Computational Psycholinguistics

Lecture 9: Learning in Neural Networks

Marshall R. Mayberry

Computerlinguistik

Universität des Saarlandes

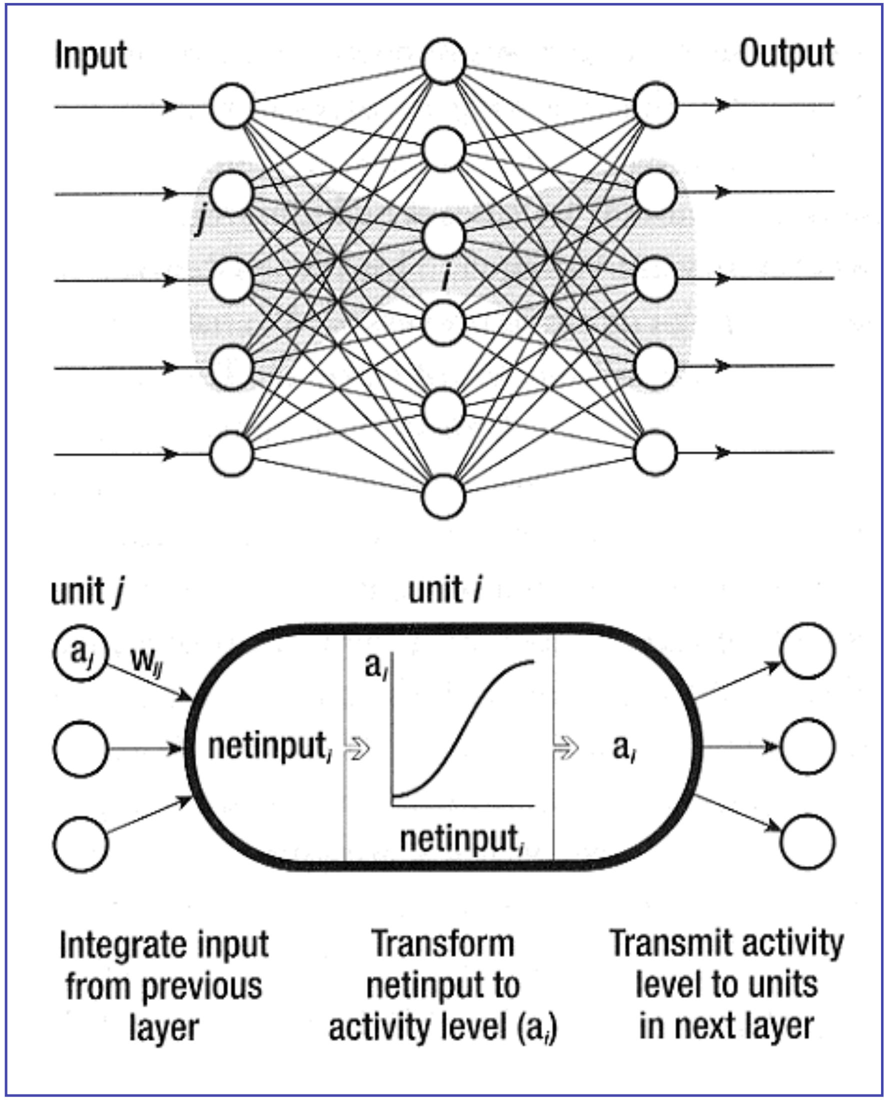
Neural network architecture

- 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: $a_j * w_{ij}$
- For a node i in the network:

$$netinput_i = \sum_j w_{ij} a_j$$

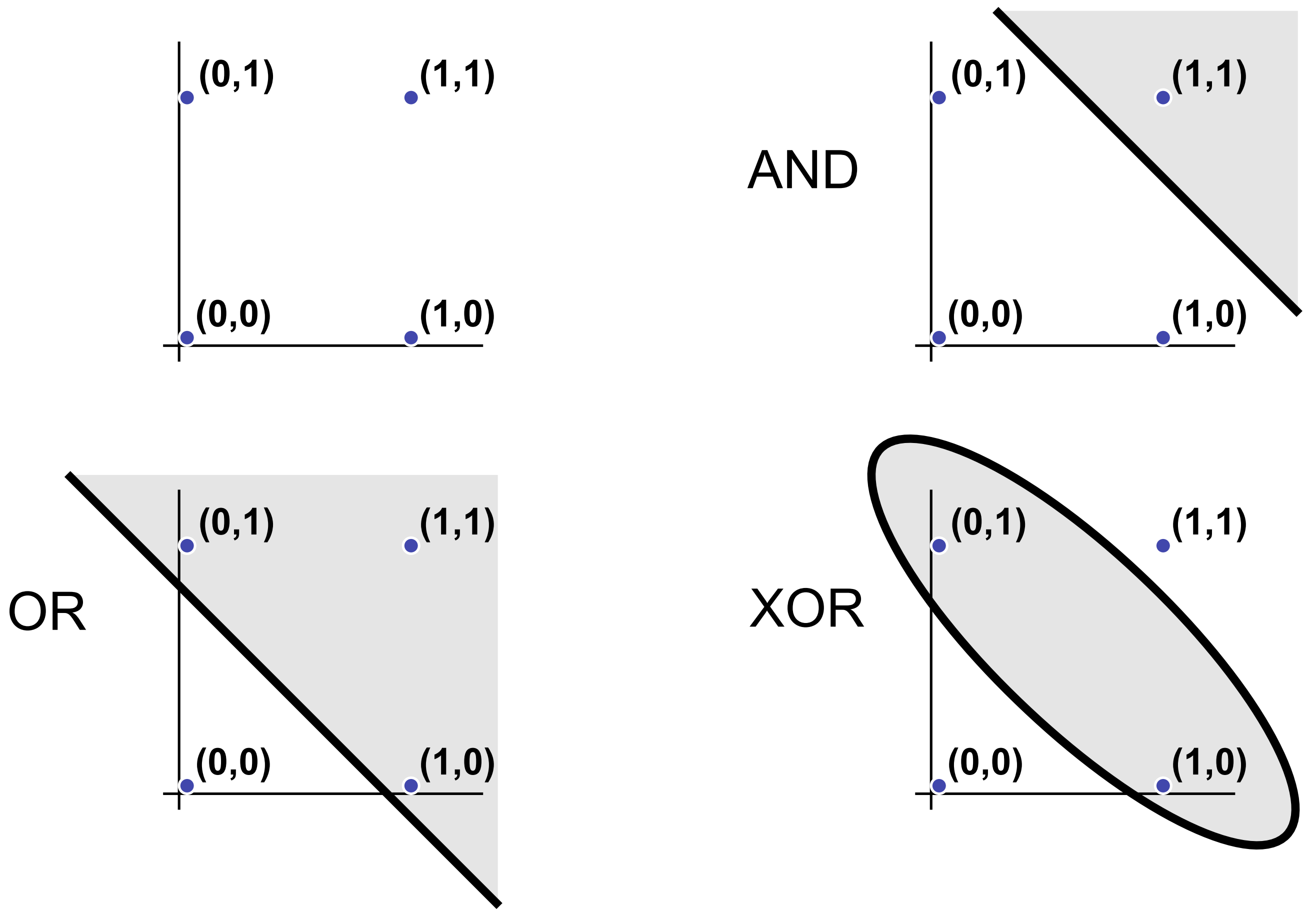
- The output activation of node i is determined by the activation function, e.g. the logistic:

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



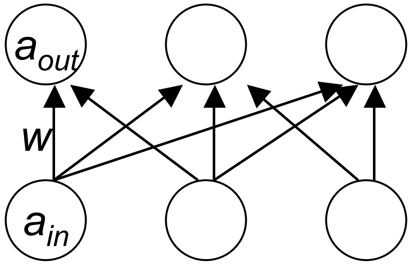
2-D Representation of Boolean Functions

- We can visual the relationship between inputs (plotted in 2-D space) and the desired output (represented as a line dividing the space):



"Perceptrons" [Rosenblatt 1958]

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



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

- Binary threshold activation function:

$$\begin{aligned} a_{out} &= 1 \text{ if net}_{out} > \theta \\ &= 0 \text{ otherwise} \end{aligned}$$

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

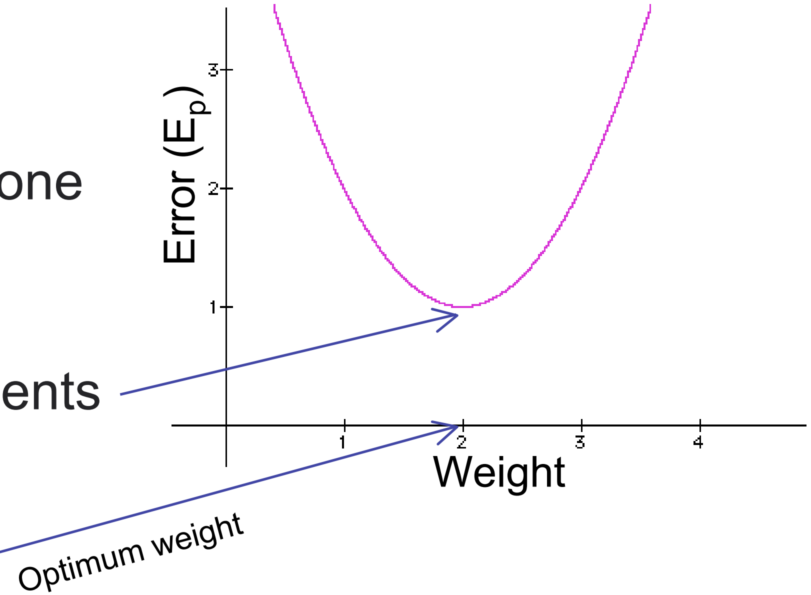
$$\text{The error, } \delta = (t_{out} - a_{out})$$

