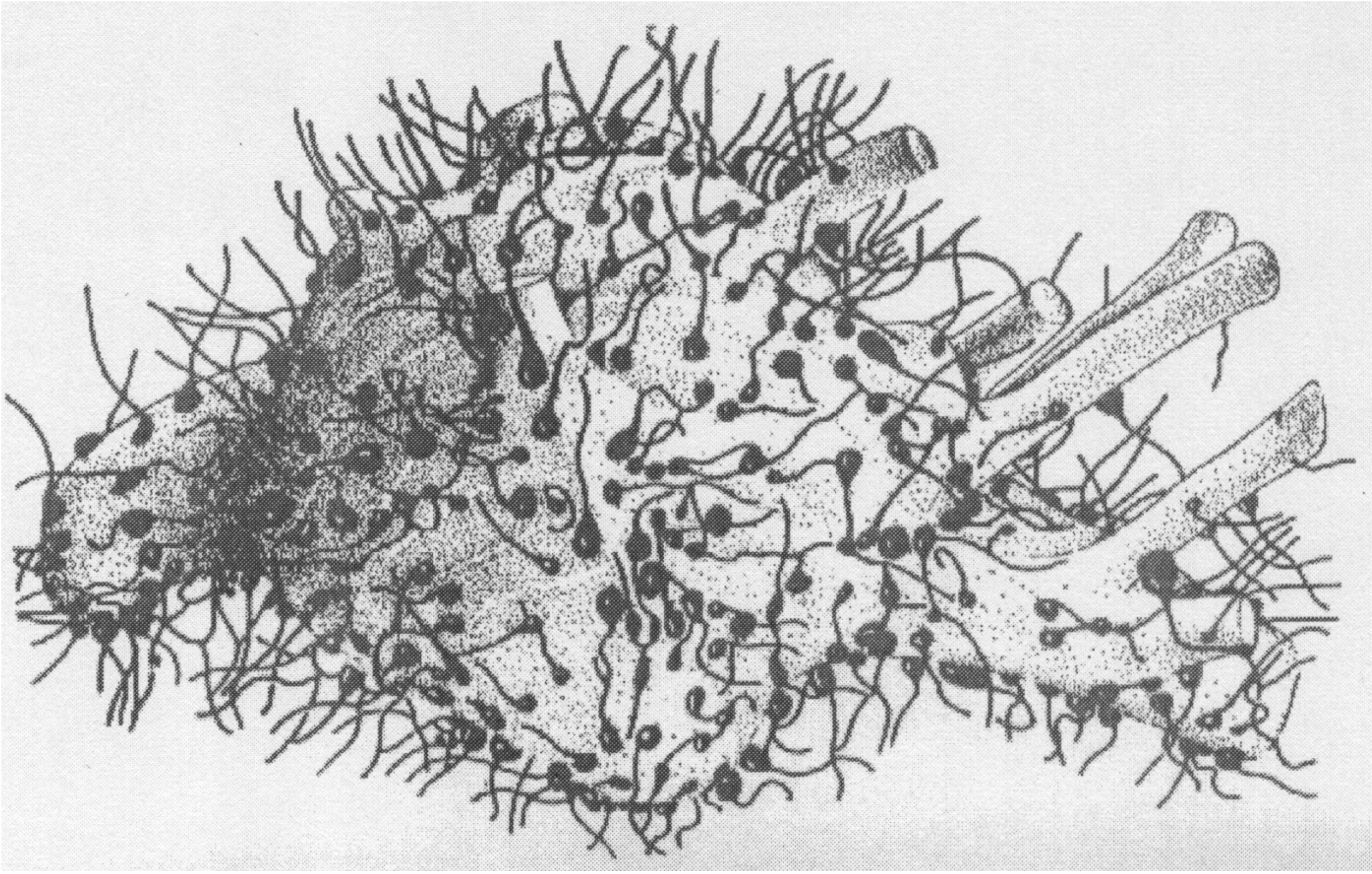
Neural Nets and Symbolic Reasoning

Learning

Outline

- Learning weights for single neurons
  - Hebbian rule
  - Generalized Hebbian rule
  - The perceptron training rule
  - The delta rule
- Multilayer networks
  - Various types of network architecture
  - A network for XOR
  - Multilayer networks and backpropagation

