks.

The burden of specifying the parameters of the fuzzy inference can be transferred to an algorithm that attempts to learn the value of the parameters

Neuro-fuzzy models: A case study

Consider a first-order Sugeno fuzzy model with two inputs \((x, y)\) and one output \((z)\).

Layer 1  Layer 2  Layer 3  Layer 4  Layer 5

\[ \sum w_{ij}f_{ij} \]

Neuro-fuzzy models: A case study

The solution is a zero-order Takagi-Sugeno model with two inputs \((x_1, x_2)\) and one output \(f\).

Layer 1  Layer 2  Layer 3  Layer 4

\[ w_{ij} = \sum \sum w_{kl} \]
Neuro-fuzzy models

**Adaptive Networks**
A network typically comprises a set of nodes connected by directed links.

Each node performs a static node function on its incoming signals to generate a single node output.

Each link specifies the direction of signal flow from one node to another.

An adaptive network is a network structure whose overall input-output behaviour is determined by a collection of modifiable parameters.

---

Neuro-fuzzy models

Neural Networks 'learn' by adapting in accordance with a training regimen: The network is subjected to particular information environments on a particular schedule to achieve the desired end-result.

There are three major types of training regimens or learning paradigms:

- **SUPERVISED**
- **UN-SUPERVISED**
- **REINFORCEMENT or GRADED**
Supervised Learning

Our focus will be on supervised learning, particularly networks that learn by using the so-called back-propagation algorithm and comprise hidden layers between the input & the output layers.

Supervised Learning

A situation in which the network is functioning as an input/output system. The network receives a vector \( \tilde{x} \) and emits another, \( \tilde{y} \). Supervised learning regimen involves the network being supplied with a sequence of examples

\[(\tilde{x}_1, \tilde{y}_1), (\tilde{x}_2, \tilde{y}_2), \ldots, (\tilde{x}_k, \tilde{y}_k)\]

of "desireable" or "correct" input/output pairs. For each input \( \tilde{x}_i \) the network is supplied \( \tilde{y}_i \), the correct output.
Supervised Learning

SUPERVISED LEARNING IS PERFORMED UNDER THE GUIDANCE OF A TEACHER

![Diagram of Supervised Learning]

Other Learning Systems

UNSUPERVISED LEARNING or SELF-ORGANISATION:

Under this regimen a network modifies itself in response to \( x \) inputs. There are no \( y' \) inputs or a grade/score (see next page).

Therefore, in unsupervised or self-organisation learning there is no EXTERNAL TEACHER or CRITIC to oversee the learning process.

![Diagram of Other Learning Systems]

In an unsupervised regimen there are no specific examples of the function to be learned by the network.
Other Learning Systems

REINFORCEMENT LEARNING or GRADED TRAINING:
This regimen is similar to supervisor training except that, instead of being given the correct output on each individual training trial or learning cycle, the network receives only a score or grade that tells it how well it has done over a sequence of input/output trials.

Other Learning Systems

Reinforcement learning is the on-line learning of an input-output mapping through a process of trial and error designed to maximize a scalar performance index called a reinforcement signal.
Learning Systems

**A neuron learns because it is adaptive:**

- **SUPERVISED LEARNING:** The connection strengths of a neuron are modifiable depending on the input signal received, its output value and a pre-determined or desired response. The desired response is sometimes called *teacher response*. The difference between the desired response and the actual output is called the *error signal*.

- **UNSUPERVISED LEARNING:** In some cases the teacher's response is not available and no error signal is available to guide the learning. When no teacher's response is available the neuron, if properly configured, will modify its weight based only on the input and/or output.

Zurada (1992:59-63)

---

Rosenblatt’s Perceptron

- **Rosenblatt’s Perceptron has the following ‘properties’:**
  1. It can receive inputs from other neurons
  2. The ‘recipient’ neuron can integrate the input
  3. The connection weights are modelled as follows:
     - If the presence of features $x_i$ stimulates the perceptron to *fire* then $w_i$ will be positive;
     - If the presence of features $x_i$ inhibits the perceptron then $w_i$ will be negative.
  4. The output function of the neuron is all-or-none
  5. Learning is a process of modifying the weights
     Whatever a neuron can compute, it can *learn* to compute!
Rosenblatt’s Perceptron

**Logic Gate**: A digital circuit that implements an elementary logical operation. It has one or more inputs but ONLY one output. The conditions applied to the input(s) determine the voltage levels at the output. The output, typically, has two values: ‘0’ or ‘1’.

**Digital Circuit**: A circuit that responds to discrete values of input (voltage) and produces discrete values of output (voltage).

**Binary Logic Circuits**: Extensively used in computers to carry out instructions and arithmetical processes. Any logical procedure may be effected by a suitable combinations of the gates. Binary circuits are typically formed from discrete components like the integrated circuits.

---

**Logic Circuits**: Designed to perform a particular logical function based on **AND**, **OR** (either), and **NOR** (neither). Those circuits that operate between two discrete (input) voltage levels, **high & low**, are described as **binary logic circuits**.

**Logic element**: Small part of a logic circuit, typically, a logic **gate**, that may be represented by the mathematical operators in symbolic logic.
Rosenblatt’s Perceptron

<table>
<thead>
<tr>
<th>Gate</th>
<th>Input(s)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Two (or more)</td>
<td>High $IFF$ both (or all) inputs are high.</td>
</tr>
<tr>
<td>NOT</td>
<td>One</td>
<td>High if input low and vice versa</td>
</tr>
<tr>
<td>OR</td>
<td>Two (or more)</td>
<td>High if any one (or more) inputs are high</td>
</tr>
</tbody>
</table>

Rosenblatt’s Perceptron

<table>
<thead>
<tr>
<th>Gate</th>
<th>Input(s)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND</td>
<td>Two (or more)</td>
<td>High if any one or more inputs low; low if all inputs high</td>
</tr>
<tr>
<td>NOR</td>
<td>Two (or more)</td>
<td>High $IFF$ all inputs are low.</td>
</tr>
<tr>
<td>XOR</td>
<td>Two (or more)</td>
<td>Low if all inputs are identical, otherwise high</td>
</tr>
</tbody>
</table>
Rosenblatt’s Perceptron

A single layer perceptron can carry out a number of logical operations which are performed by a number of computational devices.

The perceptron below performs the AND operation:

\[ \sum = w_1 x_1 + w_2 x_2 + \theta \]

\[ y = 1 \text{ if } \sum \geq \theta; \]
\[ y = 0 \text{ if } \sum < \theta \]

\[ \theta = -1.5 \]

Rosenblatt’s Perceptron

A single layer perceptron can carry out logical operations which are performed by a number of computational devices.

The perceptron below performs the OR operation:

\[ \sum = w_1 x_1 + w_2 x_2 + \theta \]

\[ y = 1 \text{ if } \sum \geq \theta; \]
\[ y = 0 \text{ if } \sum < \theta \]

\[ \theta = 0.5 \]
Rosenblatt’s Perceptron

An informal perceptron learning algorithm:

• If the perceptron fires when it should not, make each \( w_i \) smaller by an amount proportional to \( x_i \).

• If the perceptron fails to fire when it should fire, make each \( w_i \) larger by a similar amount.

Rosenblatt’s Perceptrons: A worked example

<table>
<thead>
<tr>
<th>Weight in Tons</th>
<th>Speed in Machs</th>
<th>Aircraft Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1</td>
<td>Fighter</td>
</tr>
<tr>
<td>0.25</td>
<td>2</td>
<td>Fighter</td>
</tr>
<tr>
<td>0.3</td>
<td>2</td>
<td>Fighter</td>
</tr>
<tr>
<td>0.4</td>
<td>3</td>
<td>Fighter</td>
</tr>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>Fighter</td>
</tr>
<tr>
<td>0.25</td>
<td>0.1</td>
<td>Bomber</td>
</tr>
<tr>
<td>0.4</td>
<td>0.25</td>
<td>Bomber</td>
</tr>
<tr>
<td>0.5</td>
<td>0.2</td>
<td>Bomber</td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>Bomber</td>
</tr>
<tr>
<td>0.7</td>
<td>0.25</td>
<td>Bomber</td>
</tr>
</tbody>
</table>
Rosenblatt’s Perceptrons: A worked example

Weight versus speed of an aircraft

![Graph showing weight versus speed of an aircraft]
Rosenblatt’s Perceptrons: A worked example

Figure: Weight versus speed of an aircraft

- Fighters
- Bombers

Regression lines:
- Logarithmic: \( y = \log(x) \)
- Exponential: \( y = \exp(x) \)
Rosenblatt’s Perceptrons: A worked example