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

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

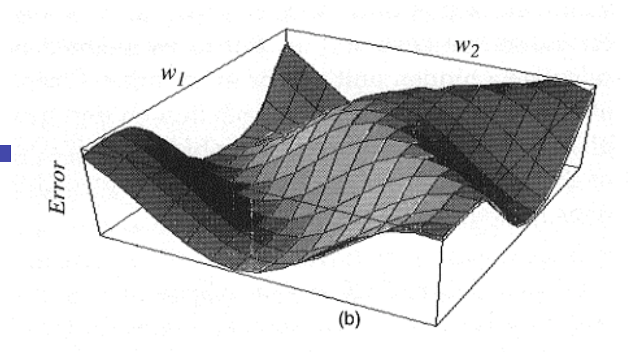
Gradient descent

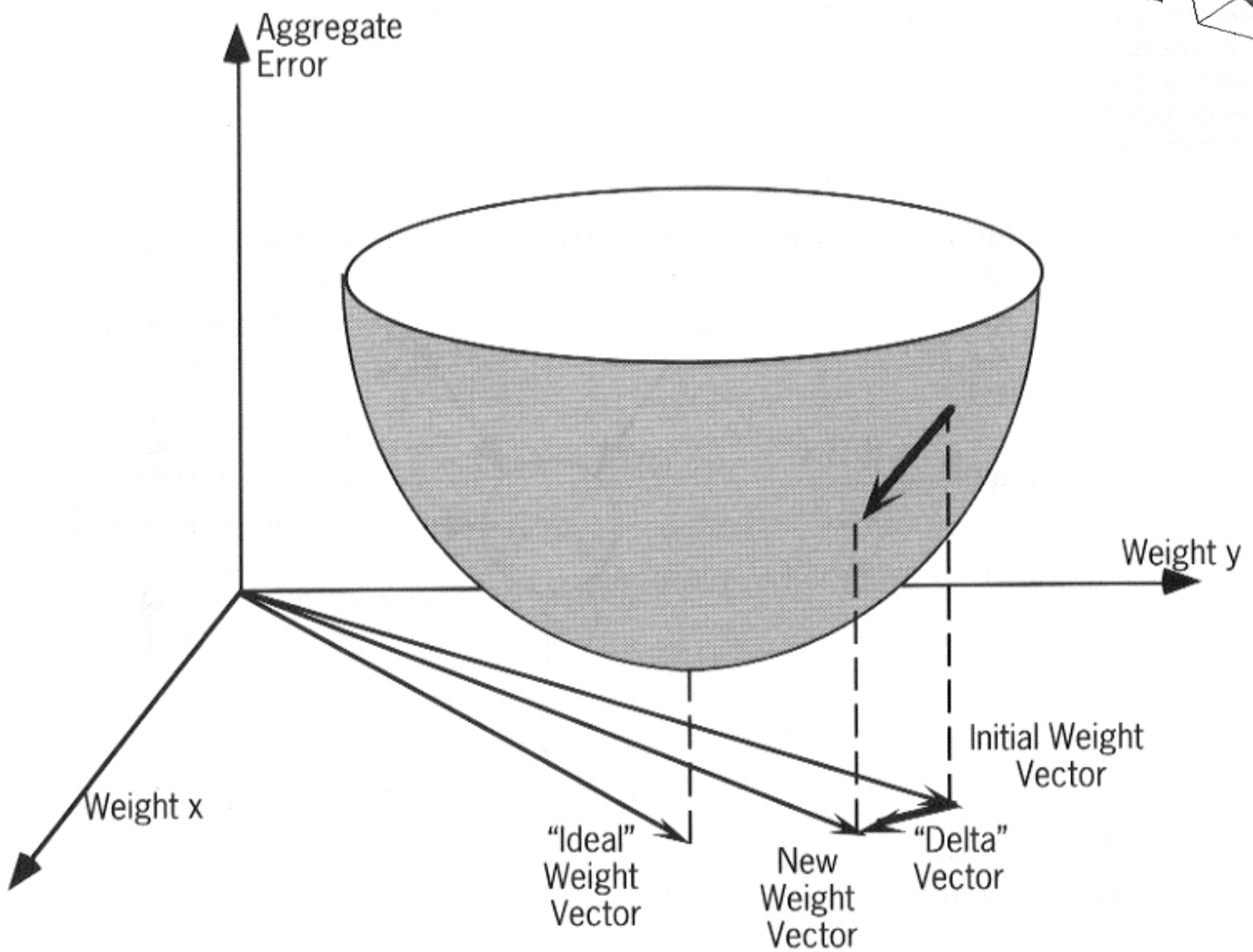
- Let's define the error on the outputs as: $E_p=(t_{out}-a_{out})^2$
  - Recall: $a_{out} = \sum w\ a_{in}$
  - This means E_p is always positive
- For a single-layer net, if we consider one weight, holding the others constant:
  - Plot Error versus varying the weight
- The lowest point on the curve,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





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 respect to *w*

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

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

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

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

Derivative of the activation function wrt w, i.e. its slope

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

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

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, the slope is constant:

