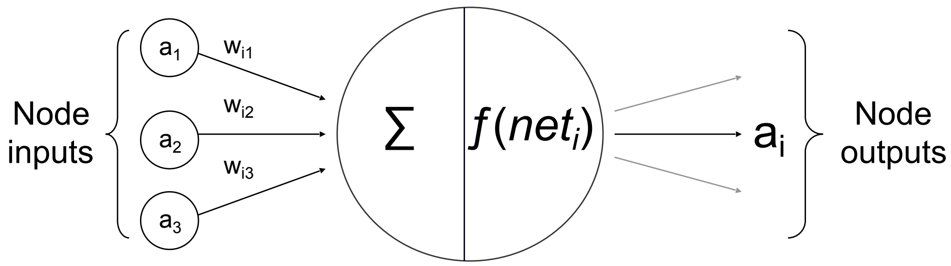
Connectionist Language Processing

Lecture 3: **Learning in Single-layer Networks**

Matthew W. Crocker
crocker@coli.uni-sb.de
Harm Brouwer
brouwer@coli.uni-sb.de

Basic Structure of Nodes



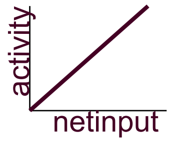
- A node can be characterised as follows:
  - Input connections representing the flow of activation from other nodes or some external source
  - Each input connection has its own weight, which determines how much influence that input has on the node
  - A node i has an output activation ai = f(neti)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

Calculating the activation: $\boldsymbol{net_i}$ is 1.25

- Linear activation:

$$f : \Re \to \Re$$

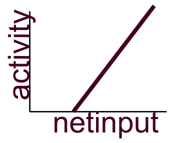
$$\boxed{\begin{aligned} f(net_i) &= net_i \\ f(1.25) &= 1.25 \end{aligned}}$$



- Linear threshold: T=0.5

$$f : \Re \to \Re$$

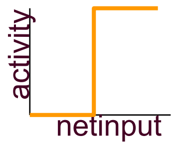
$$\boxed{\begin{aligned} &\text{IF } net_i > T \text{ then } f(net_i) = net_i - T \\ &\text{ELSE } f(net_i) = 0 \\ &f(1.25) = 1.25 - 0.5 = 0.75 \end{aligned}}$$



- Binary threshold: T=0.5

$$f : \Re \to [0,1]$$

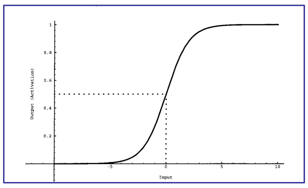
$$\boxed{\begin{aligned} &\text{IF } net_i > T \text{ then } f(net_i) = 1 \\ &\text{ELSE } f(net_i) = 0 \\ &f(1.25) = 1 \end{aligned}}$$



- Nonlinear activation:
  - Sigmoid or "logistic" function

$$f : \Re \to [0,1]$$

$$\boxed{\begin{aligned} f(net_i) &= \frac{1}{1 + e^{-net_i}} \\ f(1.25) &= 0.777 \end{aligned}}$$



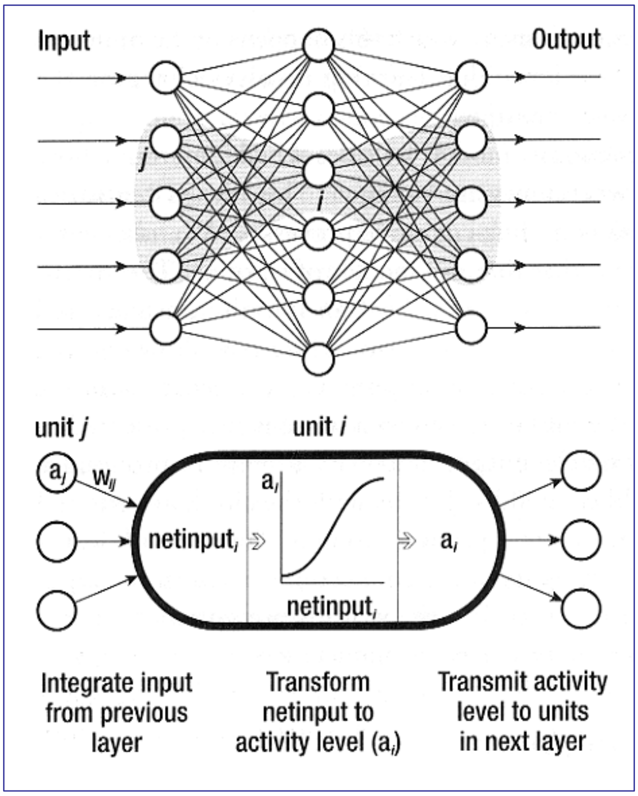
Summary of network architecture

- 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: $a_i * w_{ij}$
- 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

