A node can be characterised as follows:

- Input connections representing the flow of activation from other nodes or some external source
- Each input connection has its own weight, which determines how much influence that input has on the node
- A node i has an output activation \( a_i = f(\text{net}_i) \) which is a function of the weighted sum of its input activations, net.
- The net input is determined as follows: \( \text{net}_i = \sum_j w_{ij} a_j \)
Calculating the activation: \( net_i \) is 1.25

- Linear activation:
  \[
  f : \mathbb{R} \rightarrow \mathbb{R}
  \]
  \[
  f(1.25) = 1.25
  \]
- Linear threshold: \( T = 0.5 \)
  \[
  f : \mathbb{R} \rightarrow \mathbb{R}
  \]
  \[
  f(1.25) = 1.25 - 0.5 = 0.75
  \]
- Binary threshold: \( T = 0.5 \)
  \[
  f : \mathbb{R} \rightarrow [0,1]
  \]
  \[
  \] \[
  f(1.25) = 1
  \]
- Nonlinear activation:
  - Sigmoid or "logistic" function
    \[
    f : \mathbb{R} \rightarrow [0,1]
    \]
    \[
    f(1.25) = \frac{1}{1 + e^{-net_i}}
    \]
    \[
    f(1.25) = 0.777
    \]

Summary of network architecture

- The activation of a unit \( i \) is represented by the symbol \( a_i \).
- The extent to which unit \( j \) influences unit \( i \) is determined by the weight \( w_{ij} \).
- The input from unit \( j \) to unit \( i \) is the product: \( a_j \cdot w_{ij} \).
- For a node \( i \) in the network:
  \[
  net_i = \sum_j w_{ij} a_j
  \]
  The output activation of node \( i \) is determined by the activation function, e.g. the logistic:
  \[
  a_i = f(net_i) = \frac{1}{1 + e^{-net_i}}
  \]
Learning in connectionist networks

- **Supervised learning** in connectionist networks involves successively adjusting connection weights to **reduce the discrepancy** between the actual output activation and the correct output activation.
  - An input is presented to the network.
  - Activations are propagated through the network to its output.
  - Outputs are compared to “correct” outputs: difference is called error.
  - Weights are adjusted.

The Delta Rule

\[ \Delta w_{ij} = [a_i(\text{desired}) - a_i(\text{obtained})]a_j \varepsilon \]

- \([a(\text{desired}) - a(\text{obtained})]\) is the difference between the desired output activation and the actual activation produced by the network.
- What is the “error”?
- \(a_j\) is the activity of the contributing unit \(j\).
- How much activation is this unit responsible for?
- \(\varepsilon\) is the learning rate parameter.
- How rapidly do we want to make changes?
Training the Network

Consider the **AND** function

- Present stimulus, e.g.: 0 0
- Compute output activation
- Compared with desired output (0)
- Use Delta rule to change weights
- Repeat for all input-output pairings

An **Epoch**, consists of a single presentation of all training instances

- Here there are 4 such input-output pairings

A **Sweep**, is a presentation of a single training instance

- So, 250 epochs consists of 1000 sweeps

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \Delta w_{ij} = [a_i(\text{desired}) - a_i(\text{obtained})]a_j \varepsilon \]

"Perceptrons" [Rosenblatt 1958]

- Perceptron: a simple, one-layer, feed-forward network:

\[ \text{netinput}_{out} = \sum_{in} w \cdot a_{in} \]

\[ a_{out} = 1 \text{ if } \text{netinput}_{out} > \theta \]
\[ = 0 \text{ otherwise} \]

The error, \( \delta = (t_{out} - a_{out}) \)
\[ \Delta \theta = -\varepsilon \delta \]
\[ \Delta w = \varepsilon \delta a_{in} \]

Connectionist Language Processing – Crocker & Brouwer
• Consider the following simple perceptron:
  • Recall the convergence rule:

    The error, \( \delta = (t_{out} - a_{out}) \)
    \[ \Delta \theta = -\varepsilon \delta \]
    \[ \Delta w = \varepsilon \delta a_{in} \]

• We want to train this to learn boolean OR:
  • Note: changes have opposite signs
    • E.g if activity is less than target, \( \delta \) is positive:  
      "Threshold is decreased; Weight is increased"
  • If \( \delta \) is non-zero, threshold is always changed
    • But if \( a_{in} \) is zero, the weight is not changed
  • The changes can be calculated straight-forwardly, but do they lead to convergence on a solution to a problem?

