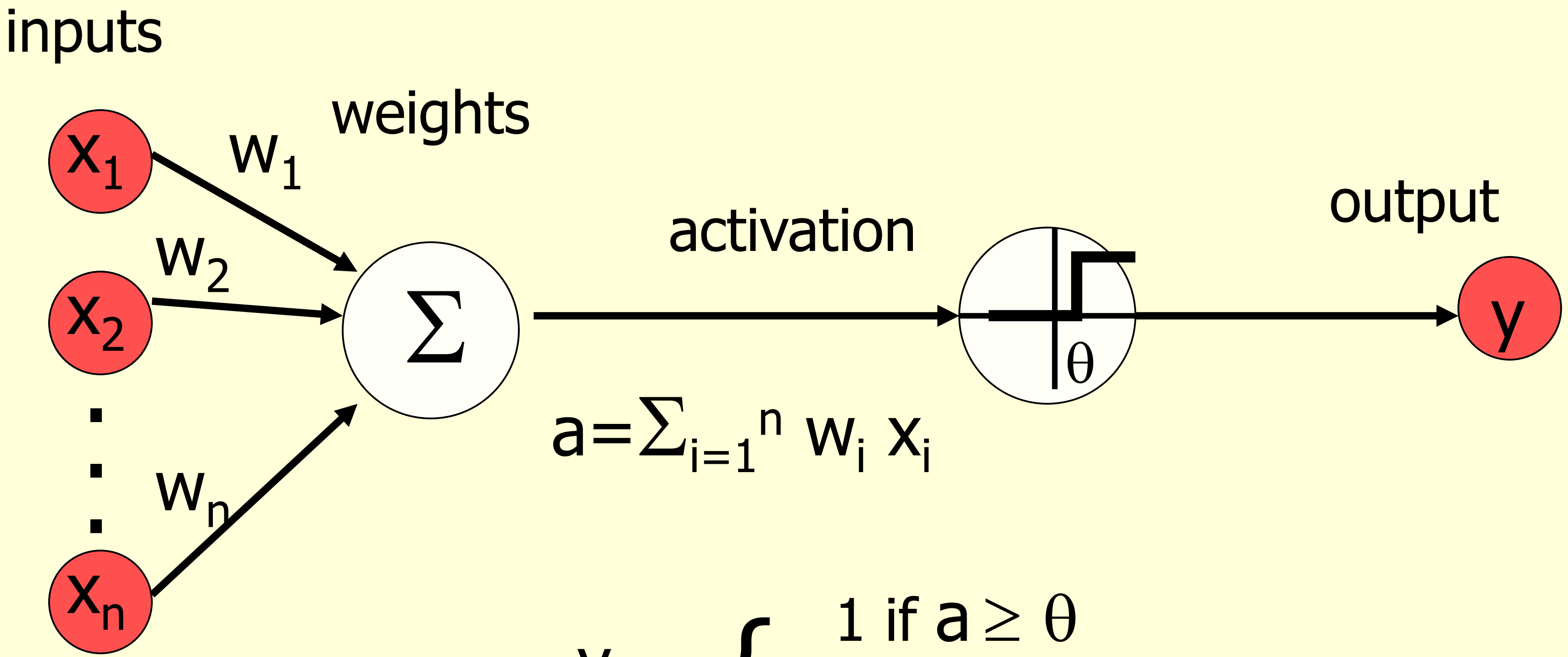
Multi Layer Perceptron

Threshold Logic Unit (TLU)