1 Learning weights for single neurons



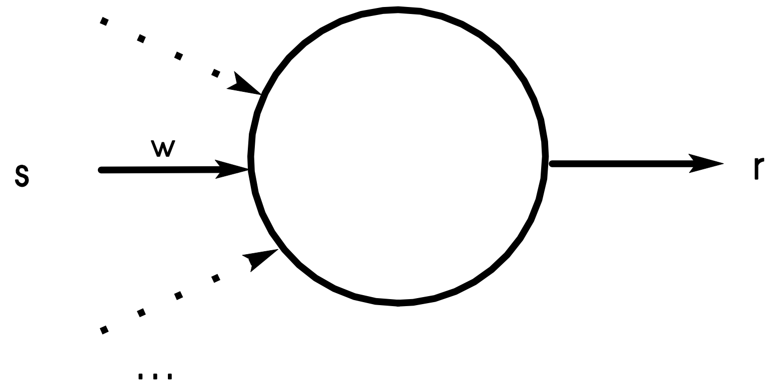
When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency as one of the cells firing B, is increased. (Hebb 1949, p. 62)

- Simultaneous activation of an input *s* and output *r* of a cell increases the corresponding weight *w*. (Unsupervised Learning!)

$S = \{0, 1\}$ possible activations

$w_{n+1} = w_n + \Delta w$

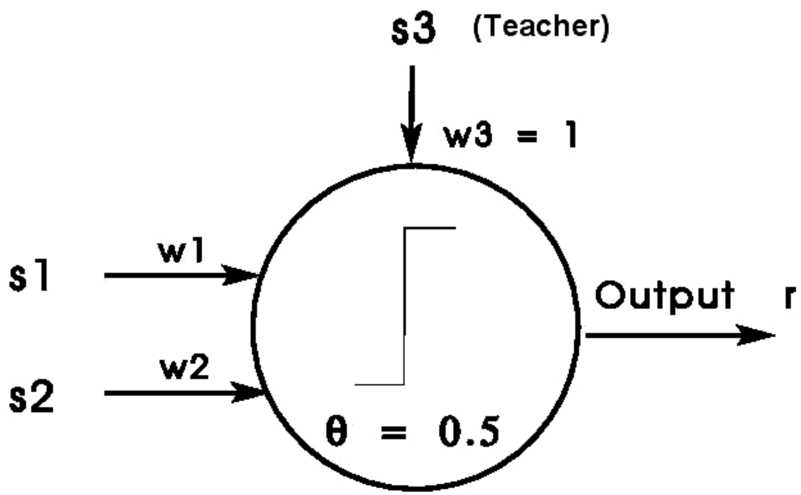
$$\Delta w = \begin{cases} \eta & \text{if } s=r=1 \\ 0 & \text{otherwise} \end{cases}$$



η is a positive constant called *learning rate*

Initial state: $\theta = 0.5$; w1 = w2 = 0

s1	s2	s3	r
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0



Teaching: $\eta = 0.3$

s1	s2	s3	r	w1	w2
0	0	0	0	0	0
1	0	1	1	0.3	0
0	1	1	1	0.3	0.3
1	1	1	1	0.6	0.6

Final state

$\theta = 0.5$; w1 = w2 = 0.6

s1	s2	s3	r
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	1

- Plasticity: ability of adaptation
- Stability: ability to preserve the learned information
- The two factors are (often) conflicting.

In the case under discussion, the system is of **low plasticity**. After some learning steps the system is saturated: The weights get maximum value and cannot further be increased - no way to learn new functions.

With regard to **stability** the system works well if the learning constant η is not too big and learning is stopped at some point.
Effect of **over-learning**: weights continue to change though the system has already learned the required function). Information can be lost!
(check it out by teaching the AND function)

$\Delta\mathbf{w} = \eta\cdot\mathbf{s}\cdot\mathrm{r}$ (**bold symbols** for vectors!)

- $S = \{0, 1\} \Rightarrow$ simple Hebbian rule
- $S = \{-1, 1\} \Rightarrow$ generalized rule; negative learning

Plasticity: if we take negative activations into account, then we find negative learning, that means negative correlations ate likewise reinforced (decreasing the corresponding weight factor). This has positive effects for plasticity: New functions can be learned. (example: Learn first the AND function and then the OR function)

Stability: Over-learning still possible but without such catastrophic consequences as in the case before (check it out by teaching the AND function).

$\Delta\mathbf{w} = \eta\cdot(t-r)\cdot\mathbf{s}$ *t* target output (teacher), *r* generated output, s input

S discrete (e.g. $S = \{-1, 1\}$)

The corresponding learning procedure can be proven to converge within a finite number of applications of the training rule to a weight vector that correctly classifies all training examples*, provided the training examples are linearly separable* and provided a sufficiently small η is used.

