
Artificial Intelligence

V. Machine Learning

2. Artificial Neural Networks

F. C. Langbein

School of Computer Science
and Informatics
Cardiff University





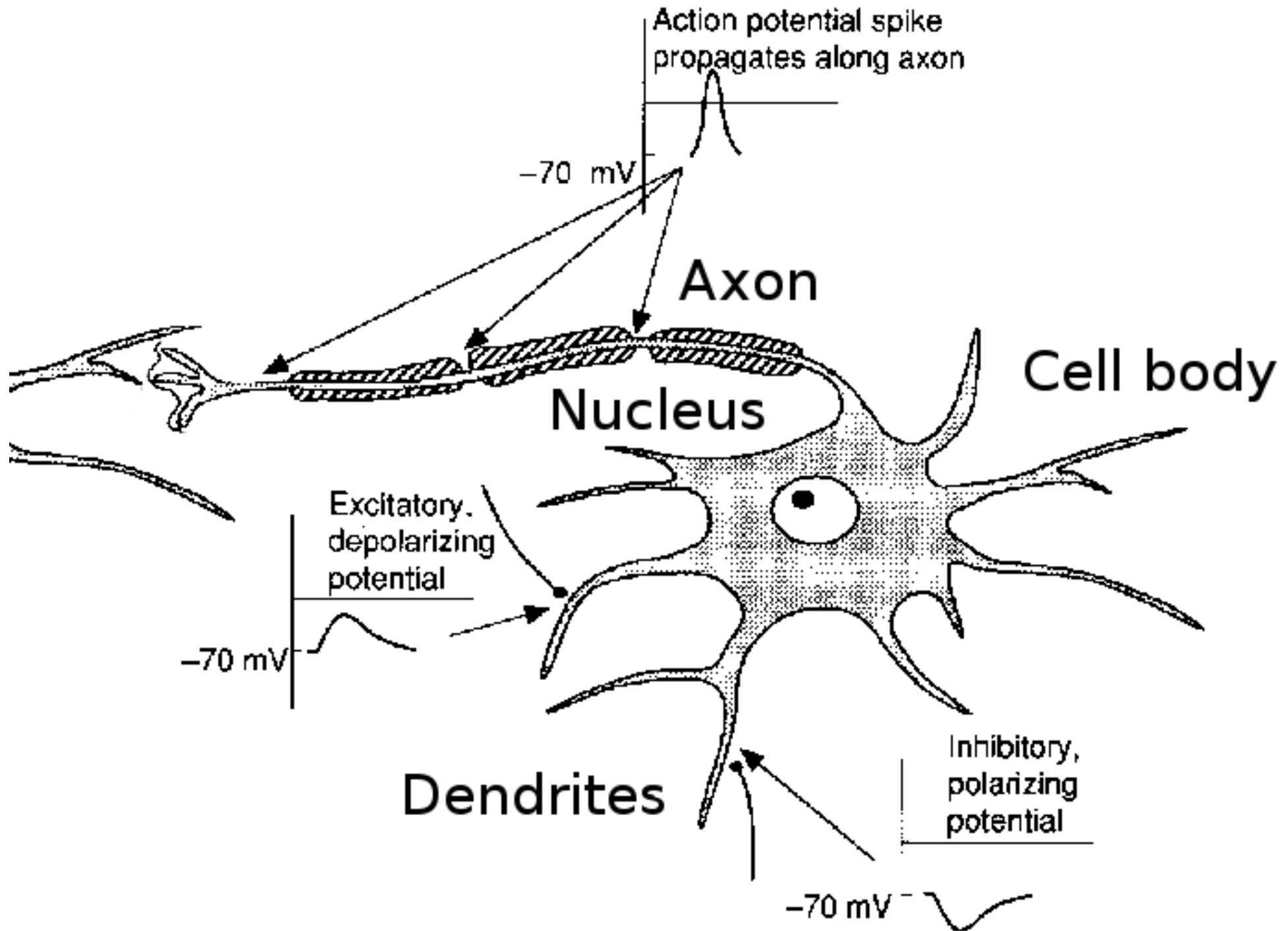
Overview

- Human Brain and Neurons
- Artificial Neurons
- Artificial Neural Network Programming
 - Network structures
 - Error minimisation
- Perceptrons
 - Perceptron learning
 - Multiclass Perceptron
 - Back propagation