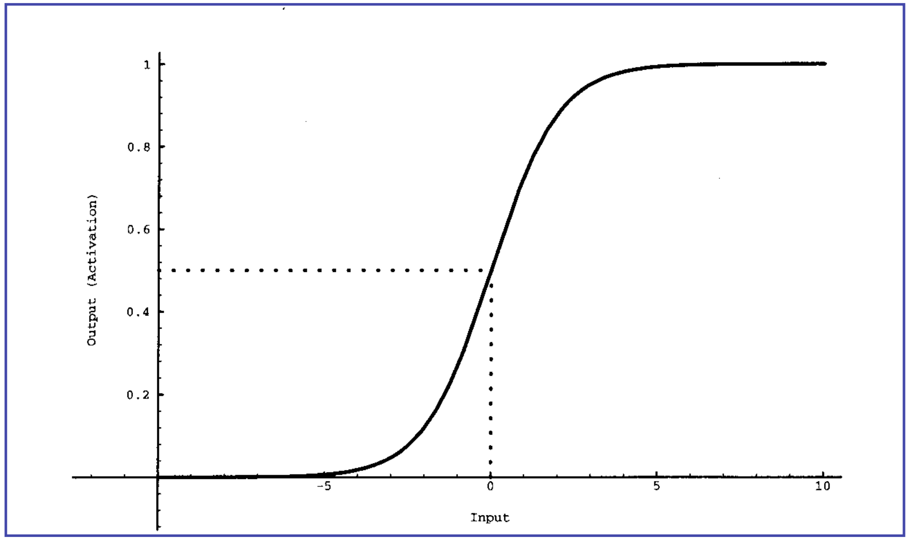
$$\Delta w = k \delta a_{in}$$

  - where k is a constant representing the learning rate ε and slope
- This corresponds to the original Delta rule:
  - It is straightforward to calculate
  - Performs gradient descent to the bottom of the error curve
  - Δw is proportional to $(t_{out} - a_{out})$, so changes get smaller as error is reduced
  - In one-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 (linear algebra)
  - they are known to be limited in the kinds of problems they can solve
- The logistic, or sigmoid, function is:
  - Nonlinear: more powerful
  - More neurologically plausible
  - Less well-understood, more difficult to analyse mathematically
- Recall:

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

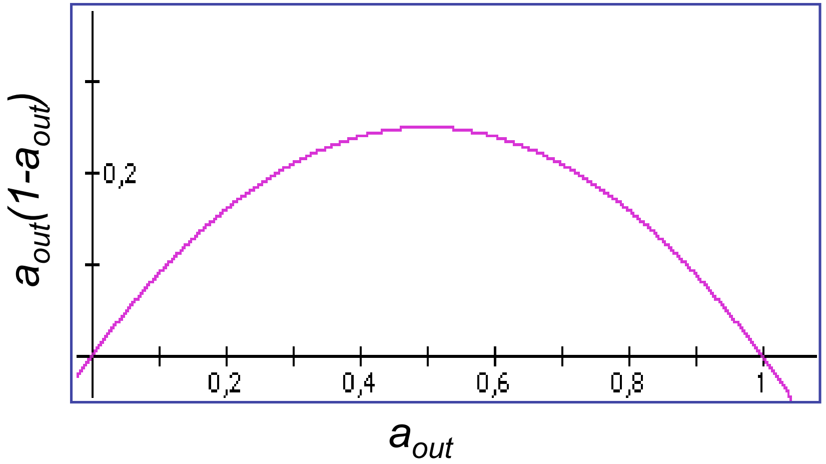


Behaviour of the logistic function

- Deriving the slope of the logistic function:

$$a_{out} = \sigma(net_{out}) = \frac{1}{1+e^{-net_{out}}}$$

$$F^{*} = \sigma'(net_{out}) = a_{out}(1-a_{out})$$

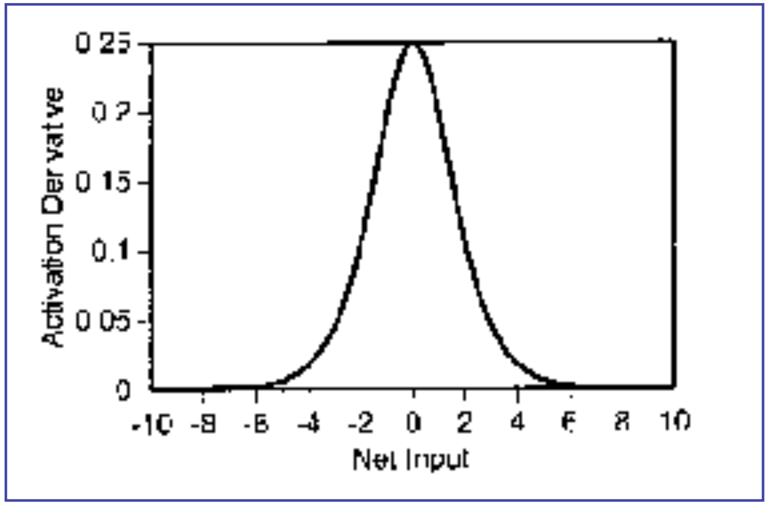


- The Delta rule, assuming the logistic function:

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

$$or$$

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



Training a network with Gradient Descent

- 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 logistic activation function):

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

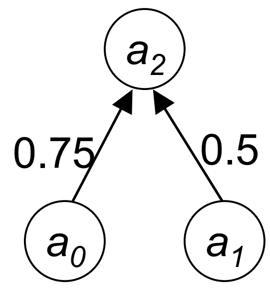
- One such cycle is called a sweep
- A sweep through each pattern is called an epoch
- We can define the global error of the network, as the average error across all input patterns, k:
  - One common measure is the square root of mean error or else root mean square (rms)

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

  - Squaring avoids positive and negative errors cancelling each other out

Training: an example

- Consider the simple feedforward network:
  - Assume an input pattern: 1 1
  - Assume a learning rate of 0.1
  - Assume a sigmoid activation
  - Desired output is: 1