Weight versus speed of an aircraft

Weight in Tons

Speed in Machs

Fighters

Bombers

Quadratic Regression Line: $y = ax^2 + bx + c$

Cubic Regression Line: $y = ax^3 + bx^2 + cx + d$
Rosenblatt’s Perceptrons: A worked example

\[ y = w_1x_1 + w_2x_2 + \theta \]

\[ \delta(y) = 1, \text{if } y \geq 0 \]
\[ \delta(y) = 0, \text{if } y < 0 \]

<table>
<thead>
<tr>
<th>Weight - (x_1)</th>
<th>Speed - (x_2)</th>
<th>(y) (fighter=0; bomber=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.25</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>0</td>
</tr>
<tr>
<td>0.25</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>0.4</td>
<td>0.25</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>0.7</td>
<td>0.25</td>
<td>1</td>
</tr>
</tbody>
</table>

Rosenblatt’s Perceptrons: A worked example

\(\Theta = -0.1, \alpha = 0.1, 1^\text{st Iteration}\)

<table>
<thead>
<tr>
<th>X_{\text{new}}</th>
<th>x_1</th>
<th>y</th>
<th>w_1</th>
<th>w_2</th>
<th>y</th>
<th>(\delta)</th>
<th>e=y-y_i</th>
<th>w_1^\text{new}</th>
<th>w_2^\text{new}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1</td>
<td>0</td>
<td>-0.90</td>
<td>0.50</td>
<td>0.31</td>
<td>1</td>
<td>1</td>
<td>-0.91</td>
<td>0.40</td>
</tr>
<tr>
<td>0.25</td>
<td>2</td>
<td>0</td>
<td>-0.91</td>
<td>0.40</td>
<td>0.47</td>
<td>1</td>
<td>1</td>
<td>-0.94</td>
<td>0.20</td>
</tr>
<tr>
<td>0.3</td>
<td>2</td>
<td>0</td>
<td>-0.94</td>
<td>0.20</td>
<td>0.02</td>
<td>1</td>
<td>1</td>
<td>-0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>0.4</td>
<td>3</td>
<td>0</td>
<td>-0.97</td>
<td>0.00</td>
<td>-0.49</td>
<td>0</td>
<td>0</td>
<td>-0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>0</td>
<td>-0.97</td>
<td>0.00</td>
<td>-0.58</td>
<td>0</td>
<td>0</td>
<td>-0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>0.25</td>
<td>0.1</td>
<td>1</td>
<td>-0.97</td>
<td>0.00</td>
<td>-0.34</td>
<td>0</td>
<td>-1</td>
<td>-0.94</td>
<td>0.01</td>
</tr>
<tr>
<td>0.4</td>
<td>0.25</td>
<td>1</td>
<td>-0.94</td>
<td>0.01</td>
<td>-0.47</td>
<td>0</td>
<td>-1</td>
<td>-0.90</td>
<td>0.04</td>
</tr>
<tr>
<td>0.5</td>
<td>0.2</td>
<td>1</td>
<td>-0.90</td>
<td>0.04</td>
<td>-0.54</td>
<td>0</td>
<td>-1</td>
<td>-0.85</td>
<td>0.06</td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>1</td>
<td>-0.85</td>
<td>0.06</td>
<td>-0.58</td>
<td>0</td>
<td>-1</td>
<td>-0.79</td>
<td>0.11</td>
</tr>
<tr>
<td>0.7</td>
<td>0.25</td>
<td>1</td>
<td>-0.79</td>
<td>0.11</td>
<td>-0.63</td>
<td>0</td>
<td>-1</td>
<td>-0.72</td>
<td>0.13</td>
</tr>
</tbody>
</table>
Rosenblatt’s Perceptrons: A worked example

\( \Theta = -0.1, \alpha = 0.1 \)

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( y )</th>
<th>( w_1 )</th>
<th>( w_2 )</th>
<th>( y' )</th>
<th>( \delta )</th>
<th>( e_i = y - y_i )</th>
<th>( w_i^{\text{new}} )</th>
<th>( w_i^{\text{new}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0</td>
<td>0</td>
<td>-0.90</td>
<td>0.50</td>
<td>0.31</td>
<td>1.0</td>
<td>1.0</td>
<td>-0.91</td>
<td>0.40</td>
</tr>
<tr>
<td>0.25</td>
<td>0.25</td>
<td>0</td>
<td>-0.91</td>
<td>0.40</td>
<td>0.47</td>
<td>1.0</td>
<td>1.0</td>
<td>-0.94</td>
<td>0.20</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
<td>0</td>
<td>-0.94</td>
<td>0.20</td>
<td>0.02</td>
<td>1.0</td>
<td>1.0</td>
<td>-0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>0.4</td>
<td>0.05</td>
<td>0</td>
<td>-0.97</td>
<td>0.00</td>
<td>-0.49</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>-0.97</td>
<td>0.00</td>
<td>-0.58</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>0.7</td>
<td>0.25</td>
<td>1</td>
<td>-0.97</td>
<td>0.00</td>
<td>-0.34</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.94</td>
<td>0.01</td>
</tr>
<tr>
<td>0.25</td>
<td>0.25</td>
<td>1</td>
<td>-0.94</td>
<td>0.01</td>
<td>-0.47</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.90</td>
<td>0.04</td>
</tr>
<tr>
<td>0.4</td>
<td>0.25</td>
<td>1</td>
<td>-0.90</td>
<td>0.04</td>
<td>-0.54</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.85</td>
<td>0.06</td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>1</td>
<td>-0.85</td>
<td>0.06</td>
<td>-0.58</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.79</td>
<td>0.11</td>
</tr>
<tr>
<td>0.7</td>
<td>0.25</td>
<td>1</td>
<td>-0.79</td>
<td>0.11</td>
<td>-0.63</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.72</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Rosenblatt’s Perceptrons: A worked example

\( \Theta = -0.1, \alpha = 0.1; 7^{\text{th}} \) Iteration

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( y )</th>
<th>( w_1 )</th>
<th>( w_2 )</th>
<th>( y' )</th>
<th>( \delta )</th>
<th>( e_i = y - y_i )</th>
<th>( w_i^{\text{new}} )</th>
<th>( w_i^{\text{new}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1</td>
<td>0</td>
<td>0.25</td>
<td>-0.17</td>
<td>-0.25</td>
<td>0.0</td>
<td>0.0</td>
<td>0.25</td>
<td>-0.17</td>
</tr>
<tr>
<td>0.25</td>
<td>0</td>
<td>0</td>
<td>0.25</td>
<td>-0.17</td>
<td>-0.38</td>
<td>0.0</td>
<td>0.0</td>
<td>0.25</td>
<td>-0.17</td>
</tr>
<tr>
<td>0.3</td>
<td>0</td>
<td>0</td>
<td>0.25</td>
<td>-0.17</td>
<td>-0.37</td>
<td>0.0</td>
<td>0.0</td>
<td>0.25</td>
<td>-0.17</td>
</tr>
<tr>
<td>0.4</td>
<td>3</td>
<td>0</td>
<td>0.25</td>
<td>-0.17</td>
<td>-0.51</td>
<td>0.0</td>
<td>0.0</td>
<td>0.25</td>
<td>-0.17</td>
</tr>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>0</td>
<td>0.25</td>
<td>-0.17</td>
<td>-0.23</td>
<td>0.0</td>
<td>0.0</td>
<td>0.25</td>
<td>-0.17</td>
</tr>
<tr>
<td>0.25</td>
<td>0.1</td>
<td>1</td>
<td>0.25</td>
<td>-0.17</td>
<td>-0.05</td>
<td>0.0</td>
<td>0.0</td>
<td>0.275</td>
<td>-0.16</td>
</tr>
<tr>
<td>0.4</td>
<td>0.25</td>
<td>1</td>
<td>0.28</td>
<td>-0.16</td>
<td>-0.03</td>
<td>0.0</td>
<td>0.0</td>
<td>0.315</td>
<td>-0.135</td>
</tr>
<tr>
<td>0.5</td>
<td>0.2</td>
<td>1</td>
<td>0.32</td>
<td>-0.14</td>
<td>0.03</td>
<td>1.0</td>
<td>0.0</td>
<td>0.315</td>
<td>-0.135</td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>1</td>
<td>0.32</td>
<td>-0.14</td>
<td>0.02</td>
<td>1.0</td>
<td>0.0</td>
<td>0.315</td>
<td>-0.135</td>
</tr>
<tr>
<td>0.7</td>
<td>0.25</td>
<td>1</td>
<td>0.32</td>
<td>-0.14</td>
<td>0.09</td>
<td>1.0</td>
<td>0.0</td>
<td>0.315</td>
<td>-0.135</td>
</tr>
</tbody>
</table>
Rosenblatt’s Perceptrons: A worked example