If the data are not linearly separable, convergence is not guaranteed! (check it out by using the exclusive OR example)

Initial state: $\theta = 0.5$; $w1 = w2 = 0$

s1	s2	r	t (teacher)
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	1

Teaching

s1	s2	r	t	w1	w2
0	0	0	0	0	0
1	0	0	1	0.3	0
0	1	0	1	0.3	0.3
1	1	1	1	0.3	0.3
0	0	0	0	0.3	0.3
1	0	0	1	0.6	0.3
0	1	0	1	0.6	0.6
1	1	1	1	0.6	0.6
0	0	0	0	0.6	0.6
1	0	1	1	0.6	0.6
0	1	1	1	0.6	0.6
1	1	1	1	0.6	0.6

Plasticity: optimal (perceptron convergence theorem!)

Stability: Over-learning not possible. Learning stops when the differences between wanted and generated output are zero.

The delta rule converges toward a best-fit approximation to the target concept if the training examples are not linearly separable.

Key idea: gradient descent as downward path on the error surface to search the find weight vector that best fits the target concept.

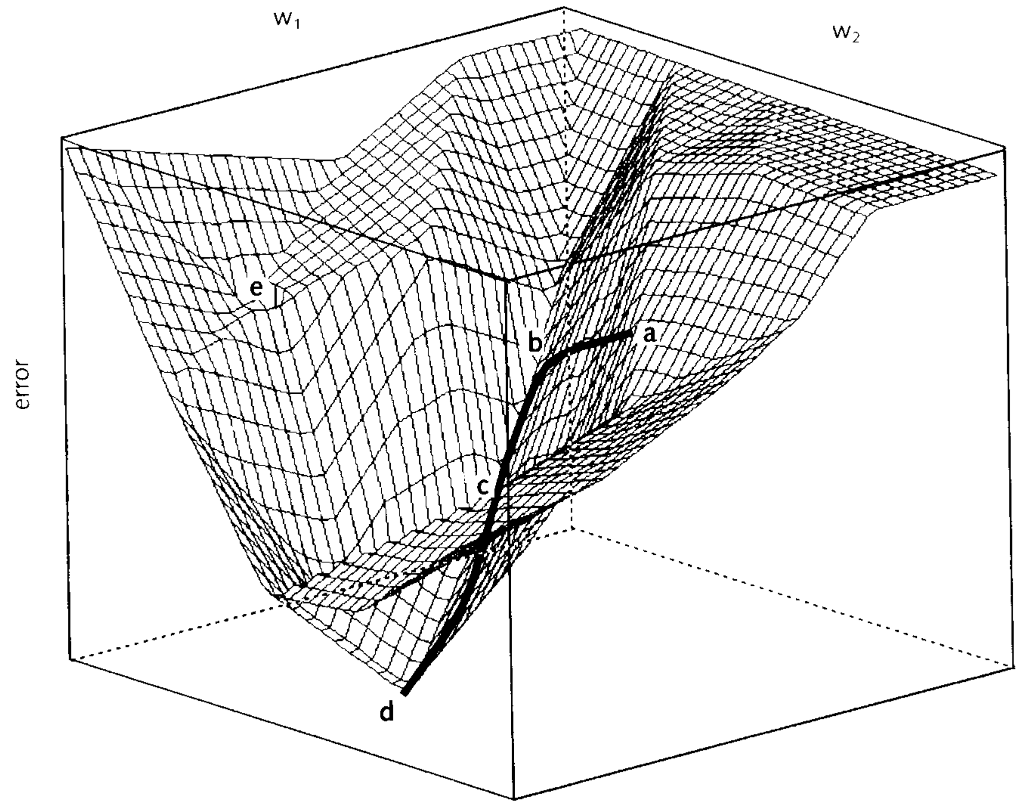


Figure 3.3 A hypothetical error surface for a neural network with two weights which represents gradient descent as downward paths on the error surface. The line from **a** to **d** shows one path of gradient descent to the global minimum, **d**. There is also a local minimum at **e**. Reprinted with permission from Elman (1993).

Defining the error

$\mathrm{E}(\mathbf{w}) = \frac{1}{2} \sum_{d \in D} (t_d - r_d)^2$

Untresholded perceptron

$r(\mathbf{s}) = \mathbf{w} \cdot \mathbf{s}$ (i.e., $r = \sum_j w_j s_j$)