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

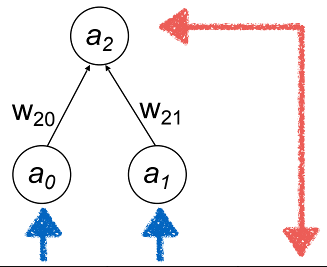
- [a_i(desired)-a_i(obtained] is the difference between the desired output activation and the actual activation produced by the network
- What is the “error”?
- a_j is the activity of the contributing unit j
- How much activation is this unit responsible for?
- ε is the learning rate parameter.
- How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

Training the Network

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

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



Input 1 a_0	Input 2 a_1	Output a_2
0	0	0
0	1	0
1	0	0
1	1	1

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

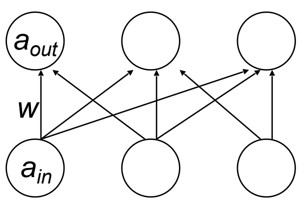
- Here there are 4 such input-output pairings

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

- So, 250 epochs consists of 1000 sweeps

"Perceptrons" [Rosenblatt 1958]

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



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

- Binary threshold activation function:

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

- Learning: the perceptron convergence rule
  - Two parameters can be adjusted:
    - The threshold
    - The weights

The error, $\delta = (t_{out} - a_{out})$

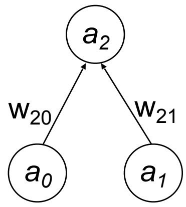
$$\Delta\theta = -\varepsilon\delta$$

$$\Delta w = \varepsilon \delta a_{in}$$

Learning OR

- Consider the following simple perceptron:
  - Recall the convergence rule:

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



Classification problem

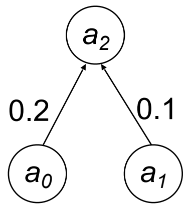
a_0	a_1	a_2
0	0	0
0	1	1
1	0	1
1	1	1

- We want to train this to learn boolean OR:
  - Note: changes have opposite signs
    - E.g if activity is less than target, ∂ is positive: *Threshold is decreased*; *Weight is increased*
  - If ∂ 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 …