\( \Theta = 0.1, \alpha = 0.1; 13^{\text{th}} \) Iteration

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( y )</th>
<th>( w_1 )</th>
<th>( w_2 )</th>
<th>( y )</th>
<th>( \delta )</th>
<th>( e_i = y - y_i )</th>
<th>( w_{1\text{new}} )</th>
<th>( w_{2\text{new}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1</td>
<td>0</td>
<td>0.44</td>
<td>-0.09</td>
<td>-0.14</td>
<td>0</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.25</td>
<td>2</td>
<td>0</td>
<td>0.44</td>
<td>-0.09</td>
<td>-0.16</td>
<td>0</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.3</td>
<td>2</td>
<td>0</td>
<td>0.44</td>
<td>-0.09</td>
<td>-0.14</td>
<td>0</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.4</td>
<td>3</td>
<td>0</td>
<td>0.44</td>
<td>-0.09</td>
<td>-0.18</td>
<td>0</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>0</td>
<td>0.44</td>
<td>-0.09</td>
<td>-0.01</td>
<td>0</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.25</td>
<td>0.1</td>
<td>1</td>
<td>0.44</td>
<td>-0.09</td>
<td>0.00</td>
<td>1</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.4</td>
<td>0.25</td>
<td>1</td>
<td>0.44</td>
<td>-0.09</td>
<td>0.05</td>
<td>1</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.5</td>
<td>0.2</td>
<td>1</td>
<td>0.44</td>
<td>-0.09</td>
<td>0.10</td>
<td>1</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>1</td>
<td>0.44</td>
<td>-0.09</td>
<td>0.12</td>
<td>1</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
<tr>
<td>0.7</td>
<td>0.25</td>
<td>1</td>
<td>0.44</td>
<td>-0.09</td>
<td>0.19</td>
<td>1</td>
<td>0</td>
<td>0.44</td>
<td>-0.085</td>
</tr>
</tbody>
</table>

Rosenblatt’s Perceptrons: A worked example

\( \Theta = 0.1, \alpha = 0.1; 13^{\text{th}} \) Iteration

**Weight versus speed of an aircraft**

<table>
<thead>
<tr>
<th>Weight in Tons</th>
<th>Speed in Machs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>0.7</td>
<td>1.5</td>
</tr>
<tr>
<td>0.8</td>
<td>2.0</td>
</tr>
<tr>
<td>0.9</td>
<td>3.0</td>
</tr>
<tr>
<td>1.0</td>
<td>3.5</td>
</tr>
</tbody>
</table>

**Fighters**

**Bombers**
Rosenblatt’s Perceptrons

Rosenblatt’s perceptrons learn in presence of a teacher. The desired signal is denoted as $d_i$ and the output as $y_i$. The error signal is denoted as $e_i$. The weights are modified in accordance with the perceptron learning rule; the weight change is denoted as $\Delta \mathbf{w}$ which is proportional to the error signal; $c$ is a proportionality constant:

$$
e_i \approx d_i - y_i;
$$
$$
y_i = \text{sgn}(\mathbf{w}_i^\mathbf{x}); \text{ and }
$$
$$
\Delta \mathbf{w}_i = c [d_i - \text{sgn}(\mathbf{w}_i^\mathbf{x})] \mathbf{x}
$$
$$
\Delta \mathbf{w}_{ij} = c [d_i - \text{sgn}(\mathbf{w}_i^\mathbf{x})] \mathbf{x}_j
$$

Consider the following set of training vectors $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$, which are to be used in training a Rosenblatt’s perceptron, labelled $j$ with the desired responses $d_1$, $d_2$, and $d_3$, and initial weights $\mathbf{w}_{j1}$, $\mathbf{w}_{j2}$, and $\mathbf{w}_{j3}$.
Rosenblatt’s Perceptrons

The Method:

The Perceptron $j$ has to learn all the three patterns $x_1$, $x_2$, and $x_3$, such that when we show patterns as same as the three or similar patterns the perceptron recognizes them.

How will the perceptron indicate that it has recognised the patterns? By responding as $d_1$, $d_2$, and $d_3$, respectively when shown $x_1$, $x_2$, and $x_3$.

We have to show the patterns repeatedly to the perceptron. At each showing (training cycle) the weights change in an attempt to produce the correct desired response.

Consider the following set of training vectors $x_1$, $x_2$, and $x_3$, which are to be used in training a Rosenblatt’s perceptron together with the weights etc. Show how the learning proceeds:

$$x_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}; \quad x_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}; \quad x_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$$

with the desired responses and weights initialized as:

$$d_1 = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}; \quad d_2 = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}; \quad w = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}; \quad c = 0.1$$
Rosenblatt’s Perceptrons

Learning cycles for the perceptron above. Note the initial weights are selected at random:

\[ w^{1r} = \begin{bmatrix} 1 & -1 & 0 & 0.5 \end{bmatrix} \]

**Step 1.** Input is \( x \), and the desired output is \( d_i \):

\[
net_1 = w^{1r} x_1 = \begin{bmatrix} 1 & -1 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \\ 0 \\ -1 \end{bmatrix} = 2.5
\]

\[ y = \text{sgn}(net_1) = 1 \]

Correction in this step is necessary since \( d_i = -1 \) and the output of the perceptron in response to \( x_i \) is 2.5; the error signal is \( r = d_i - y = -2 \)

---

**Step 1 (contd).** The updated weight vector is

\[
w^2 = w^1 + 0.1 (d_i - y_i) * x_i
\]

\[
w^2 = w^1 + 0.1 (-1 - 1) * x_1
\]

\[
w^2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + 0.1 \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}
\]

\[ w^{2r} = \begin{bmatrix} 0.8 & -0.6 & 0 & 0.7 \end{bmatrix} \]
Step 1 (contd). The updated weight vector is

\[ w^2 = w^1 + 0.1 (-1 - 1) \times x_1 \]

\[ w^2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.4 \\ +0.2 \end{bmatrix} = \begin{bmatrix} 1 - 0.2 \\ -1 + 0.4 \\ 0 \\ 0.5 + 0.2 \end{bmatrix} \]

\[ w^{2t} = [0.8 - 0.6 0 0.7] \]

Correction in this step is not necessary since \( d_2 = -1 \) and the output of the perceptron in response to \( x_2 \) is \( sgn(-1.6) = -1 \); the error signal is \( r \approx -1(-1) = 0 \).
Rosenblatt’s Perceptrons

**Step 3.** Input is $x_3$ and the desired output is $d_3$.

$$w^{3f} = w^{2f} = \begin{bmatrix} 0.8 & -0.6 & 0 \ 0.7 \end{bmatrix}$$

$$\text{net}^3 = w^{3f} x_3 = \begin{bmatrix} -1 & 1 & 0.5 & -1 \end{bmatrix} \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} = -2.1$$

Correction in this step is necessary since $d_3 = +1$ and the output of the perceptron in response to $x_3$ is $\text{sgn}(-2.1) = -1$; the error signal is $r = -1 \cdot (-1) = 2$.

Rosenblatt’s Perceptrons

**Step 3 (contd).** The updated weight vector is

$$w^{4f} = w^{3f} + 0.1 \times (1 + 1) \times x_3$$

$$w^4 = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \ 0.7 \end{bmatrix} + 0.2 \times \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.1 \\ 0.5 \end{bmatrix}$$

$$w^{4f} = \begin{bmatrix} 0.6 & -0.4 & 0.1 & 0.5 \end{bmatrix}$$
Rosenblatt’s Perceptrons

**Step 4.** Input is \( x_i \) and the desired output is \( d_i \):

\[
\begin{align*}
\mathbf{w}^{4t} & = \begin{bmatrix} 0.6 & -0.4 & 0.1 & 0.5 \end{bmatrix} \\
\text{net}^{4t} = \mathbf{w}^{4t} \cdot x_1 & = \begin{bmatrix} 0.6 & -0.4 & 0.1 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = 0.9
\end{align*}
\]

Correction in this step is necessary since \( d_i = -1 \) and the output of the perceptron in response to \( x_3 \) is \( \text{sgn}(0.9) = +1 \); the error signal is \( r = 1 - (1) = -2 \); recall that \( \text{net}_1 \) was \( 2.5 \) and \( r \) was \( -3.5 \).

---

Rosenblatt’s Perceptrons

**Definition of OR**

\[ \begin{array}{c|ccc}
\text{Input} & \text{Output} \\
\hline
X_1 & X_2 & Y \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array} \]

- \( \bigcirc \) Denotes 1
- \( \bigotimes \) Denotes 0

Decision line
Rosenblatt’s Perceptrons

Definition of AND

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>$X_2$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Output: $Y = \begin{cases} 
1 & \text{for } (X_1, X_2) = (1,1) \\
0 & \text{for } (X_1, X_2) = (1,0) \\
0 & \text{for } (X_1, X_2) = (0,0) \\
0 & \text{for } (X_1, X_2) = (0,1) 
\end{cases}$

(0,0) (1,0) (0,1) (1,1)

$\bullet$ Denotes 1
$\bigcirc$ Denotes 0

Linearly separable classes

$C_1$ $C_2$

Linearly non-separable classes

$C_1$ $C_2$

A single layer perceptron can perform a number of logical operations that are performed by other computational devices.