Gradient of E

$$\nabla E(w) = [\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}]$$

Training rule

$\mathbf{w}^{n+1} = \mathbf{w}^{n} + \Delta\mathbf{w}$

$\Delta\mathbf{w} = -\eta \cdot \nabla E(\mathbf{w})$

Calculating $\nabla E(\mathbf{w})$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - r_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - r_d)^2$$

$$= \sum_{d \in D} (t_d - r_d) \frac{\partial}{\partial w_i} (t_d - \sum_j w_j \cdot s_{jd})$$

$$= \sum_{d \in D} (t_d - r_d)(-s_{id})$$

$\Delta\mathbf{w} = \eta \sum_{d \in D} (t_d - r_d)\, \mathbf{s}_d$

Difference to the perceptron training rule: summation over all learning items

Tresholded perceptron

$r(\mathbf{s}) = f(\mathbf{w} \cdot \mathbf{s})$ (i.e., $r = f(\sum_j w_j s_j)$)

with sigmoid function

$f(net) = 1/(1+\exp(-net/T))$

first derivation of the sigmoid function

$f'(net) = f(net) \cdot (1-f(net))$

Use the gradient method with $E(\mathbf{w}) = ½ \sum_{d \in D} (t_d - r_d)^2$ and calculate the corresponding learning rule:

$\Delta\mathbf{w} = ??$

The delta rule for tresholded perceptrons

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - r_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - r_d)^2$$

$$= \sum_{d \in D} (t_d - r_d) \frac{\partial}{\partial w_i} (t_d - f(\sum_j w_j \cdot s_{jd}))$$

$$= \sum_{d \in D} (t_d - r_d)(-f'(net) \cdot s_{id})$$

$$\Delta \mathbf{w} = \eta \sum_{d \in D} (t_d - r_d) \cdot f'(net) \cdot \mathbf{s}_d$$

$$= \eta \sum_{d \in D} (t_d - r_d) \cdot r_d \cdot (1 - r_d) \cdot \mathbf{s}_d$$

Stochastic approximation to gradient descent

Whereas the gradient descent training rule $\Delta\mathbf{w} = \eta \sum_{d \in D} (t_d - r_d)\, \mathbf{s}_d$ computes weights after summing over *all* the training examples in D, the stochastic approximation method approximates the gradient descend by updating weights incrementally, following the calculation of the error for *each* individual example.

Defining the error (with regard to an individual training example d)