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

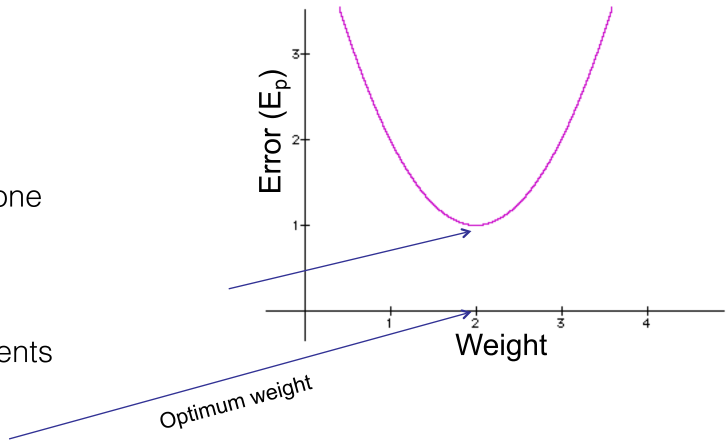
$$\theta = 1$$
$$\varepsilon = 0.5$$



In	w_{20}	w_{21}	θ	a_2	t_2	δ	Δθ	Δw_{20}	Δw_{21}
0 0	.2	.1	1.0	0	0	0	0	0	0

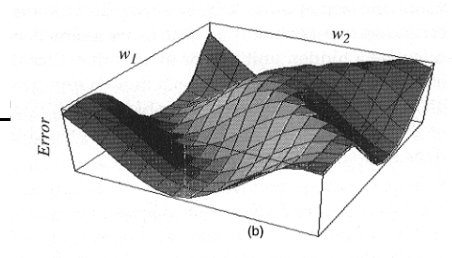
Gradient descent

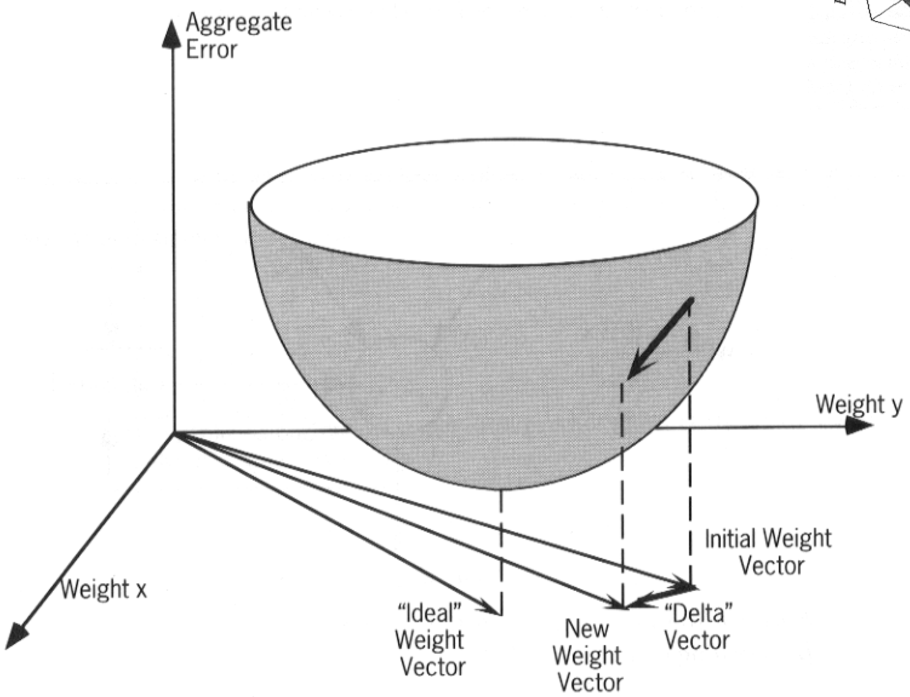
- Let's define the error on the outputs as: $E_p=(t_{out}-a_{out})^2$
  - Recall: $a_{out} = \sum w\, a_{in}$
  - This means E_p is always positive
- For a single layer net, if we consider one weight, holding the others constant:
  - Plot Error versus varying the weight
- The lowest point on the curve, 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 ***<u>slope of the curve</u>*** 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:

$$\Delta w = -\varepsilon \frac{\partial E}{\partial w}$$

Slope: Rate of change of *E*, with *w*

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

Error = $(t_{out} - a_{out})^2$

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

Derivative of the activation function with respect to w, i.e. its slope

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

$$\Delta w = 2\varepsilon \delta F * a_{in}$$

$$\begin{bmatrix} \delta = (t_{out} - a_{out}) \\ F* = \text{slope of the activation function} \end{bmatrix}$$

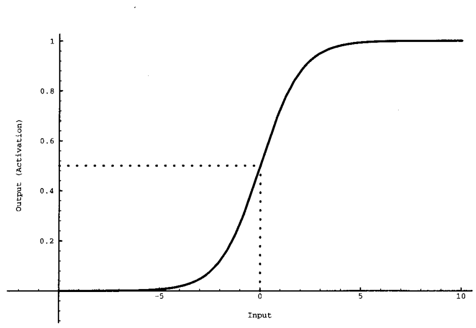
Gradient descent and the delta rule

- The perceptron convergence rule: $\Delta w = \varepsilon \delta a_{in}$
- Our revised learning rule, based on gradient descent is: $\Delta w = 2\varepsilon \delta F * a_{in}$
  - 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 straight-forward to calculate
  - Performs gradient descent to the bottom of an the error curve
  - Δw 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, it 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:

$$a_i = f(net_i) = \frac{1}{1 + e^{net_i}}$$



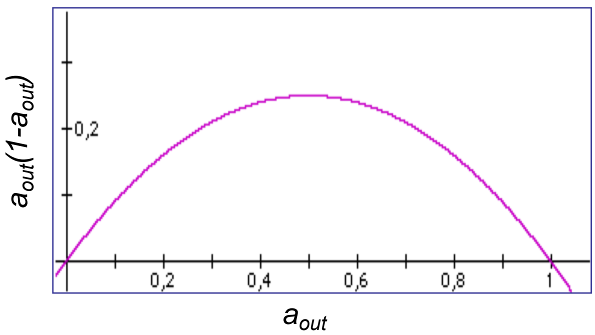
  - Non-linear, more powerful
  - More neurologically plausible
  - Less well-understood, more difficult to analyse mathematically