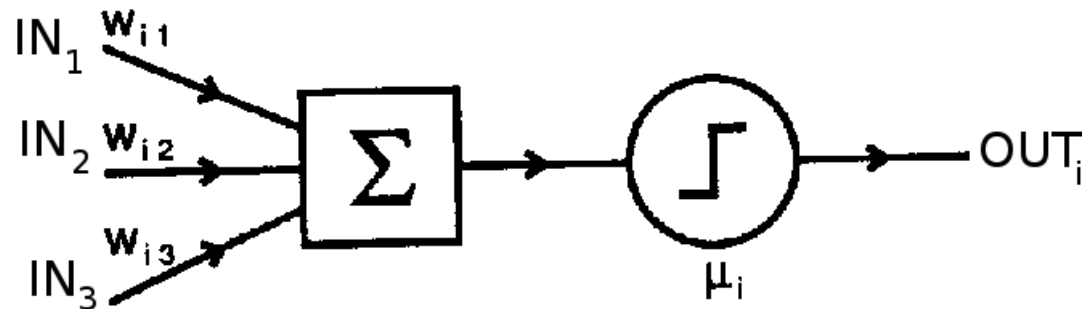
The Human Brain

- Robust and fault tolerant
- Capable to learn by example
- Can handle inconsistent information
- Massively parallel ($\sim 10^{11}$) with slow neurons (10 – 100Hz)
- Compact with small power requirements
- Good in selective omission of information
 - instant recognition of (certain) faces (< 1s)
 - speech and music recognition
 - assessing and planning
- Not so good on arithmetic tasks, etc.

A Neuron



McCulloch-Pitts Neuron



- **Linear** units: output proportional to weighted input

$$OUT_i = I_i := \sum_l w_{i,l} IN_l - \mu_i$$

- **Threshold** / Heaviside units: two output levels

$$OUT_i = 1 \text{ iff } I_i > 0, 0 \text{ otherwise}$$

- **Sigmoid** / logistic units: non-linear output function

$$OUT_i = \frac{1}{(1 + \exp(-I_i))}$$

Programming Paradigm

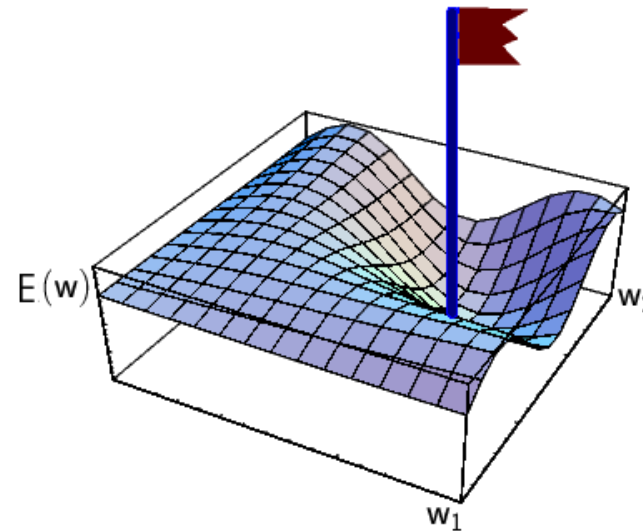
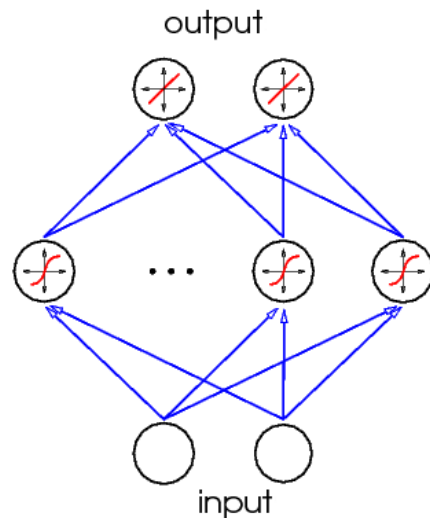
- Construct network of neurons
 - Learn by example: **minimise error** for examples such that learned function **generalises** to unknown cases
- **Feed-forward** networks: signals travel in one way only
 - Often arranged in layers
(usually three layers: inputs, hidden layer, outputs)
 - Good for pattern recognition
- **Feedback** networks: contains loops
 - Very powerful, but also very complicated
 - Dynamically change states until equilibrium reached
 - Equilibrium changes when input changes

Minimisation

- Given learning examples

$$D = \{(p_1, q_1), \dots, (p_n, q_n)\} \text{ with } p_l \in \mathbb{R}^I, q_l \in \mathbb{R}^O$$