However, the single layer perceptron cannot perform the exclusive-OR or XOR operation. The reason is that a single layer perceptron can only classify two classes, say $C_1$ and $C_2$, should be sufficiently separated from each other to ensure the decision surface consists of a hyperplane.
Rosenblatt’s Perceptrons

• A perceptron computes a binary function of its input. A group of perceptrons can be trained on sample input-output pairs until it learns to compute the correct function.

• Each perceptron, in some model, can function independently of others in the group, they can be separately trained – linearly separable.

• Thresholds can be varied together with weights.

• Given values of $x_1$ and $x_2$ to train such that the perceptron outputs 1 for white dots and 0 for black dots.

Rosenblatt’s Perceptrons

Rosenblatt’s contribution

• What Rosenblatt proved was that if the patterns were drawn from two linearly separable classes, then the perceptron algorithm converges and positions the decision surface in the form of a hyperplane between the two classes → the perceptron convergence theorem (Haykin 117).
Rosenblatt’s Perceptron

The XOR ‘problem’

- The simple perceptron cannot learn a linear decision surface to separate the different outputs, *because no such decision surface exists*.
- Such a non-linear relationship between inputs and outputs as that of an XOR-gate are used to simulate vision systems that can tell whether a line drawing is connected or not, and in separating figure from ground in a picture.

Solution? Employ two separate line-drawing stages.
Rosenblatt’s Perceptrons

Definition of XOR

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>$X_2$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$X_2$ decision line (0,0) to (1,1)

$X_1$ decision line (0,1) to (1,0)

The XOR ‘problem’

One line drawing to separate the pattern where both the inputs are ‘0’ leading to an output ‘0’

Another line drawing to separate the remaining three I/O patterns:

- where either of the inputs is ‘0’ leading to an output ‘1’
- where both the inputs are ‘0’ leading to an output ‘0’
Rosenblatt’s Perceptrons

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>XOR</th>
<th>X2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The XOR ‘solution’

In effect we use two perceptrons to solve the XOR problem: The output of the first perceptron becomes the input of the second.

If the first perceptron sees both inputs as ‘1’ it sends a massive inhibitory signal to the second perceptron causing it to output ‘0’.

If either of the inputs is ‘0’ the second perceptron gets no inhibition from the first perceptron and outputs 1, and outputs ‘1’ if either of the inputs is ‘1’.
Rosenblatt’s Perceptron

**The XOR ‘solution’**
The multilayer perceptron designed to solve the XOR problem has a major defect.

The perceptron convergence theorem does not extend to multilayer perceptrons. The perceptron learning algorithm can adjust the weights between the inputs and outputs, but it cannot adjust weights *between* perceptrons.

For this we have to wait for the back-propagation learning algorithms.

Rosenblatt’s Perceptrons

**Multilayer Perceptron**
The perceptron built around a single neuron is limited to performing pattern classification with only two classes (hypotheses). By expanding the output (computation) layer of perceptron to include more than one neuron, it is possible to perform classification with more than two classes- but the classes have to be separable.
Back-propagation Algorithm: Supervised Learning

Backpropagation (BP) is amongst the ‘most popular algorithms for ANNs’: it has been estimated by Paul Werbos, the person who first worked on the algorithm in the 1970’s, that between 40% and 90% of the real world ANN applications use the BP algorithm. Werbos traces the algorithm to the psychologist Sigmund Freud’s theory of psychodynamics. Werbos applied the algorithm in political forecasting.

• David Rumelhart, Geoffrey Hinton and others applied the BP algorithm in the 1980’s to problems related to supervised learning, particularly pattern recognition.

• The most useful example of the BP algorithm has been in dealing with problems related to prediction and control.

Back-propagation Algorithm

BASIC DEFINITIONS

1. Backpropagation is a procedure for efficiently calculating the derivatives of some output quantity of a non-linear system, with respect to all inputs and parameters of that system, through calculations proceeding backwards from outputs to inputs.

2. Backpropagation is any technique for adapting the weights or parameters of a nonlinear system by somehow using such derivatives or the equivalent.

According to Paul Werbos there is no such thing as a “backpropagation network”, Werbos used an ANN design called a multilayer perceptron.
Paul Werbos provided a ‘rule for updating the weights of a multi-layered network undergoing supervised learning. It is the weight adaptation rule which is called backpropagation.

Typically, a fully connected feedforward network is used to be trained using the BP algorithm: activation in such networks travels in a direction from the input to the output layer and the units in one layer are connected to every other unit in the next layer.

There are two sweeps of the fully connected network: forward sweep and backward sweep.

**Forward Sweep:** This sweep is similar to any other feedforward ANN – the input stimuli is given to the network, the network computes the weighted average from all the input units and then passes the average through a squash function. The ANN generates an output subsequently.

The ANN may have a number of hidden layers, for example, the multi-net perceptron, and the output from each hidden layer becomes the input to the next layer forward.
Back-propagation Algorithm

There are two sweeps of the fully connected network: forward sweep and backward sweep.

**Backwards Sweep:** This sweep is similar to the forward sweep, except what is ‘swept’ are the error values. These values essentially are the differences between the actual output and a desired output. \( e_j = d_j - o_j \)

The ANN may have a number of hidden layers, for example, the multi-net perceptron, and the output from each hidden layer becomes the input to the next layer forward.

In the backward sweep output unit sends errors back to the first proximate hidden layer which in turn passes it onto the next hidden layer. No error signal is sent to the input units.

---

Back-propagation Algorithm

The back-propagation algorithm is used to train a multilayer perceptron by using a set of training examples that are presented to the perceptron \( n \) number of times.

The training examples are usually a pair: an input value and the expected or desired value.

It is expected that after \( n \) iterations the perceptron will learn the relationship between all or many of the input-output value pairs.
**Back-propagation Algorithm**

Now consider the error signal at the output of a neuron $j$ at iteration $n$ - i.e. presentation of the $n^{th}$ training examples:

$$e_j(n) = d_j(n) - y_j(n)$$

where $d_j$ is the desired output and $y_j$ is the actual output and the total error energy overall the neurons in the output layer:

$$E(n) = \frac{1}{2} \sum_{j=1}^{C} e_{j}^2(n)$$

where ‘$C$’ is the number of all the neurons in the output layer.

---

**Back-propagation Algorithm**

If $N$ is the total number of patterns, the averaged squared error energy is:

$$E_{av} = \frac{1}{N} \sum_{n=1}^{N} E(n)$$

Note that $e_j$ is a function of $y_j$ and $y_j$ is a function of $w_{ij}$ (the weights of connections between neurons in two adjacent layers ‘i’ and ‘j’).

Remember that the perceptron is presented the input-output value training vectors $N$ times.
Back-propagation Algorithm

A derivation of the BP algorithm

\[ E(n) \text{ is a function of the error } (e_j = d_j - y_j) \]
\[ e_j(n) \text{ is a function of the output } (y_j) \]
\[ y_j(n) \text{ is a function of the weighted sum } (v_j = \sum_i w_{ij} * x_k) \]
\[ v_j(n) \text{ is a function of the weights } (w_{ij}) \]

\[ E(n) \text{ is a function of a function of a function of } w_{ij}(n) \]

Back-propagation Algorithm

Partial derivative or partial differential coefficient

We wish to compute the changes in the error energy (E) with respect to the changes in the weights of the neurons in the output layer; that is, connection strength between the last hidden layer and the output layer.

Effectively, we have to compute the derivative of \( E \) with respect to \( w_{ij} \). But \( E \) is a function of the error signal \( (e) \), and the error signal, \( e \), is a function of the weighted sum \( (y) \), and \( y \) is a function of the output of the activation function \( (\nu) \).
1. We have to compute the variation of $E$ w.r.t the error signal, whilst keeping its (indirect) variation with $y$ and $\nu$ constant;
2. Similarly we have to compute the variation of the error signal with the $y$ whilst keeping $e$’s variation with the activation function and the weights ($w_{ij}$) constant;
3. The variation of the weighted sum w.r.t with the activation is then computed without taking into account its dependence on the weights;
4. Finally, the variation of the activation w.r.t the weights is computed;
5. The product of the four stages helps to compute the (indirect) variation of $E$ w.r.t the weights.