---

**Learning OR continued …**

The error, \( \delta = (t_{out} - a_{out}) \)
\[ \Delta \theta = -\varepsilon \delta \]
\[ \Delta w = \varepsilon \delta a_{in} \]

<table>
<thead>
<tr>
<th>( a_0 )</th>
<th>( a_1 )</th>
<th>( a_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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( t_1 )</th>
<th>( t_2 )</th>
<th>( \delta )</th>
<th>( \Delta \theta )</th>
<th>( \Delta w_{20} )</th>
<th>( \Delta w_{21} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\( \theta = 1 \)
\( \varepsilon = 0.5 \)

\( w_{20} = 0.2 \)
\( w_{21} = 0.1 \)
Gradient descent

- Let's define the error on the outputs as: \( E_p = (t_{out} - a_{out})^2 \)
  - Recall: \( a_{out} = \sum w a_{in} \)
  - This means \( E_p \) is always positive

- For a single layer net, if we consider one weight, holding the others constant:
  - Plot Error versus varying the weight

- The lowest point on the curve, represents the minimum error possible for:
  - For pattern \( p \)
  - By varying a given weight \( w \)

- Learning: the network is always at some point on the error curve
  - Use the slope of the curve to change the weights in the right direction
  - If slope is positive, then decrease the weight
  - If slope is negative, increase the weight

Visualising the error
Gradient descent continued

- We need calculus to allow us to determine how the error varies when a particular weight is varied:

\[
\Delta w = -\varepsilon \frac{\partial E}{\partial w}
\]

\[
\Delta w = -\varepsilon \frac{\partial (t_{out} - a_{out})^2}{\partial w}
\]

\[
\Delta w = -\varepsilon \frac{\partial [t_{out} - F(\sum_i w \cdot a_{in})]^2}{\partial w}
\]

\[
\Delta w = 2\varepsilon [t_{out} - F(\sum_i w \cdot a_{in})] \cdot F'(\sum_i w \cdot a_{in}) \cdot a_{in}
\]

\[
\Delta w = 2\varepsilon F^* a_{in}
\]

\[\delta = (t_{out} - a_{out})\]

\[F^* = \text{slope of the activation function}\]

Gradient descent and the delta rule

- The perceptron convergence rule: \(\Delta w = \varepsilon \delta a_{in}\)

- Our revised learning rule, based on gradient descent is: \(\Delta w = 2\varepsilon F^* a_{in}\)
  - where \(F^*\) is the slope of the activation function

- If the activation function is linear, it's slope is constant:
  - where \(k\) is a constant representing the learning rate and slope

- This corresponds to the original Delta rule: \(\Delta w = k\delta a_{in}\)
  - It is straightforward to calculate
  - Performs gradient descent to the bottom of the error curve
  - \(\Delta w\) is proportional to \((t_{out} - a_{out})\), so changes get smaller as error is reduced
  - In 2-layer networks, there is a single minimum: gradient descent learning is therefore guaranteed to find a solution, if one exists.
Learning with the Sigmoid activation function

- Networks with linear activation functions:
  - have mathematically well-defined learning capacities
  - they are known to be limited in the kinds of problems they can solve

- The logistic, or sigmoid, function is:
  \[ a_i = f(net_i) = \frac{1}{1 + e^{-net_i}} \]
  - Non-linear, more powerful
  - More neurologically plausible
  - Less well-understood, more difficult to analyse mathematically

Behaviour of the logistic function

- Deriving the slope of the logistic function:
  \[ a_i = f(net_i) = \frac{1}{1 + e^{-net_i}} \]
  \[ F* = f'(net_i) = a_{out}(1 - a_{out}) \]

- The Delta rule, assuming the logistic function:
  \[ \Delta w = 2\varepsilon \delta F * a_{in} \]
  or
  \[ \Delta w = 2\varepsilon (t_{out} - a_{out})a_{out}(1 - a_{out})a_{in} \]
Training a network

- The training phase involves
  - Presenting an input pattern, and computing the output for the network using the current connection weights: \( a_{out} = f(\sum in \ w_{out,in} \times a_{in}) \)
  - Calculating the error between the desired and the actual output \( (t_{out} - a_{out}) \)
  - Using the Delta rule (appropriate for the activation function):
    \[
    \Delta w = \eta (t_{out} - a_{out}) a_{out} (1 - a_{out}) a_{in}
    \]
  - One such cycle is called a sweep, and a sweep through each pattern is called an epoch.