$E^d(\mathbf{w}) = ½\, (t_d - r_d)^2$

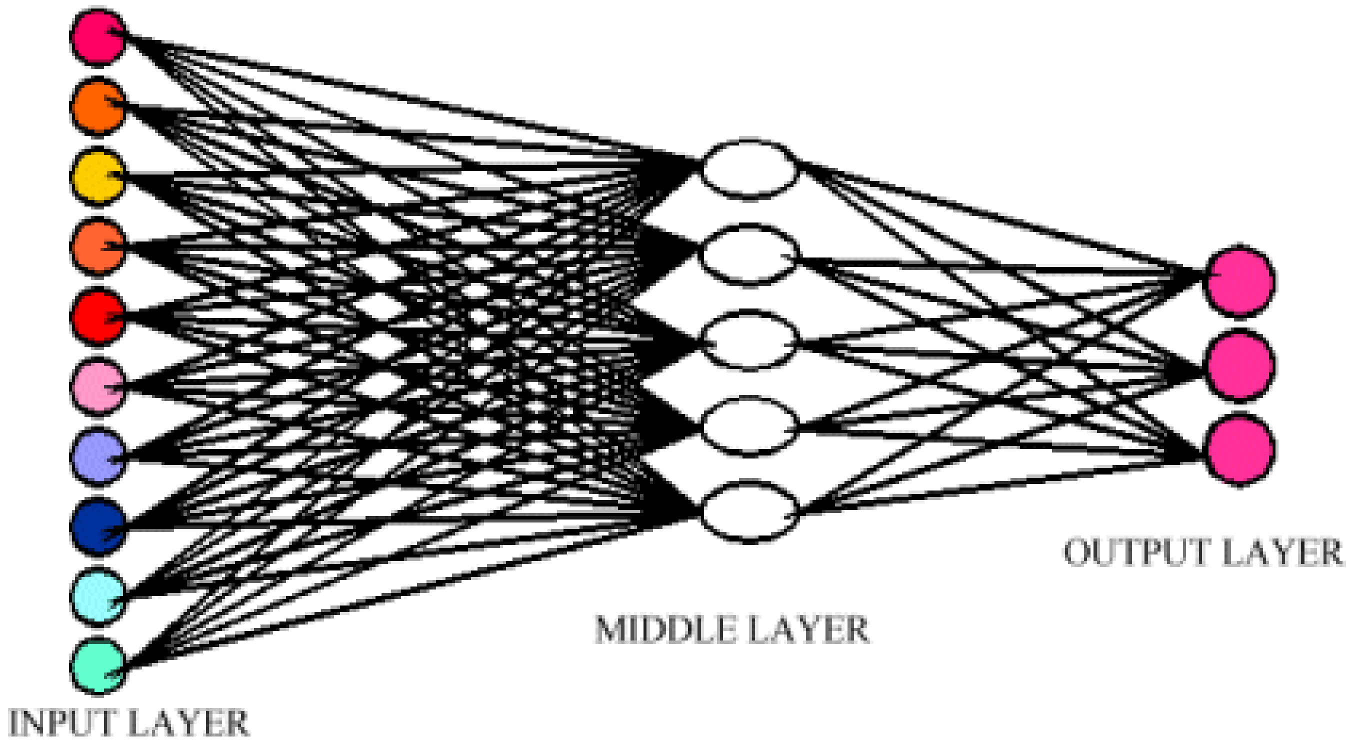
Training rule

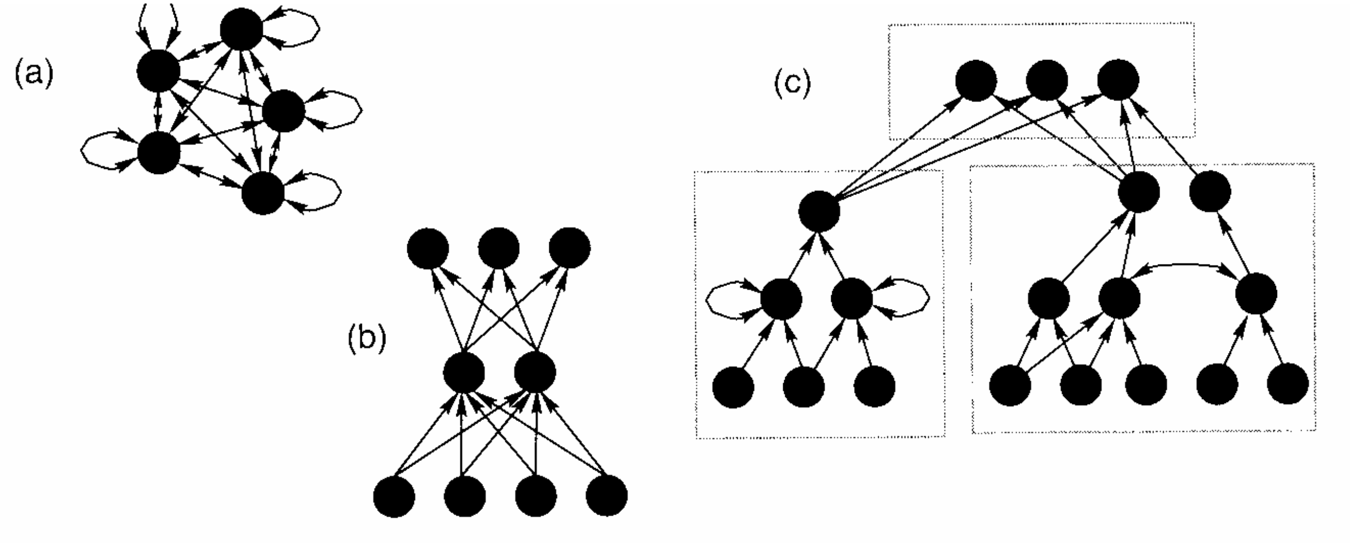
$\Delta\mathbf{w}^{\mathbf{d}} = -\eta \cdot \nabla E^d(\mathbf{w})$; $\Delta\mathbf{w}^{\mathbf{d}} = \eta\, (t_d - r_d) \cdot r_d \cdot (1\text{-}r_d) \cdot \mathbf{s}_d$

Advantages

- Faster convergence to a local minimum
- With appropriate learning rate η there is a good chance also to find the global minimum (if there are multiple local minima)