In the language of differential calculus, or more precisely partial differential calculus, the variation of a function of a number of two or more variables (four in our case, $e, y, \nu, w_{ij}$) is written as a partial derivative:

$$\frac{\partial E}{\partial w_{ij}}$$

In the following 7 slides I will briefly define a derivative and a partial derivative – you may skip the slides if you know about both.
Back-propagation Algorithm

Derivative or differential coefficient

For a function $f(x)$ at the argument $x$, the limit of the differential coefficient as $\Delta x \to 0$

$$\frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Typically defined for a function of a single variable, if the left and right hand limits exist and are equal, it is the gradient of the curve at $x$, and is the limit of the gradient of the chord adjoining the points $(x, f(x))$ and $(x + \Delta x, f(x + \Delta x))$. The function of $x$ defined as this limit for each argument $x$ is the first derivative $y = f'(x)$. 
**Back-propagation Algorithm**

Partial derivative or partial differential coefficient

The derivative of a function of two or more variables with respect to one of these variables, the others being regarded as constant; written as:

\[
\frac{\partial f}{\partial x}
\]

**Back-propagation Algorithm**

Chain rule of calculus

A theorem that may be used in the differentiation of a function of a function

\[
\frac{dy}{dt} = \frac{dy}{dx} \cdot \frac{dx}{dt}
\]

where \( y \) is a differentiable function of \( x \), and \( x \) is a differentiable function of \( t \). This enables a function \( y=f(t) \) to be differentiated by finding a suitable function \( x \), such that \( f \) is a composition of \( y \) and \( x \).
Back-propagation Algorithm

Chain rule of calculus

Similarly for partial differentiation

\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x}
\]

where \( f \) is a function of \( u \) and \( u \) is a function of \( x \).

Back-propagation Algorithm

Total or Exact Differential

The differential of a function of two or more variables with regard to a single parameter in terms of which these variables are expressed, is equal to the sum of the products of each partial derivative of the function with the corresponding increment. If \( z = f(x, y) \), \( x = U(t) \), \( y = V(t) \) then under appropriate conditions, the total differential is:

\[
dz = \frac{\partial z}{\partial x} \, dx + \frac{\partial z}{\partial y} \, dy
\]
Back-propagation Algorithm

A derivation of the BP algorithm

The total error energy $E(n)$ can be computed by summing up the instantaneous energy over all the neurons in the output layer

$$E(n) = \sum_{j \in C} \frac{1}{2} e_j^2(n);$$

where the set $C$ includes all the neurons in the output layer.

The total error energy $E(n)$ can be computed by summing up the instantaneous energy over all the neurons in the output layer

$$E_{av} = \frac{1}{N} \sum_{n=1}^{N} E(n)$$

The instantaneous error energy, $E(n)$ and therefore the average energy $E_{av}$ is a function of the free parameters, including synaptic weights and bias levels.

The originators of neural networks, Widrow and Hoff, suggested the delta learning rule – that the weights should be changed in proportion to the error ($e$) in a given training cycle and in proportion to the input signal ($y$) of a given synapse

$$\Delta w_{ij} = \eta e_i y_j$$

The originators of the BP algorithm suggest that

$$\Delta w_{ij}(n) = -\eta \frac{\partial E}{\partial w_{ij}}$$

where $\eta$ is the learning rate parameter of the BP algorithm. The minus sign indicates the use of gradient descent in the weight; seeking a direction for weight change that reduces the value of $E(n)$.
**Back-propagation Algorithm**

*A derivation of the BP algorithm*

The back-propagation algorithm trains a multilayer perceptron by propagating *back* some measure of *responsibility* to a hidden (non-output) unit.

**Back-propagation:**

- is a local rule for synaptic adjustment;
- takes into account the position of a neuron in a network to indicate how a neuron’s weight are to change.

---

**Back-propagation Algorithm**

*A derivation of the BP algorithm*

Layers in back-propagating multi-layer perceptron

1. First layer – comprises fixed input units;

2. Second (and possibly several subsequent layers) comprise trainable ‘hidden’ units carrying an internal representation.

3. Last layer – comprises the trainable output units of the multi-layer perceptron.
**Back-propagation Algorithm**

*A derivation of the BP algorithm*

Modern back-propagation algorithms are based on a formalism for propagating back the changes in the error energy $E$, with respect to all the weights $w_{ij}$ from a unit ($i$) to its inputs ($j$).

More precisely, what is being computed during the backwards propagation is the rate of change of the error energy $E$ with respect to the networks weight. This computation is also called the computation of the gradient:

$$ \frac{\partial E}{\partial w_{ij}} $$

The error energy $E$ is a function of the error $e$, the output $y$, the weighted sum of all the inputs $v$, and of the weights $w_{ij}$:

$$ \therefore E \equiv E (e, w_{ij}, y, v) $$

According to the chain rule then:

$$ \frac{\partial E (n)}{\partial w_{ij}} = \frac{\partial E (n)}{\partial e_j} \cdot \frac{\partial e_j}{\partial w_{ij}} $$

83

84
Back-propagation Algorithm

A derivation of the BP algorithm

Further applications of the chain rule suggests that:

\[
\frac{\partial E(n)}{\partial w_{ji}} = \frac{\partial E(n)}{\partial e_j} \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial w_{ji}}
\]

\[
\frac{\partial E(n)}{\partial w_{ji}} = \frac{\partial E(n)}{\partial e_j} \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial v_j} \frac{\partial v_j}{\partial w_{ji}}
\]

Back-propagation Algorithm

Each of the partial derivatives can be simplified as:

\[
\frac{\partial E(n)}{\partial e_j} = \frac{\partial}{\partial e_j} \left( \frac{1}{2} \sum_{j \in \mathcal{C}} e_j^2(n) \right) = e_j
\]

\[
\frac{\partial e_j}{\partial y_j} = \frac{\partial}{\partial y_j} \left[ d_j - y_j \right] = -1
\]

\[
\frac{\partial y_j}{\partial v_j} = \frac{\partial}{\partial v_j} \phi(v_j(n)) = \phi'(v_j(n))
\]

\[
\frac{\partial v_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \left[ \sum_i w_{ji}(n)y_i(n) \right] = y_j(n)
\]

\varphi \quad \varphi \text{ is the activation function}
Back-propagation Algorithm

A derivation of the BP algorithm

The rate of change of the error energy ($E$) with respect to changes in the synaptic weights is:

$$\frac{\partial E(n)}{\partial w_{ji}} = -e_j(n) \varphi'(v_j(n))y_i$$

Back-propagation Algorithm

A derivation of the BP algorithm

The back-propagation learning rule is formulated to change the connection weights $w_{ij}$ so as to reduce the error energy $E$ by gradient descent

$$\Delta w_{ij}(n) = w_{ij}^{new}(n) - w_{ij}^{old}(n)$$

$$= -\eta \frac{\partial E(n)}{\partial w_{ij}}$$

$$= \eta e_j(n)\varphi'(v_j(n))y_i$$
Back-propagation Algorithm

A derivation of the BP algorithm

The so-called delta rule suggests that

$$\Delta w_{ij}(n) = -\eta \frac{\partial E(n)}{\partial w_{ij}}$$

$$\downarrow$$

$$\Delta w_{ij}(n) = \eta y_i(n) \delta_j(n)$$

$\delta$ is called the local gradient.
Back-propagation Algorithm

A derivation of the BP algorithm

The local gradient helps in determining the changes in the weights $\Delta w_{ij}$

$$
\delta_j(n) = - \frac{\partial E(n)}{\partial v_j} \left( - \frac{\partial E(n)}{\partial e_j} \cdot \frac{\partial e_j(n)}{\partial v_j} \cdot \frac{\partial y_j(n)}{\partial e_j} \right)
$$

$$
\frac{\partial E(n)}{\partial e_j} = e_j, \quad \frac{\partial e_j(n)}{\partial y_j} = -1, \quad \frac{\partial y_j(n)}{\partial v_j} = \varphi'(v_j(n))
$$

$$
\delta_j(n) = e_j(n) \varphi'(v_j(n))
$$
1. The error signal at the output layer of a multilayer perceptron is generated by taking the difference between the actual output ($y$) and the desired output ($d$) 

$$e_{\text{network}} = d - y_{\text{output layer}}$$

2. The hidden neurons contribute to the error signal of the network, but the hidden neurons are not accessible as is the case with the neurons in the output layer.

There are two points to remember here:

2. The hidden neurons contribute to the error signal of the network, but the hidden neurons are not accessible as is the case with the neurons in the output layer.