- Network function: $h_w : \mathbb{R}^I \rightarrow \mathbb{R}^O$

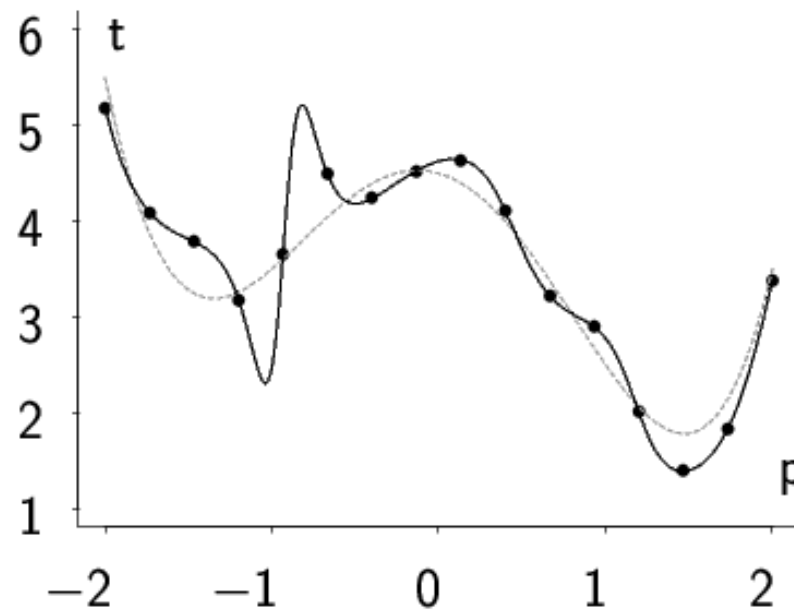
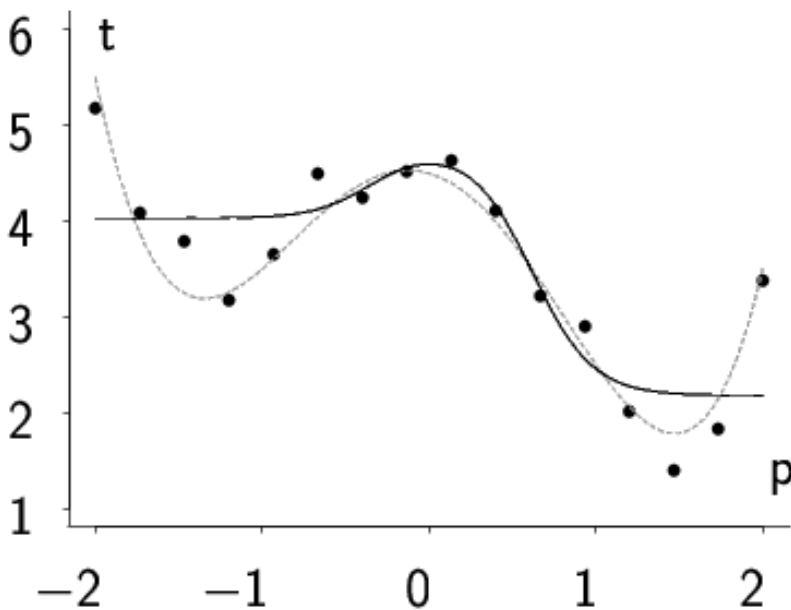


- Minimise errors $h_w(p_l) \neq q_l$ by adjusting w , e.g.