2 Multilayer networks



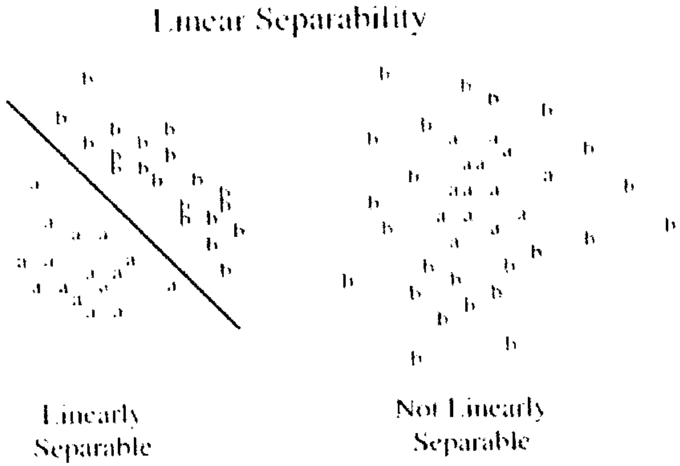


(a) A fully recurrent network

(b) A three layer feedforward network

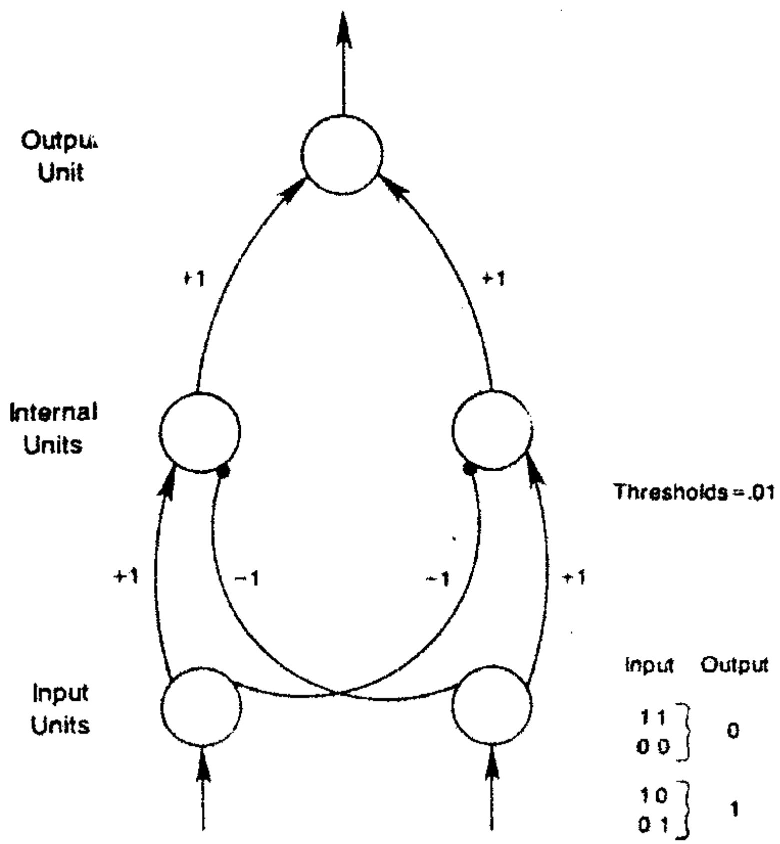
(c) A complex network consisting of several modules. Arrows indicate direction and flow of excitation or inhabitation

- Single perceptrons can only express linear decision surfaces
- Nonlinear activation functions are important: multiple layers of cascaded linear units still produce only linear functions.



- Importance of the sigmoid function: a unit very much like a perceptron (at least for small T), but based on a smoothed, differentiable threshold function.

Recognizing connectedness



Feedforward network with two hidden units and an output union.

The layer of hidden (internal) units form "internal representations" of the input pattern.

How to adjust the weights for the hidden units?

Backpropagation

Given a feedforward net containing two layers of sigmoid units. For each <**s**, **t**> in training examples, do the following:

Propagate the input forward through the network:

1. Input the instance s to the network and compute the output r_x of every unit x in the network.

Propagate the errors back through the network:

2. For each network output unit i, calculate its error term δ_i

   $\delta_i \leftarrow r_i(1-r_i)(t_i-r_i)$

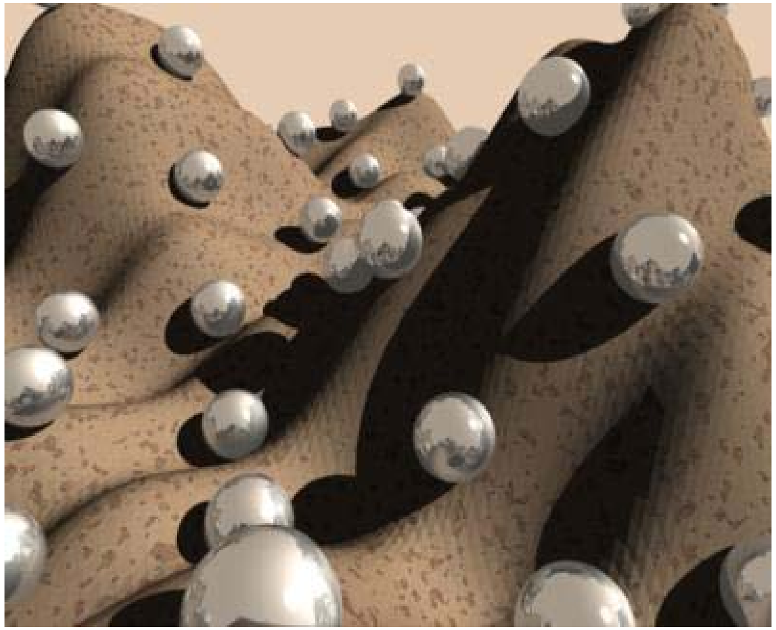
3. For each hidden unit i, calculate each error term δ