- a) What we know is that the error signal for a hidden neuron will have to be calculated in terms of all the neurons to which it is connected.
- b) The hidden neuron may be connected to the neurons in the output layer;
- c) The hidden neuron may be connected to hidden neurons in other layers and ultimately to the neurons in the output layers.
Back-propagation Algorithm

A derivation of the BP algorithm

The case of the output neuron $j$:

$$\Delta w_{ij}(n) = \eta e_j(n) \varphi'(v_j(n))y_i(n)$$

The weight change requires the computation of the local gradient

\[\Delta w_{ij}(n) = \eta e_j(n) \varphi'(v_j(n))y_i(n)\]

Logistic function $\varphi(v_j(n)) = \frac{1}{1 + \exp(-v_j(n))}$

Derivative of $\varphi(v_j(n))$ w.r.t. $v_j(n)$

$$\varphi'(v_j(n)) = \left(\frac{1}{1 + \exp(-v_j(n))}\right)^2 \exp(-v_j(n))$$

$$\varphi'(v_j(n)) = \left(\frac{1}{1 + \exp(-v_j(n))}\right) \left(1 - \frac{1}{1 + \exp(-v_j(n))}\right)$$
Back-propagation Algorithm

A derivation of the BP algorithm

The case of the output neuron $j$: The so-called delta rule suggests that

$$\Delta w_{ij}(n) = \eta e_j(n) \varphi'(v_j(n)) y_i(n);$$

$$y_j(n) = \varphi(v_j(n))$$

$$\therefore \varphi'(v_j(n)) = y_j(n)(1 - y_j(n))$$

and

$$\Delta w_{ij}(n) = \eta e_j(n) y_j(n)(1 - y_j(n)) y_i(n)$$

There is a strong (3rd power) dependence of weight change on the output produced $y$ but this is countered by the presence of $(1-y)$ and a negative linear variation in $y$. 
Back-propagation Algorithm

A derivation of the BP algorithm

The case of the hidden neuron \((j)\) - the delta rule suggests that

\[
\Delta w_{ij}(n) = \eta \delta_j(n) y_i(n)
\]

The index \(j\) refers to the neuron on the right of the hidden neuron \(i\): so, the neuron \(j\) is in the output layer. We will denote the local gradient of the output layer as \(\delta_o\) and the local gradient for the last hidden layer as \(\delta_h\).

Recall that for the output neuron the local gradient comprised the error signal \((e=d-y)\) and the derivative of the activation function. As we do not have an explicit error signal for the hidden layer, so it is not clear how to change the weights. **This problem is solved by using the back-propagation algorithm.**
Back-propagation Algorithm

A derivation of the BP algorithm

The case of the hidden neurons (j):
Recall that the local gradient of the hidden neuron \( h \) is given as \( \delta_h \) and that \( y_h \) is the output and equals \( \varphi(v_h) \)

\[
\delta_h(n) = -\frac{\partial E(n)}{\partial v_h(n)} = -\frac{\partial E(n)}{\partial y_h(n)} \cdot \frac{\partial y_h(n)}{\partial v_h(n)}
\]

\[
\delta_h(n) = -\frac{\partial E(n)}{\partial y_h(n)} \varphi'(v_h(n))
\]

The index \( k \) refers to neuron in the output node and \( j \) refers to the hidden node

The ‘error’ energy related to the hidden neurons is given as

\[
E(n) = \frac{1}{2} \sum_o e_o^2(n)
\]

The rate of change of the error (energy) with respect to the function signal \( y_h(n) \) is given as:

\[
\frac{\partial E(n)}{\partial y_h(n)} = \sum_o e_o \frac{\partial e_o^2(n)}{\partial y_h(n)}
\]
Back-propagation Algorithm

A derivation of the BP algorithm

The case of the hidden neurons \((h)\): The rate of change of the error (energy) with respect to the function signal \(y_h(n)\) is given as:

\[
\frac{\partial E(n)}{\partial y_h(n)} = \sum_o e_o(n) \frac{\partial e_o(n)}{\partial v_o(n)} \frac{\partial v_o(n)}{\partial y_h(n)}
\]

The index \(o\) refers to neuron in the output node and \(h\) refers to the hidden node.
Back-propagation Algorithm

A derivation of the BP algorithm

The case of the hidden neurons \((h)\): The rate of change of the error (energy) with respect to the function signal \(y_h(n)\) is given as:

\[
v_o(n) = \sum_h w_{ho}(n) y_h(n)
\]

\[
\frac{\partial v_o(n)}{\partial y_h(n)} = w_{ho}(n)
\]

\[
\frac{\partial E(n)}{\partial y_h(n)} = -\sum_o e_o(n) \ast \varphi'(v_o(n)) \ast w_{ho}(n)
\]

\[
\frac{\partial E(n)}{\partial y_h(n)} = -\sum_o \delta_o(n) \ast w_{ho}(n)
\]

\[
\delta_h(n) = -\frac{\partial E(n)}{\partial y_h(n)} \varphi'(v_h(n))
\]

\[
\frac{\partial E(n)}{\partial y_h(n)} = -\sum_o \delta_o(n) \ast w_{ho}(n)
\]

\[
\therefore \delta_h(n) = \varphi'(v_h(n)) \ast \sum_o \delta_o(n) \ast w_{ho}(n)
\]

Weight Correction= (learning rate parameter) * local gradient * input signal of neuron \(h\)

\[
\Delta w_{ho} = \eta \delta \varphi'
\]
### Back-propagation Algorithm

**A derivation of the BP algorithm**

The case of the hidden neurons (j): Recall that the local gradient of the hidden neuron $j$ is given as $\delta_j$ and that $y_j$ is the output and equals

$$\delta_j(n) = \varphi'(v_j(n)) \sum_o \delta_o(n) \cdot w_{ho}(n)$$

The index $o$ refers to all the neurons on the right of the hidden neuron $h$: so, for example, the neurons $(h)$ in the hidden layer connected directly to the output neurons in the output layer, the index $o$ will just have one value and $\delta_o$ here will refer to all the neurons in the output layer. The local gradient for the last hidden layer is $\delta_h$.

\[
\delta_h(n) = \varphi'(v_h(n)) \sum_o \delta_o(n) \cdot w_{ho}(n)
\]
Substituting the equation for the gradient of the last hidden neuron $\delta_h$ into the one for hidden neuron in the last-but-one layer $\delta_{h'}$, a recursive combination

$$
\delta_{h'}(n) = \varphi'(v_{h'}(n)) \sum_h \left( \varphi'(v_h(n)) \sum_o \delta_o(n) * w_{ho}(n) \right) * w_{h'h}(n)
$$

Now for the third hidden layer, denoted as $h''$, we perform another recursion that will take us from all the way from the output layer neurons ($o$) to third hidden layer:

$$
\delta_{h''}(n) = \varphi'(v_{h''}(n)) * \sum_{h'} \left( \varphi'(v_{h'}(n)) \sum_h \delta_h(n) * w_{h'ho}(n) \right) * w_{h'h''}(n)
$$

$$
\delta_{h''}(n) = \varphi'(v_{h''}(n)) \sum_{h'} \left( \varphi'(v_{h'}(n)) \sum_h \varphi'(v_h(n)) \sum_o \varphi'(v_o(n)) * e_o(n) * w_{ho}(n) \right) * w_{h'h''}(n) * w_{h'h''}(n)
$$
The back-propagation algorithm for updating weights of all the neurons in the neural network suggests that:

\[
\Delta w_{ji}(n) = \eta \delta_j(n) y_j(n)
\]

The output, \( o_j \), of a neuron \( j \) from stimulus received from \( n \) neurons, through the vector \( x_i \) and the bias \( w_{0j} \) is:

\[
\begin{align*}
\text{Step 1a: } & v_j = w_{0j} + \sum_{i=1}^{n} w_{ij} * x_i \\
\text{Step 1b: } & y_j = \frac{1}{1 + e^{-v_j}}
\end{align*}
\]
### Back-propagation Algorithm: A 4-step program

**Step 2**: \( \delta_{j'} = (1 - y_{j'}) \cdot y_{j'} \cdot \sum_k w_{kj'} \delta_k \)

The error, \( \delta_{j'} \), of a hidden neuron \( j' \) from stimulus received from the previous layers, \( k \), of a multi-layer system, with strength \( w_{kj'} \), is:

**Step 3**: \( \delta_j = (1 - y_j) \cdot y_j \cdot (d_j - y_j) \)

The error, \( \delta_j \), of an output neuron \( j \) from stimulus, \( o_j \), received from the last hidden layer, is given in terms of the desired vector \( d_j \) is:

**Step 4**: \( \Delta w_{kj} = \eta \cdot \delta_j \cdot x_k \)

Recall that the weight change, \( \Delta w_{kj} \), is computed as
Consider a 2x2x1 network:

First Layer Connectivity

\[
\begin{bmatrix}
w_{13} & w_{23} \\
w_{14} & w_{24}
\end{bmatrix} = \begin{bmatrix}
-0.2 & 0.1 \\
-0.1 & 0.3
\end{bmatrix}
\]

Bias weight

\[
\begin{bmatrix}
w_{31} & w_{32} & w_{34}\end{bmatrix} = \begin{bmatrix}
0.1 & 0.1 & 0.2
\end{bmatrix}
\]

Second Layer Connectivity

\[
\begin{bmatrix}
w_{35} & w_{45}
\end{bmatrix} = \begin{bmatrix}
0.2 & 0.3
\end{bmatrix}
\]

The desired vector \( d = 0.9 \)

---

Consider an input vector \( x \):

\[
\begin{bmatrix}
x_1 & x_2
\end{bmatrix} = \begin{bmatrix}
0.1 & 0.9
\end{bmatrix}
\]
Back-propagation Algorithm: Worked Example #1