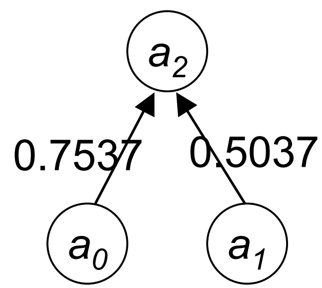
- Determine the weight changes for 1 sweep:

$$a_2 = f(1 \times 0.75 + 1 \times 0.5) = 0.78$$

$$\delta_2 = (t - a_2) f'(0.78) = 0.23 \times 0.16 = 0.037$$

$$\Delta w_{20} = \varepsilon \delta_2 o_0 = 0.1 \times 0.037 \times 1 = 0.0037$$

$$\Delta w_{21} = \varepsilon \delta_2 o_1 = 0.1 \times 0.037 \times 1 = 0.0037$$



The dynamics of weight changes

- Learning rate: predetermined constant
- 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
    - Less 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

- Observe how error steadily falls during training

Calculating Global RMS Error

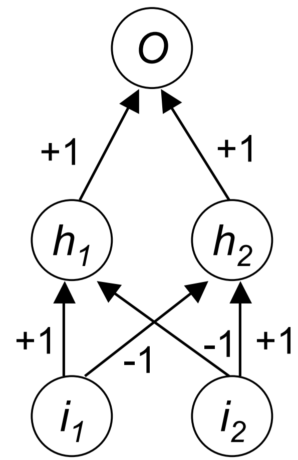
	Calculation of Global RMS error: for (auto1), ch. 5, Plunkett & Elman							
	Observed Output				Target			
pattern 1	0,321	0,196	0,255	0,264	1,000	0,000	0,000	0,000
pattern 2	0,227	0,612	0,169	0,211	0,000	1,000	0,000	0,000
pattern 3	0,287	0,188	0,342	0,276	0,000	0,000	1,000	0,000
pattern 4	0,296	0,207	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			

Intermediate Summary

- Learning rules:
  - Perceptron convergence rule
  - Delta rule
    - Depends on the (slope of the) activation function
  - For one-layer networks using these rules:
    - A solution will be found, if it exists
  - How do we know if the network has learned successfully?
- Error:
  - For learning, we use $(t_{out} - a_{out})$ to change weights
  - To characterise the performance of the network as a whole, we need a measure of global error:
    - Across all outputs
    - Across all training patterns
  - One possible measure is RMS
    - Another is entropy: doesn't really matter, since we only need to know if performance is improving or deteriorating on a relative basis

Solving XOR with hidden units

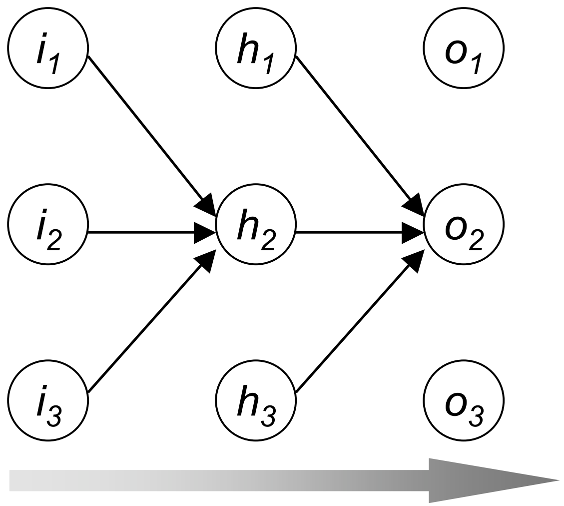
- Consider the following network:
  - two-layer, feedforward
  - 2 units in a "hidden" layer
  - Hidden and output units are threshold units: $\theta = 1$
- Representations at hidden layer:



Input	Hidden		Target
	h_1	h_2	
0 0	0	0	0
1 0	1	0	1
0 1	0	1	1
1 1	0	0	0

- Problem: current learning rules cannot be used for hidden units:
  - Why? We don't know what the "error" is at these nodes (no target)
  - "Delta" requires that we know the desired activation $\boxed{\Delta w = 2\varepsilon\delta F^* a_{in}}$

Backpropagation of Error



(a) Forward propagation of activity :

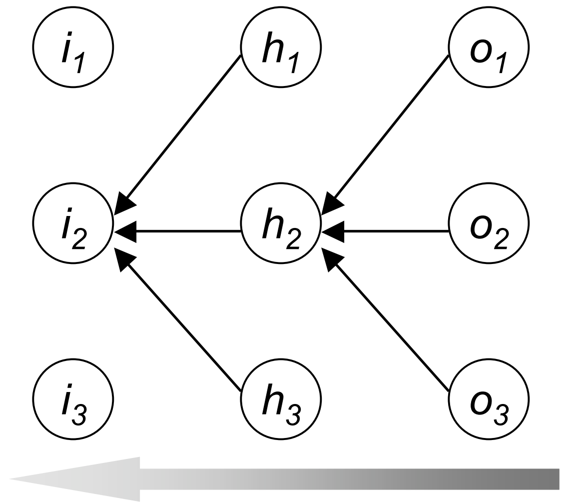
$$\text{net}_{out} = \sum w_{oh} \cdot a_{hidden}$$

$$a_{out} = f(\text{net}_{out})$$

(b) Backward propagation of error :

$$\text{err}_{hidden} = \sum w_{oh} \cdot \delta_{out}$$

$$\delta_{hidden} = f'(\text{net}_{hidden}) \cdot \text{err}_{hidden}$$



Learning in Multi-layer Networks

- The generalised Delta rule:

$$\Delta w_{ij} = \varepsilon \delta_i a_j$$

For output nodes:

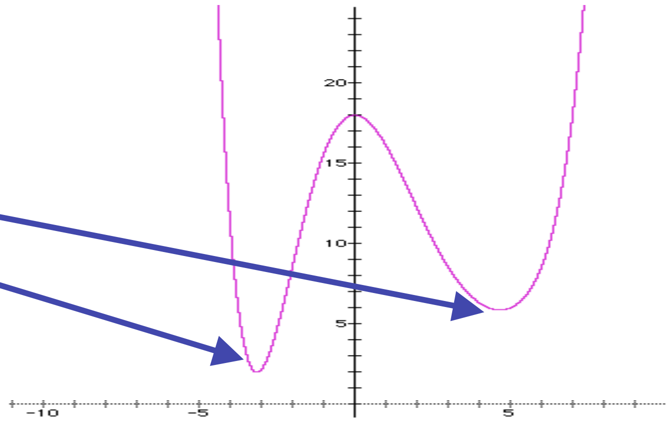
$$\delta_k = \sigma'(net_k)(t_k - a_k)$$

For hidden nodes:

$$\delta_i = \sigma'(net_i)\sum_k w_{ki}\delta_k$$

where, $\sigma'(net_i) = a_i(1 - a_i)$

- Multi-layer networks can, in principle, learn any mapping function:
  - Not constrained to problems which are linearly separable
- While there exists a solution for any mapping problem, backpropagation is not guaranteed to find it
  - Unlike the perceptron convergence rule
- Why? Local minima:
  - Backprop can get trapped here
  - Global minimum (solution) is here
  - Not real problem in practice



Example of Backpropagation

- Consider the following network, containing a single hidden node
- Calculate the weight changes for both layers of the network, assuming learning rate ε = 0.1 and targets of: 1 1

The generalised Delta rule:

$$\Delta w_{ij} = \varepsilon \delta_i a_j$$

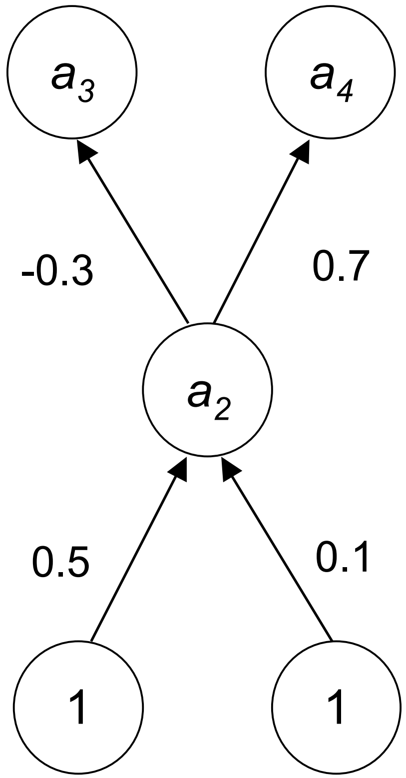
For output nodes:

$$\delta_k = \sigma'(net_k)(t_k - a_k)$$

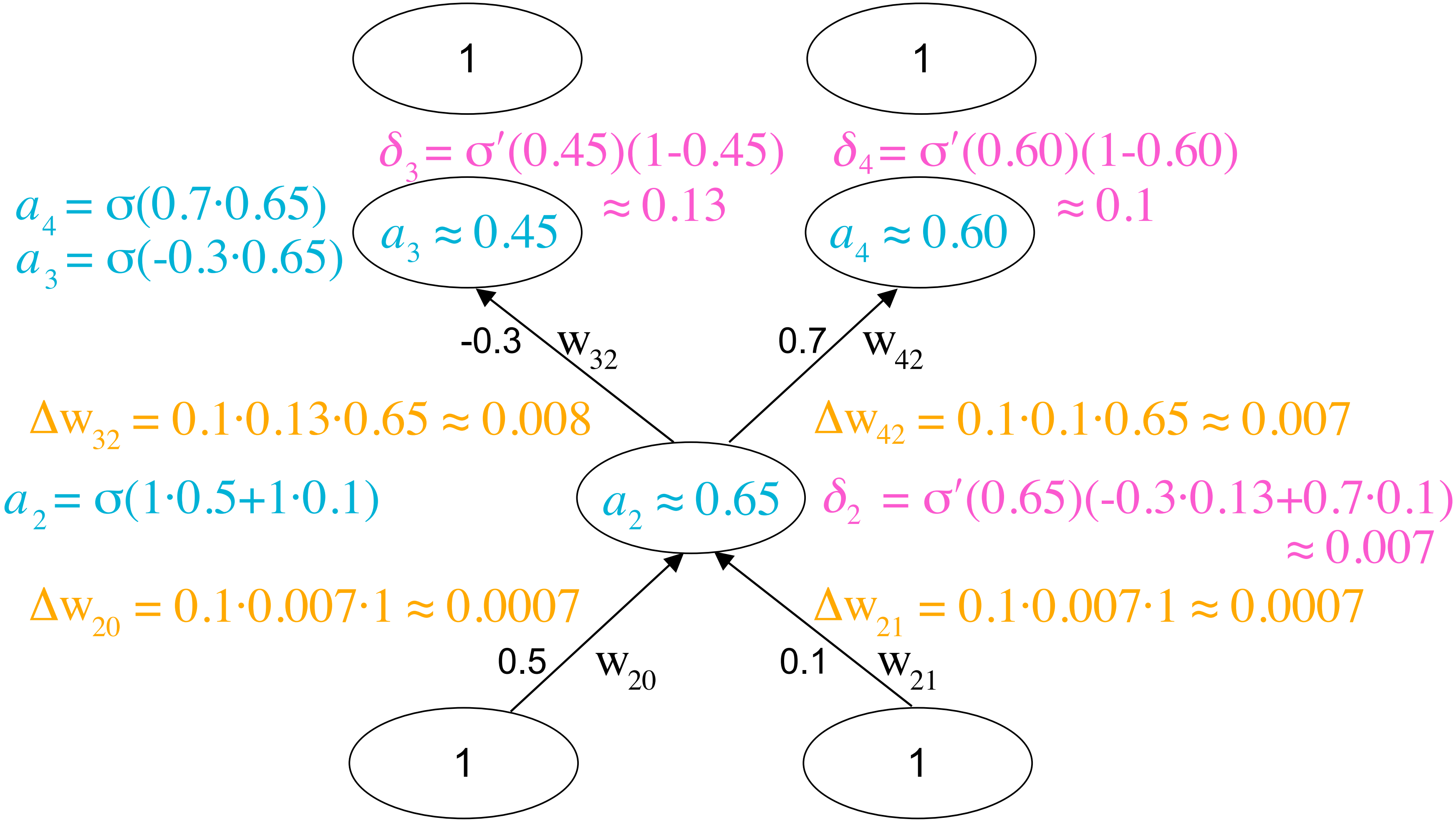
For hidden nodes:

$$\delta_i = \sigma'(net_i)\sum_k \delta_k w_{ki}$$

where, $\sigma'(net_i) = a_i(1 - a_i)$

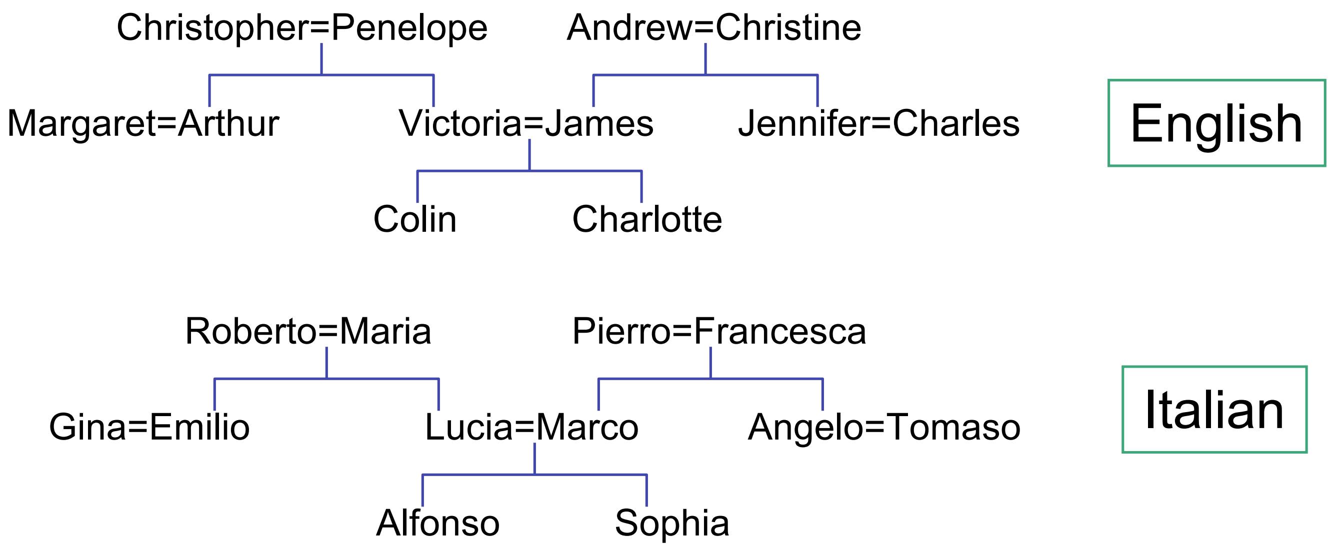


Forward and Backpropagation



The Family Tree Problem

- Family trees encode more complex relationships:



- 24 people, 12 relationships
  - Mother, daughter, sister, wife, aunt, niece (+ masculine versions)
- Training: trained on 100 of 104 possible relationships
- Learned the other 4: e.g. Victoria's son is Colin