   $\delta_i \leftarrow r_i(1-r_i)\Sigma_{k\in outputs}(w_{ki}\ \delta_k)$

4. Update each network weight $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$

   where $\Delta w_{ij} = \eta\ \delta_i\ s_{ij}$

Backpropagation is a widely used algorithm, and many variations have been developed. The most common is to make the weight update on the *n*th iteration dependent on the update that occurred during the (*n-1*)th iteration.

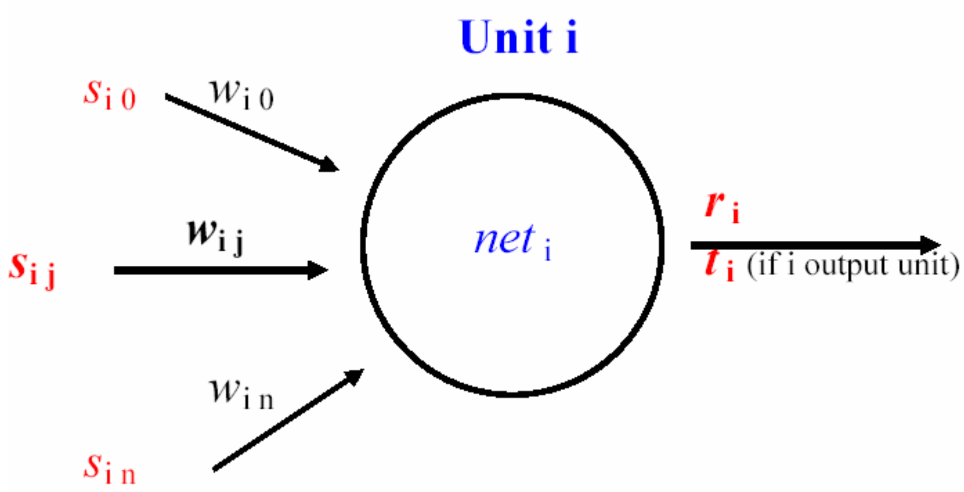


$$\Delta w_{ij}(n) = \eta\ \delta_i\ s_{ij} + \alpha\ \Delta w_{ij}(n-1)$$

The constant $0 \leq \alpha \leq 1$ is called *momentum.*

The momentum term has the same effect as adding momentum to a ball rolling down the error surface. This can have the effect of gradually increasing the step size in search regions where the gradient in unchanging, or of overcoming small local minima.

Deriving the stochastic gradient descend rule

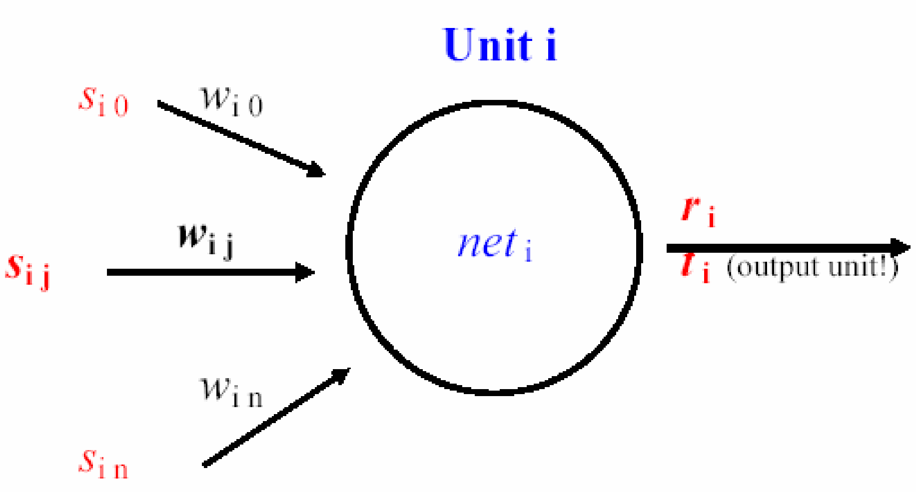


$\Delta w_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}}$, where $E_d(\boldsymbol{w}) = \frac{1}{2} \sum_{k \in outputs}(t_k - r_k)^2$