$$E(w) = \frac{1}{2} \sum_{(p_l, q_l) \in D} (h_w(p_l) - q_l)^2$$

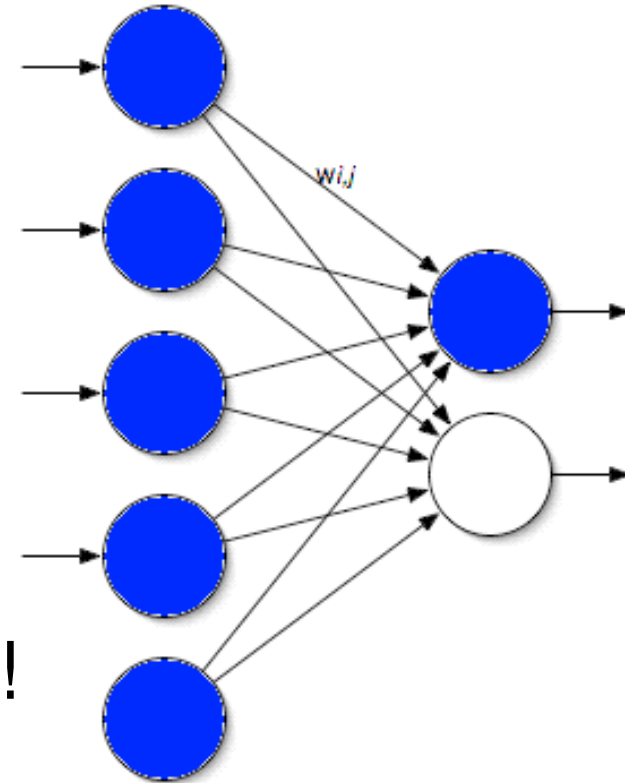
Generalisation

- Error may be minimised by gradient descent, back-propagation, etc. algorithms
- But resulting network should also be a good **generalisation** of the examples
- Under-fitting and over-fitting example



Perceptrons

- **Single-layer** network: outputs directly connected to inputs
- Sufficient to consider one output node
- Represents many Boolean functions efficiently with *threshold*
- *Majority function*: 1 iff $\geq n/2$ inputs are 1
 - Set all weights to 1 with threshold $n/2$
 - Decision tree would need $O(2^n)$ nodes!
- *Cannot* represent all Boolean functions
 - Condition $\sum_l w_l p_l > \mu$ means that all 1 inputs have to lie on one side of a *hyper-plane* (AND, OR, but not XOR)
- ➔ **Linear separator**



Perceptron Learning

➤ Learning algorithm for **linearly separable** training set

- Gradient descent

- Squared error for single training example (p, q) :

$$E = \frac{1}{2}(h_w(p) - q)^2$$

- Gradient:

$$\begin{aligned}\frac{\partial E}{\partial w_l} &= (h_w(p) - q) \frac{\partial}{\partial w_l} h_w(p) \\ &= (h_w(p) - q) \frac{\partial}{\partial w_l} G \left(\sum_k w_k p_k - \mu \right) \\ &= (h_w(p) - q) G' \left(\sum_k w_k p_{l,k} - \mu \right) p_l\end{aligned}$$

Perceptron Learning

➤ Learning algorithm for **linearly separable** training set

- Gradient descent

$$w_l \leftarrow w_l - \alpha * \text{Err} * G'(In) * p_l$$

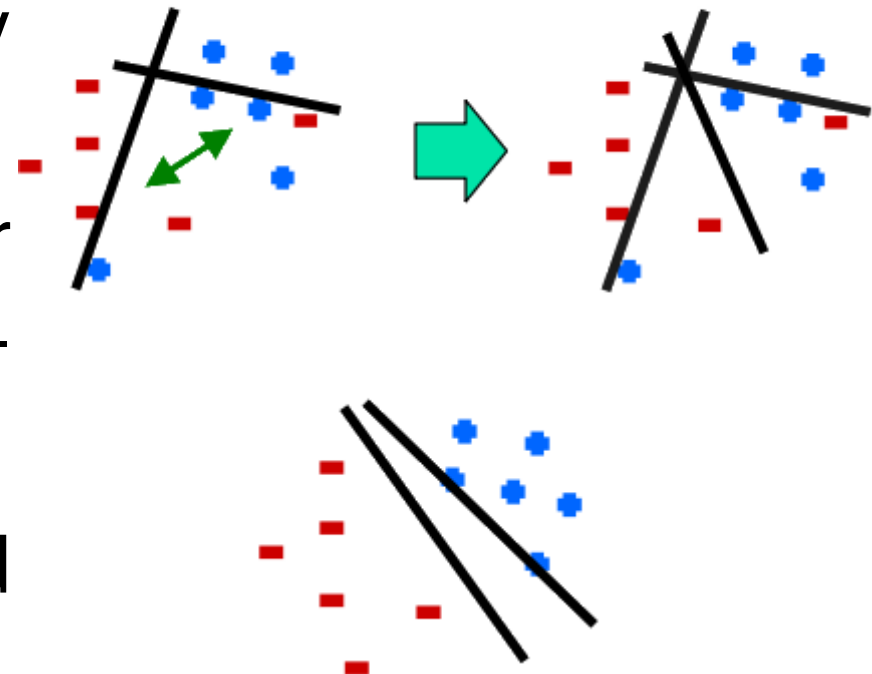
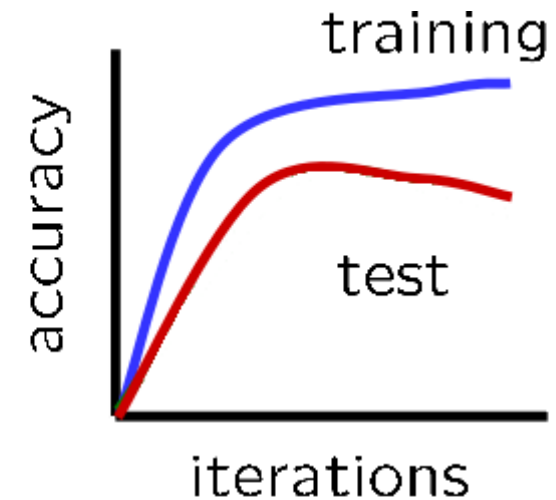
– with learning rate $\alpha > 0$