- We can define the global error of the network, as the average error across all input patterns, \( k \):
  - One common measure is the square root of mean error
  - Squaring avoids positive and negative error cancelling each other out

Training: an example

- Assume an input pattern: 1 1
- Assume a learning rate of 0.1
- Assume a sigmoid activation
- Desired output is: 1
- Determine the weight changes for 1 sweep:
  \[
  a_2 = f(1 \times 0.75 + 1 \times 0.5) = 0.77
  \]
  \[
  \delta_2 = (t - a_2) f'(0.77) = 0.23 \times 0.16 = 0.037
  \]
  \[
  \Delta w_{20} = \eta \delta_2 a_0 = 0.1 \times 0.037 \times 1 = 0.0037
  \]
  \[
  \Delta w_{21} = \eta \delta_2 a_1 = 0.1 \times 0.037 \times 1 = 0.0037
  \]

\[
\text{rms error} = \sqrt{\frac{\sum (t_k - o_k)^2}{k}}
\]
The dynamics of weight changes

- Learning rate: predetermined constant (though can be changed during training)
- The error: large error = large weight change
- The slope of the activation function:
  - The derivative of the logistic is largest for netinputs around 0, and for activations around .5
  - Small netinputs co-occur with small weights
  - Small weights tend to occur early in training
  - The result: bigger changes during early stages of learning
    - More resilience in older network: harder to teach new tricks!
- The momentum: This parameter determines how much of the previous weight change affects the current weight change

Calculating Error

- Consider a simple network for learning the AND operation
- After training (1000 sweeps, 250 epochs), we can calculate the global (RMS) error as follows:

<table>
<thead>
<tr>
<th>Input</th>
<th>Target</th>
<th>Output</th>
<th>(t-o)^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>0.147</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>0</td>
<td>0.297</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>0</td>
<td>0.334</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>0</td>
<td>0.552</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RMS:</td>
</tr>
</tbody>
</table>

  \[
  \text{rms error} = \sqrt{\frac{\sum_{k} (t_k - o_k)^2}{k}}
  \]
Calculating Global RMS Error

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Observed Output</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.321 0.196 0.255 0.264</td>
<td>1.000 0.000 0.000 0.000</td>
</tr>
<tr>
<td>2</td>
<td>0.227 0.612 0.169 0.211</td>
<td>0.000 1.000 0.000 0.000</td>
</tr>
<tr>
<td>3</td>
<td>0.287 0.188 0.342 0.276</td>
<td>0.000 0.000 1.000 0.000</td>
</tr>
<tr>
<td>4</td>
<td>0.296 0.207 0.300 0.268</td>
<td>0.000 0.000 0.000 1.000</td>
</tr>
</tbody>
</table>

**Error (t-o)**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.679</td>
<td>-0.196</td>
<td>-0.255</td>
<td>-0.264</td>
<td></td>
</tr>
<tr>
<td>-0.227</td>
<td>0.388</td>
<td>-0.169</td>
<td>-0.211</td>
<td></td>
</tr>
<tr>
<td>-0.287</td>
<td>-0.188</td>
<td>0.658</td>
<td>-0.276</td>
<td></td>
</tr>
<tr>
<td>-0.296</td>
<td>-0.207</td>
<td>-0.3</td>
<td>0.732</td>
<td></td>
</tr>
</tbody>
</table>

**Error^2**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.461041</td>
<td>0.038416</td>
<td>0.065025</td>
<td>0.069696</td>
<td>0.634178</td>
</tr>
<tr>
<td>0.051529</td>
<td>0.150544</td>
<td>0.028561</td>
<td>0.044521</td>
<td>0.275155</td>
</tr>
<tr>
<td>0.082369</td>
<td>0.035344</td>
<td>0.432964</td>
<td>0.076176</td>
<td>0.626853</td>
</tr>
<tr>
<td>0.087616</td>
<td>0.042849</td>
<td>0.09</td>
<td>0.535824</td>
<td>0.756289</td>
</tr>
</tbody>
</table>

RMS Error: 0.757046

\[
\text{RMS error} = \sqrt{\frac{\sum_{k} (t_k - o_k)^2}{k}}
\]

**Summary – Learning Rules**

- Perceptron convergence rule
- Delta rule
  - Depends on the (slope of the) activation function
- For 2-layer networks using these rules:
  - A solution will be found, if it exists
- How do we know if network has learned successfully?
Summary – Error

• For learning, we use \((t_{out} - a_{out})\) for each output unit, to change weights

• To characterise the performance of the network as a whole, we need a measure of global error:
  • Across all output units
  • Across all training patterns

• One possible measure is RMS
  • Another is entropy: doesn’t matter too much, since we only need to know if performance is improving or deteriorating on a relative basis
  • But, low overall error doesn’t always mean the network has learned successfully!