Skip index *d* in the following and use $r_i = f(net_i)$; $net_i = \sum_k w_{ik}\, s_{ik}$

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta \frac{\partial E}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_{ij}} = -\eta \frac{\partial E}{\partial net_i} \cdot s_{ij} = \eta \cdot \delta_i \cdot s_{ij}$$

where $\delta_i = -\frac{\partial E}{\partial net_i}$



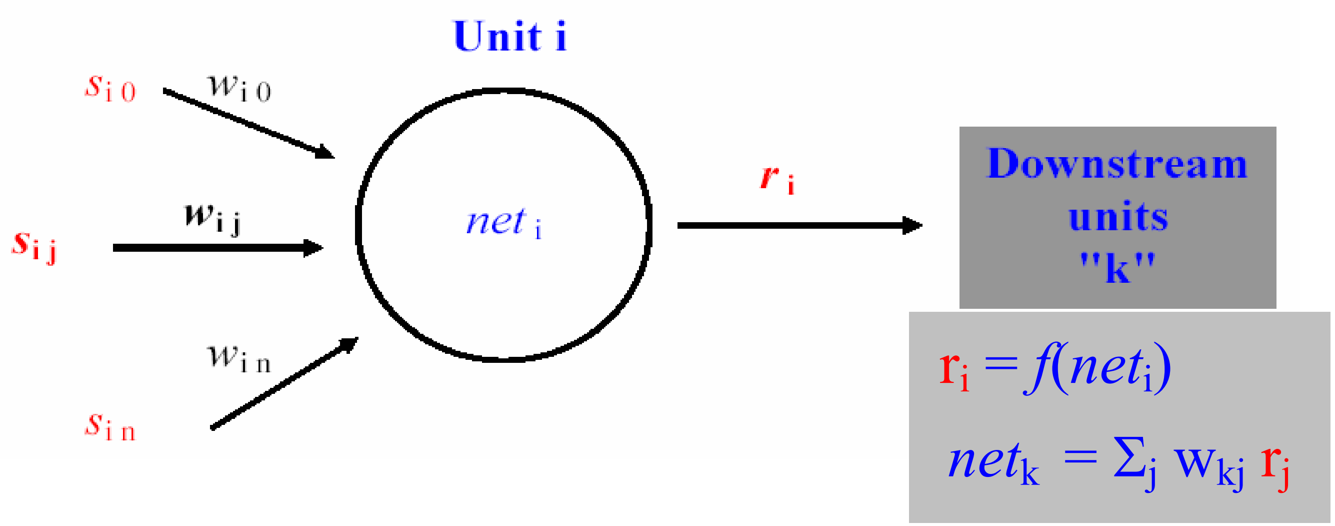
$r_i = f(net_i)$, f sigmoid function

$net_i = \sum_k w_{ik}\, s_{ik}$

$$\delta_i = -\frac{\partial E}{\partial net_i} = -1/2 \frac{\partial}{\partial net_i} \sum_{k \in output} (t_k - r_k)^2 = (t_i - r_i) \cdot \frac{\partial}{\partial net_i} r_i(net_i) = (t_i - r_i) \cdot r_i \cdot (1 - r_i)$$

That is exactly what we used in step 2 of the back propagation algorithm:

$\delta_i \leftarrow r_i(1\text{-}r_i)(t_i\text{-}r_i)$



net_i can influence E (via the network outputs) only through the units in downstream(i)!

$$\delta_i = -\frac{\partial E}{\partial net_i} = -\sum_{k \in downstream(i)} \frac{\partial E}{\partial net_k}\frac{\partial net_k}{\partial net_i} = \sum_{k \in downstream(i)} \delta_k \frac{\partial net_k}{\partial net_i}$$

$$= \sum_{k \in downstream(i)} \delta_k \frac{\partial net_k}{\partial r_i}\frac{\partial r_i}{\partial net_i} = \sum_{k \in downstream(i)} \delta_k w_{ki} r_i (1 - r_i)$$

That is exactly what we used in step 3 of the back propagation algorithm:

$\delta_i \leftarrow r_i(1-r_i)\Sigma_{k \in downstream(i)} (w_{ki}\, \delta_k)$

Representational power of feedforward networks

- *Boolean functions*: Every Boolean function can be represented exactly by some network with two layers of units. The number of hidden units may grow exponentially in the worst case with the number of network inputs.
- *Continuous functions*: Every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units.
- *Arbitrary functions*: Any function can be approximated to arbitrary accuracy by network with three layers of units. (for details see Mitchell's "machine learning", p. 105.