What does the Network Learn

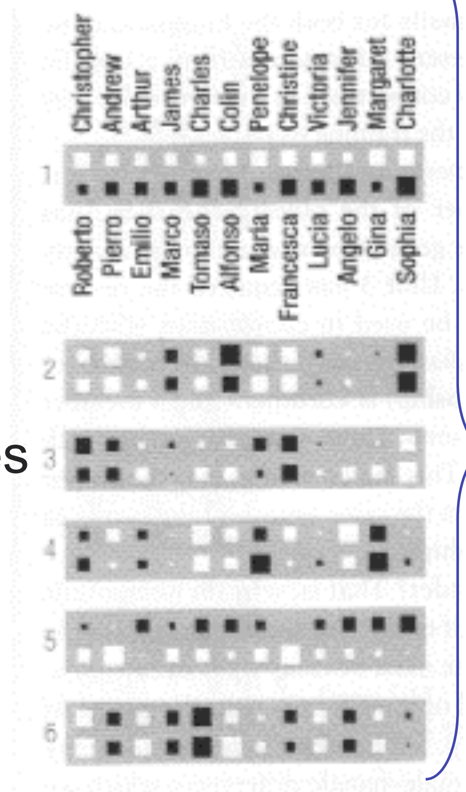
- E.g. Victoria's son is Colin:
  - Input: Victoria & Son
  - Output: Colin
- In a single-layer network:
  - Victoria would activate all the people Victoria was (known to be) related to
  - Son would activate all people who are (known to be) sons
    - Colin would be partially activated, because he is James' son
  - But Colin would not have very high activation
    - James and Arthur are both sons, and related to Victoria
- A solution to this problem requires deduction:
  - Transitive inference:
    - Victoria's husband is James AND James' son is Colin
    - THEREFORE Victoria's son is Colin
- Thus the structure of the tree is learned from exemplars

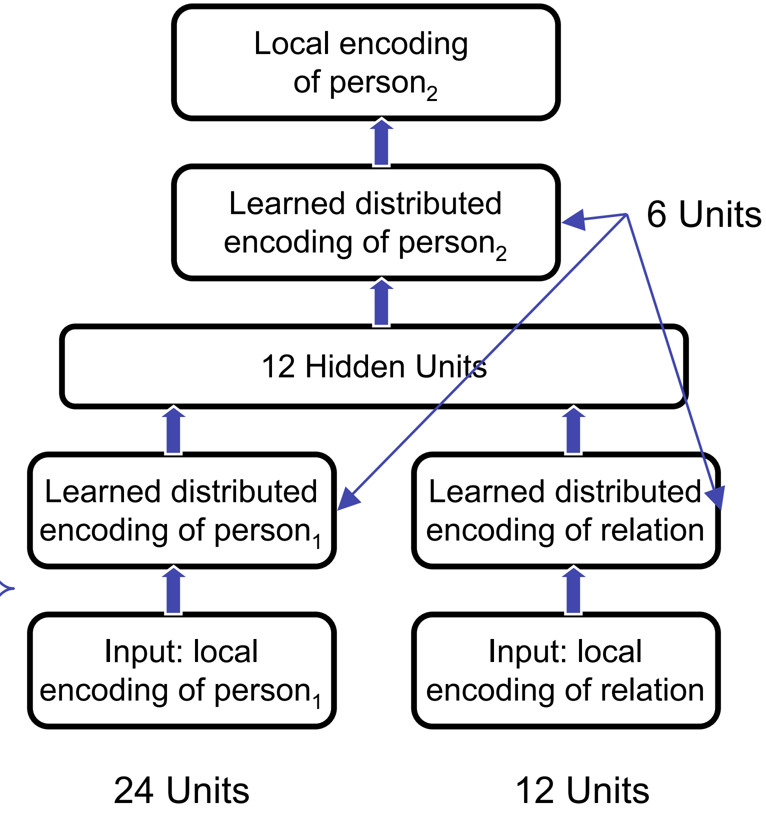
Learning family tree relationships

- The network architecture, using hidden units:

- The learned encoding of people:

1. Active for English
2. Active for older generation
3. Active for the leaves
4. Encodes right side
5. Active for Italian
6. Encodes left side





Some comments

- Single layer networks (perceptrons)
  - Can only solve problems which are linearly separable
  - But a solution is guaranteed by the perceptron convergence rule
- Multi-layer networks (with hidden units)
  - Can in principle solve any input-output mapping function
  - Backpropagation performs gradient descent of the error surface
  - Can get caught in a local minimum
  - Cannot be guaranteed to find the solution
- Finding solutions:
  - Manipulate learning rule parameters: learning rate, momentum
  - Brute force search (sampling) of the error surface to find a set of starting position in weight space
    - Computationally impractical for complex networks

Biological plausibility

- Backpropagation requires bi-directional signals
  - Forward propagation of activation
  - Backward propagation of error
  - Nodes must “know” the strengths of all synaptic connections to compute error: non-local
- Axons are uni-directional transmitters
- Possible justification:
  - Backpropagation explains *what* is learned,
  - Not *how* it is learned
- Network architecture:
  - Successful learning crucially depends on the number of hidden units
  - There is no way to know, a priori, what that number is
- Another solution: use a network with a local learning rule
  - E.g. Hebbian learning