Consider a 2x2x1 network:

**Forward Pass**

\[ v_3 = w_{03} + \sum_{i=1}^{2} x_i w_{i,3} \]

\[ = 0.1 + (0.1 \times (-0.2)) + 0.9 \times 0.1 \]

\[ = 0.170 \]

\[ y_3 = \frac{1}{1 + e^{-v_3}} = \frac{1}{1 + e^{-0.170}} = 0.542 \]

---

**Forward Pass**

\[ v_4 = w_{04} + \sum_{i=1}^{2} x_i w_{i,4} \]

\[ = 0.1 + (0.1 \times (-0.1)) + 0.9 \times 0.3 \]

\[ = 0.36 \]

\[ y_4 = \frac{1}{1 + e^{-v_4}} = \frac{1}{1 + e^{-0.36}} = 0.589 \]
Back-propagation Algorithm: Worked Example #1

Consider a 2x2x1 network:

**Forward Pass**

\[ V_5 = w_{05} + \sum_{i=3}^{4} x_i w_{i,5} \]
\[ = 0.2 + 0.542 \times 0.2 + 0.589 \times 0.3 \]
\[ = 0.485 \]
\[ y_5 = \frac{1}{1 + e^{-V_5}} = \frac{1}{1 + e^{-0.485}} = 0.619 \]

**BACKWARD PASS:**

Calculating the local gradient at the output layer
\[ \delta_5 = (d - y_5) \times y_5 \times (1 - y_5) \]
\[ = (0.9 - 0.619) \times 0.619 \times (1 - 0.619) \]
\[ = 0.066 \]
Back-propagation Algorithm: Worked Example #1

Consider a 2x2x1 network:

**Backward Pass**

**Gradients at the hidden layer**

\[
\delta_h(n) = \varphi'(v_h(n)) \sum_o \delta_o(n) \ast w_{ho}(n)
\]

There is only one output layer:

\[
\delta_o(n) = \varphi'(v_h(n)) \ast \delta_o(n) \ast w_{ho}(n)
\]

For the special case of sigmoid squash function

\[
\varphi(v_h(n)) = \frac{1}{1 + e^{-v_h}}
\]

\[
\varphi'(v_h(n)) = \frac{\partial \varphi(v_h(n))}{\partial v_h(n)}
\]

\[
\vdash \frac{\partial \varphi(v_h(n))}{\partial v_h(n)} = \varphi(v_h(n)) \ast (1 - \varphi(v_h(n)))
\]

\[
\vdash \delta_h(n) = \varphi(v_h(n)) \ast (1 - \varphi(v_h(n))) \ast \delta_o(n) \ast w_{ho}(n)
\]

\[
\delta_h(n) = y_h(n) \ast (1 - y_h(n)) \ast \delta_o(n) \ast w_{ho}(n)
\]

---

**Backward Pass**

**Gradients at the hidden neuron 4**

\[
\delta_4(n) = y_4(n) \ast (1 - y_4(n)) \ast \delta_5(n) \ast w_{4,5}(n)
\]

\[
\delta_4(n) = 0.589 \ast (1 - 0.589) \ast 0.066 \ast 0.3
\]

\[
\delta_4(n) = 0.005
\]
Back-propagation Algorithm: Worked Example #1

Consider a 2x2x1 network:

\[ \delta_3(n) = y_3(n) \cdot (1 - y_3(n)) \cdot \delta_5(n) \cdot w_{3,5}(n) \]
\[ \delta_3(n) = 0.542 \cdot (1 - 0.542) \cdot 0.066 \cdot 0.2 \]
\[ \delta_3(n) = 0.003 \]

Backward Pass Gradients at the hidden neuron 3

**Weight update:**

\[ \Delta w_{i,j} = \eta \cdot \delta_j \cdot y_i \]
\[ w_{i,j}^{new} = w_{i,j}^{old} + \Delta w_{i,j} \]
\[ = w_{i,j}^{old} + \eta \cdot \delta_j \cdot y_i \]
Consider a 2x2x1 network:

**Weight update**: 
\[ \Delta w_{i,j} = \eta \cdot \delta_j \cdot y_i \]
\[ w_{new}^{new} = w_{old}^{old} + \Delta w_{4,5} \]
\[ w_{4,5}^{new} = w_{4,5}^{old} + \eta \cdot \delta_5 \cdot y_4 \text{ (let } \eta = 0.25) \]
\[ w_{4,5}^{new} = 0.3 + 0.25 \cdot 0.066 \cdot 0.589 \]
\[ w_{4,5}^{new} = 0.3 + 0.01 = 0.31 \]
Back-propagation Algorithm: Worked Example #1

Example: Perform a complete forward and backward sweep of a 2-2-1 (2 input units, 2 hidden layer units and 1 output unit) with the following architecture. The target output $d=0.9$. The input is $[0.1 \ 0.9]$.

Example of a forward pass and a backward pass through a 2-2-1 feedforward network.

Inputs, outputs and errors are shown in boxes.

Target value is 0.9 so error for the output unit is:

$((0.900 - 0.288) \times 0.288) \times (1 - 0.288) = 0.125$
Back-propagation Algorithm: Worked Example #2

New weights calculated following the errors derived above

ANFIS: Computations for a Takagi-Sugeno System

The example given in Negnevitsky relates to the ‘logical’ operation 1-XOR rather than XOR (Fig 8.7, pg 272):
ANFIS: Computations for a Takagi-Sugeno System

The example given in Negnevitsky relates to the ‘logical’ operation 1-XOR rather than XOR.

Also missing is the ‘rule’:

If \( x_1 \) is \( L \) and \( x_2 \) is \( L \) Then \( y \) is \( L \)

Although the absence of rule may not matter as it may have either been misread into the rule base by an expert or induced by an expert as

If \( x_2 \) is \( L \) Then \( y \) is \( L \)

Remember that a Takagi-Sugeno System is expressed as

\[
\text{IF } x \text{ is } A \text{ and } y \text{ is } B \text{ THEN } z = f(x,y) \\
\text{IF } x \text{ is } A \text{ and } y \text{ is } B \text{ THEN } z = px + qy + r
\]

Note that the output function \( z \) is LINEAR but the membership functions are invariably non-linear being, for example, of the form

\[
A(x) = \frac{1}{1+((x-a)/c)^{2b}}
\]

where \( a, b, \) and \( c \) are parameters to be ‘learnt’
### ANFIS: Computations for a Takagi-Sugeno System

<table>
<thead>
<tr>
<th></th>
<th>Forward Pass</th>
<th>Backward Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters: Premise</td>
<td>No change</td>
<td>Gradient Descent</td>
</tr>
<tr>
<td>Parameters: Consequent</td>
<td>Least Square Estimator</td>
<td>No change</td>
</tr>
<tr>
<td>Compute</td>
<td>Node Outputs</td>
<td>Error Signals</td>
</tr>
</tbody>
</table>
Fuzzy If-Then Rules

- **Mamdani style**
  If pressure is high then volume is small

- **Sugeno style**
  If speed is medium then resistance = 5*speed

Fuzzy Inference System (FIS)

- If speed is low then resistance = 2
- If speed is medium then resistance = 4*speed
- If speed is high then resistance = 8*speed

**Rules**

- Rule 1: \( w_1 = .3; \ r_1 = 2 \)
- Rule 2: \( w_2 = .8; \ r_2 = 4*2 \)
- Rule 3: \( w_3 = .1; \ r_3 = 8*2 \)

**Resistance**

\[
\text{Resistance} = \frac{\sum (w_i \cdot r_i)}{\sum w_i} = 7.12
\]
ANFIS: Mamdani’s model

- Layer 1: input layer
- Layer 2: input membership or fuzzification layer
  - Neurons represent fuzzy sets used in the antecedents of fuzzy rules determine the membership degree of the input.
  - Activation fn: the membership fn.
- Layer 3: fuzzy rule layer
  - Each neuron corresponds to a single fuzzy rule.
  - Conjunction of the rule antecedents:
  - Output: the firing strength of fuzzy rule $R_i$: $\mu_{R_i} = \mu_{A_j} \times \mu_{B_k}$
  - The weights b/t layer 3 and layer 4: the normalized degree (k.a. certainty factors) of confidence of the corresponding fuzzy rules. They’re adjusted during training.
- Layer 4: Output membership layer
  - Disjunction of the outputs: $\mu_{R_i} = \mu_{A_j} \oplus \mu_{B_k} = \sum \mu_{R_i}$
  - the integrated firing strength of fuzzy rule neurons $R_j$ and $R_k$.
  - Activation fn: the output membership fn.
- Layer 5: defuzzification layer
  - Each neuron represents a single output.
  - E.g.) centroid method.