Behaviour of the logistic function

- Deriving the slope of the logistic function:

$$a_i = f(net_i) = \frac{1}{1 + e^{-net_i}}$$

$$F^* = f'(net_i) = a_{out}(1 - a_{out})$$

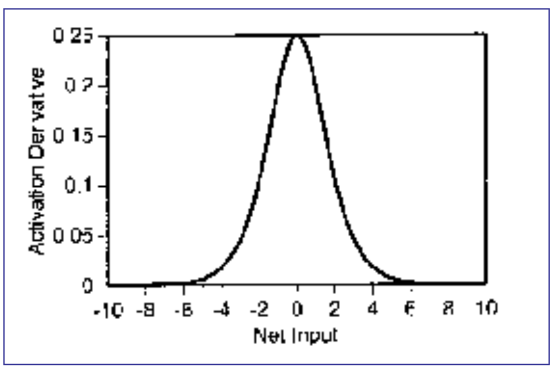


- The Delta rule, assuming the logistic function:

$$\Delta w = 2\varepsilon \delta F^* a_{in}$$

$$or$$

$$\Delta w = 2\varepsilon (t_{out} - a_{out}) a_{out} (1 - a_{out}) a_{in}$$



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):

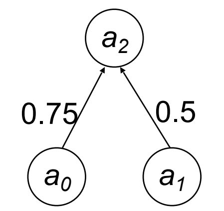
$$\Delta w = \eta(t_{out} - a_{out})a_{out}(1 - a_{out})a_{in}$$

- One such cycle is called a <u>sweep,</u> and a sweep through each pattern is called an <u>epoch</u>

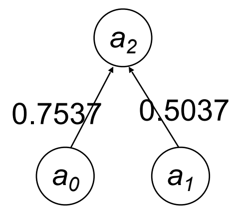
$$\text{rms error} = \sqrt{\frac{\sum_k (\vec{t}_k - \vec{o}_k)^2}{k}}$$

- We can define the <u>global error</u> 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:



$$a_2 = f(1 \times 0.75 + 1 \times 0.5) = 0.77$$
$$\delta_2 = (t - a_2)f'(0.77) = 0.23 \times 0.16 = 0.037$$
$$\Delta w_{20} = \eta\delta_2 o_0 = 0.1 \times 0.037 \times 1 = 0.0037$$
$$\Delta w_{21} = \eta\delta_2 o_1 = 0.1 \times 0.037 \times 1 = 0.0037$$

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:

Input	Target	Output	(t-o)^2
0 0	0	0,147	0,022
0 1	0	0,297	0,088
1 0	0	0,334	0,112
1 1	1	0,552	0,201
		RMS:	0,325

$$\text{rms error} = \sqrt{\frac{\sum_k (\vec{t}_k - \vec{o}_k)^2}{k}}$$

- Observe how error steadily falls during training

Connectionist Language Processing – Crocker & Brouwer

Calculating Global RMS Error

Calculation of Global RMS error: for (auto1), ch. 5, Plunkett & Elman

	Observed Output				Target			
pattern 1	0,321	0,196	0,255	0,264	1,000	0,000	0,000	0,000
pattern 2	0,227	0,612	0,169	0,211	0,000	1,000	0,000	0,000
pattern 3	0,287	0,188	0,342	0,276	0,000	0,000	1,000	0,000
pattern 4	0,296	0,207	0,300	0,268	0,000	0,000	0,000	1,000

Error (t-o)

0,679	-0,196	-0,255	-0,264
-0,227	0,388	-0,169	-0,211
-0,287	-0,188	0,658	-0,276
-0,296	-0,207	-0,3	0,732

Error^2

0,461041	0,038416	0,065025	0,069696	0,634178
0,051529	0,150544	0,028561	0,044521	0,275155
0,082369	0,035344	0,432964	0,076176	0,626853
0,087616	0,042849	0,09	0,535824	0,756289
			RMS Error	0,757046

$$\text{rms error} = \sqrt{\frac{\sum_k (\vec{t}_k - \vec{o}_k)^2}{k}}$$

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!