- Compute weights for a single output node

```
function PERCEPTRON-LEARNING (training)  
repeat  
  for each (p, q) in training  
     $in \leftarrow \sum_k w_k p_k - \mu$   
     $err \leftarrow G(in) - q$   
     $w_l \leftarrow w_l - \alpha * err * G'(in) * p_l$   
until CONVERGED
```

Issues with Perceptrons

- **Overtraining:** accuracy on test data rises, then falls
 - Similar to overfitting, but not quite as bad
- **Regularisation:** if data is not linearly separable, weights may jump
 - Averaging weight vectors over time can help (averaged perceptron)
- **Mediocre generalisation:** may find barel separating solution



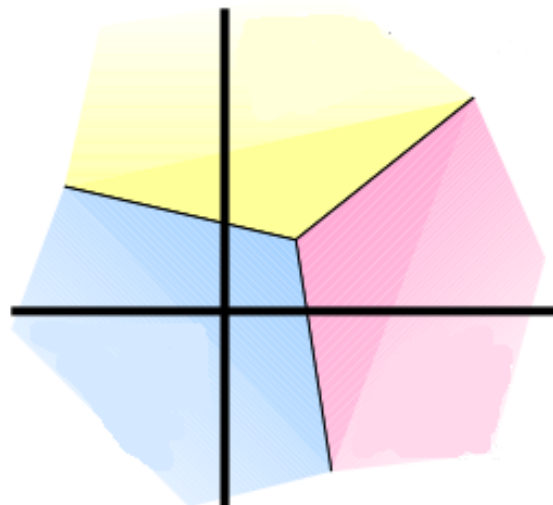
Multiclass Perceptron

- To classify data into more than two classes:
 - Use weight vector w_c for each class c
 - Calculate output for each class

$$\text{OUT}(\text{IN}, w_c) = \sum_l w_{c,l} \text{IN}_l$$

- Highest output gives class

$$c^* = \arg \max_c (\text{OUT}(\text{IN}, w_c))$$





Back Propagation

- Learning for multi-layer networks similar to perceptron algorithm
 - But we have several outputs:
 - each training example has an output vector
 - Error in hidden layers unclear
- ➔ **Back-propagate** error from output layer to hidden layers

Back Propagation

- For output layer: same as before with error vector

$$w_{k,l} \leftarrow w_{k,l} - \alpha * \Delta_l * p_k$$

- with modified error $\Delta_l = \underbrace{[h_w(p) - q]_l}_{l\text{-th error}} * G'(In_l)$

- To propagate the error to the hidden layers:

- k-th hidden node responsible for fraction of Δ_l
 - according to strength of connection between hidden layer and output layer
- So for hidden layers use

$$\Delta_k = \sum_j w_{k,j} \Delta_j * G'(In_k)$$

- with inputs a_k (instead of p_k) from previous layer

Back Propagation

```
function BACK-PROPAGATION-LEARNING (training)
repeat
  for each (p, q) in training
    for each node k in input layer do  $a_k \leftarrow p_k$ 
    for  $l \leftarrow 2, \dots, M$ , each node k in layer l do
       $in_k \leftarrow \sum_j w_{j,k} a_j - \mu_k$ ,  $a_k \leftarrow G(in_k)$ 
    for each node l in output layer do
       $\Delta_l \leftarrow G'(in_l) * (a_l - q_l)$ 
    for  $l \leftarrow M - 1, \dots, 1$  do
      for each node k in layer l do
         $\Delta_k \leftarrow G'(in_k) \sum_j w_{k,j} \Delta_j$ 
        for each node j in layer l + 1 do
           $w_{k,j} \leftarrow w_{k,j} - \alpha * a_k * \Delta_j$ 
until CONVERGED
```