Learning
- A various learning algorithm may be applied: Back propagation
- Adjustment of weights and modification of input/output membership functions.
- Sum-Product composition and centroid defuzzification was adopted, a corresponding ANFIS was constructed easily.
- Extra complexity with max-min composition – no better learning capability or approximation power.
- More complicated than Sugeno ANFIS or Tsukamoto ANFIS
ANFIS: Mamdani’s model

First-Order Sugeno FIS

- Rule base
  If X is A and Y is B₁ then Z₁ = p₁₁x₁ + q₁₁y₁ + r₁
  If X is A and Y is B₂ then Z₂ = p₂₁x₁ + q₂₁y₁ + r₂

- Fuzzy reasoning

\[
Z = \frac{w₁Z₁ + w₂Z₂}{w₁ + w₂}
\]
Fuzzy Inference Systems (FIS)

Also known as
- Fuzzy models
- Fuzzy associate memories (FAM)
- Fuzzy controllers

Rule base (Fuzzy rules)  Data base (MFs)

Fuzzy reasoning

input  output

Adaptive Networks

Architecture:
- Feedforward networks with diff. node functions
- Squares: nodes with parameters
- Circles: nodes without parameters

Goal:
- To achieve an I/O mapping specified by training data

Basic training method:
- Backpropagation or steepest descent
Derivative-Based Optimization

Based on first derivatives:
- Steepest descent
- Conjugate gradient method
- Gauss-Newton method
- Levenberg-Marquardt method
- And many others

Based on second derivatives:
- Newton method
- And many others

Fuzzy Modeling

• Given desired i/o pairs (training data set) of the form \((x_1, ..., x_n; y)\), construct a FIS to match the i/o pairs

• Two steps in fuzzy modeling
  structure identification --- input selection, MF numbers
  parameter identification --- optimal parameters
Neuro-Fuzzy Modeling

Basic approach of ANFIS

- Adaptive networks
- Neural networks
- Fuzzy inference systems

ANFIS (Adaptive Neuro-Fuzzy Inference System)

Fuzzy reasoning:

\[ z_1 = p_1x + q_1y + r_1 \]

\[ z_2 = p_2x + q_2y + r_2 \]

\[ z = \frac{w_1z_1 + w_2z_2}{w_1 + w_2} \]

ANFIS (Adaptive Neuro-Fuzzy Inference System)

\[ x \quad y \]

\[ A_1 \quad A_2 \quad B_1 \quad B_2 \]

\[ w_1 \quad w_2 \]

\[ \Sigma w_i = \Sigma \]
**Four-Rule ANFIS**

- Input space partitioning

**ANFIS (Adaptive Neuro-Fuzzy Inference System)**

\[ x \begin{array}{c} A_1 \\ A_2 \end{array} \quad y \begin{array}{c} B_1 \\ B_2 \end{array} \]

\[ \sum w_i^r z_i \]

\[ \sum w_i z_i \]

**Neuro Fuzzy System**

- A neuro fuzzy system is capable of identifying bad rules in prior/existing knowledge supplied by a domain expert.
- e.g.) 5-rule neuro-fuzzy system for XOR operation.
  - Use back propagation to adjust the weights and to modify input-output membership fns.
  - Continue training until the error (e.g. sum of least mean square) is less than e.g.) 0.001
  - Rule 2 is false and removed.
Neuro Fuzzy System

- A neuro fuzzy system which can automatically generate a complete set of fuzzy if-then rules, given input-output linguistic values.
- Extract fuzzy rules directly from numerical data.
- e.g.) 8 rule neuro fuzzy system for XOR operation: \(2^2 \times 2 = 8\) rules.
  - Set the initial weights b/t layer 3-4 to 0.5.
  - After training, eliminate all rules whose certainty factors are less than some sufficiently small number, e.g. 0.1.
Neuro Fuzzy System

- A neuro fuzzy system which can automatically generate a complete set of fuzzy if-then rules, given input-output linguistic values.
- Extract fuzzy rules directly from numerical data.
- E.g.) 8 rule neuro fuzzy system for XOR operation: \(2^2 \times 2 = 8\) rules.
- Set the initial weights b/t layer 3-4 to 0.5.
- After training, eliminate all rules whose certainty factors are less than some sufficiently small number, e.g. 0.1.

ANFIS Architecture: Sugeno’s ANFIS

- Assume that FIS has x inputs \(x, y\) and one output \(z\).

Sugeno’s ANFIS:

- Rule1: If \(x\) is \(A_1\) and \(y\) is \(B_1\), then \(f_1 = p_1x + q_1y + r_1\).
- Rule2: If \(x\) is \(A_2\) and \(y\) is \(B_2\), then \(f_2 = p_2x + q_2y + r_2\).
ANFIS Architecture: Sugeno’s ANFIS

- **Layer 1:** fuzzification layer
  - Every node \( i \) in the layer 1 is an adaptive node with a node function
  - \( O_{1,i} = \mu_{A_i}(x) \) for \( i=1,2 \) or membership grade of a fuzzy set \( A_1,A_2 \)
  - \( O_{1,i} = \mu_{A_3}(y) \) for \( i=3,4 \)
  - Parameters in this layer:

- **Layer 2:** rule layer
  - a fixed node labeled \( P \) whose output is the product of all the incoming signals:
  - \( O_{2,i} = \mu_{A_i}(x) \mu_{B_i}(y) \) for \( i=1,2 \)

- **Layer 3:** normalization layer
  - a fixed node labeled \( N \).
  - The i-th node calculates the ratio of the i-th rule’s firing strength to the sum of all rules’ firing strength: \( O_{3,i} = \frac{w_i}{\sum w_j} \) for \( i=1,2 \)
  - Outputs of this layer are called \( \mu_i \).

- **Layer 4:** defuzzification layer
  - an adaptive node with a node fn \( O_{4,i} = f_i = (p_i x + q_i y + r_i) \) for \( i=1,2 \)
  - where \( w_i \) is a normalized firing strength from layer 3 and \( \{p_i, q_i, r_i\} \) is the parameter set of this node

- **Layer 5:** summation neuron
  - a fixed node which computes the overall output as the summation of all incoming signals
  - Overall output = \( O_{5,1} = \sum f_i = \sum w_i f_i / \sum w_i \)

---

**How does an ANFIS learn?**

- A learning algorithm of the least-squares estimator + the gradient descent method

**Forward Pass:** adjustment of consequent parameters, \( p_i, q_i, r_i \)

- Rule consequent parameters are identified by the least-square estimator,
- Find a least-square estimate of \( k=[r1 \ p1 \ q1 \ .. \ r2 \ p2 \ q2 \ .. \ rn \ pn \qn] \) that minimizes the squared error
- \( E = e^2 / 2 = (O_d - O)^2 / 2 \)
- The consequent parameters are adjusted while the antecedent parameters remain fixed.

**Backward Pass:** adjustment of antecedent parameters

- The antecedent parameters are tuned while the consequent parameters are kept fixed.
- E.g.) Bell activation fn: \( [1 + ((x-a)/c)^b]^{-1} \).
  - Consider a correction applied to parameter of \( a \), \( \Delta a \), \( a = a + \Delta a \)
  - where \( \Delta a \).
ANFIS Architecture: Sugeno’s ANFIS

- How does an ANFIS learn?
  - A learning algorithm of the least-squares estimator + the gradient descent method
  - **Forward Pass:** adjustment of *consequent parameters*, \( p_i, q_i, r_i \).
    - Rule consequent parameters are identified by the least-square estimator.
    - Find a least-square estimate of \( k = [r_1 p_1 q_1, r_2 p_2 q_2, \ldots, r_n p_n q_n] \),
      that minimizes the squared error
    - \( E = e^2 / 2 = (O_d - O)^2 / 2 \)
    - The consequent parameters are adjusted while the antecedent parameters remain fixed.
  - **Backward Pass:** adjustment of *antecedent parameters*
    - The antecedent parameters are tuned while the consequent parameters are kept fixed.
    - E.g.) Bell activation fn: \( 1 + ((x-a)/c)^b \)^{-1}.
      Consider a correction applied to parameter of \( a, \Delta a, a = a + \Delta a \).

The structure of the network is not unique.

**Figure 12.2.** ANFIS architecture for the Sugeno fuzzy model, where weight normalization is performed at the very last layer.
ANFIS Architecture: Tsukamoto ANFIS

Tsukamoto ANFIS:

Figure 12.3. (a) A two-input two-rule Tsukamoto fuzzy model; (b) equivalent ANFIS architecture.

ANFIS Architecture

- Improvement: 2 input first-order Sugeno fuzzy model with 9 rules
- How the 2-dimensional input space is partitioned into 9 overlapping fuzzy regions, each of which is governed by a fuzzy if-then rule.

Figure 12.4. (a) ANFIS architecture for a two-input Sugeno fuzzy model with nine rules; (b) the input space that are partitioned into nine fuzzy regions.