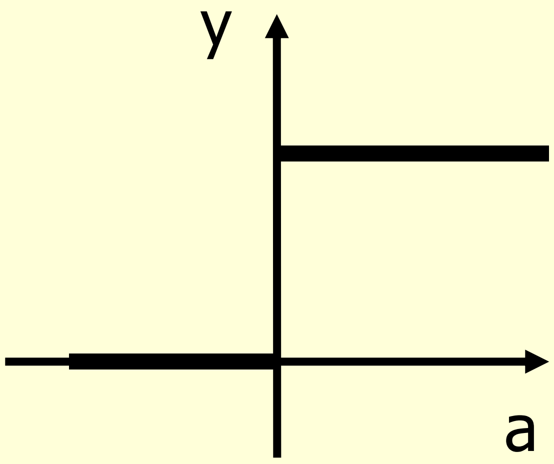


$a = \sum_{i=1}^{n} w_i x_i$

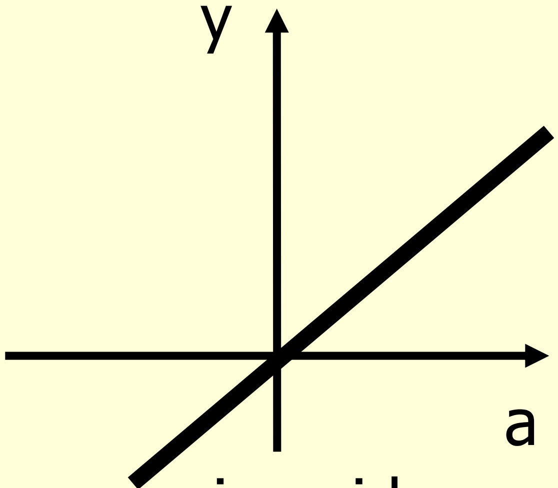
$$y = \begin{cases} 1 \text{ if } a \geq \theta \\ 0 \text{ if } a < \theta \end{cases}$$

Activation Functions

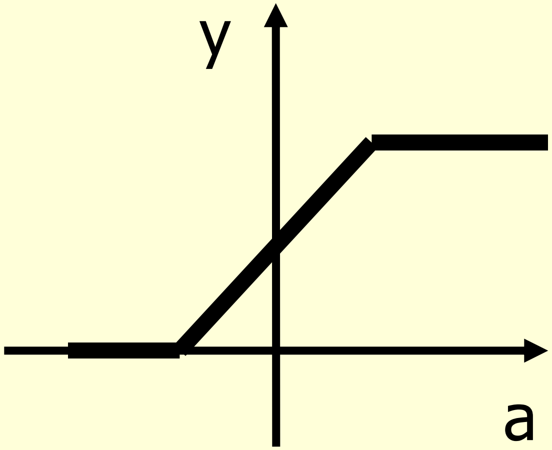
threshold



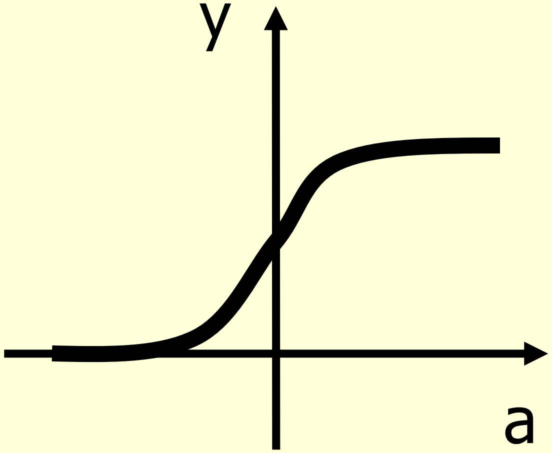
linear



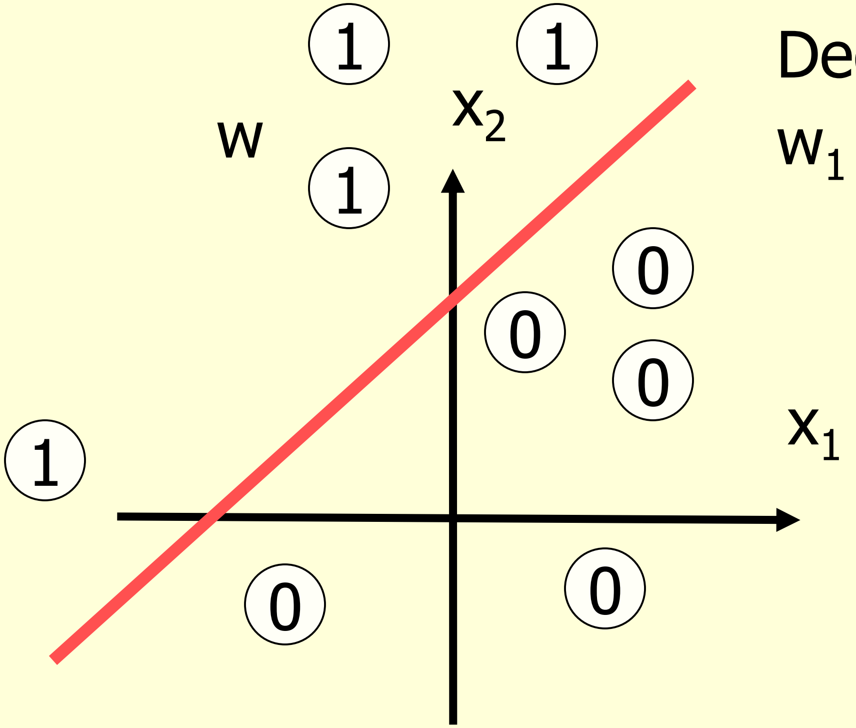
piece-wise linear



sigmoid



Decision Surface of a TLU

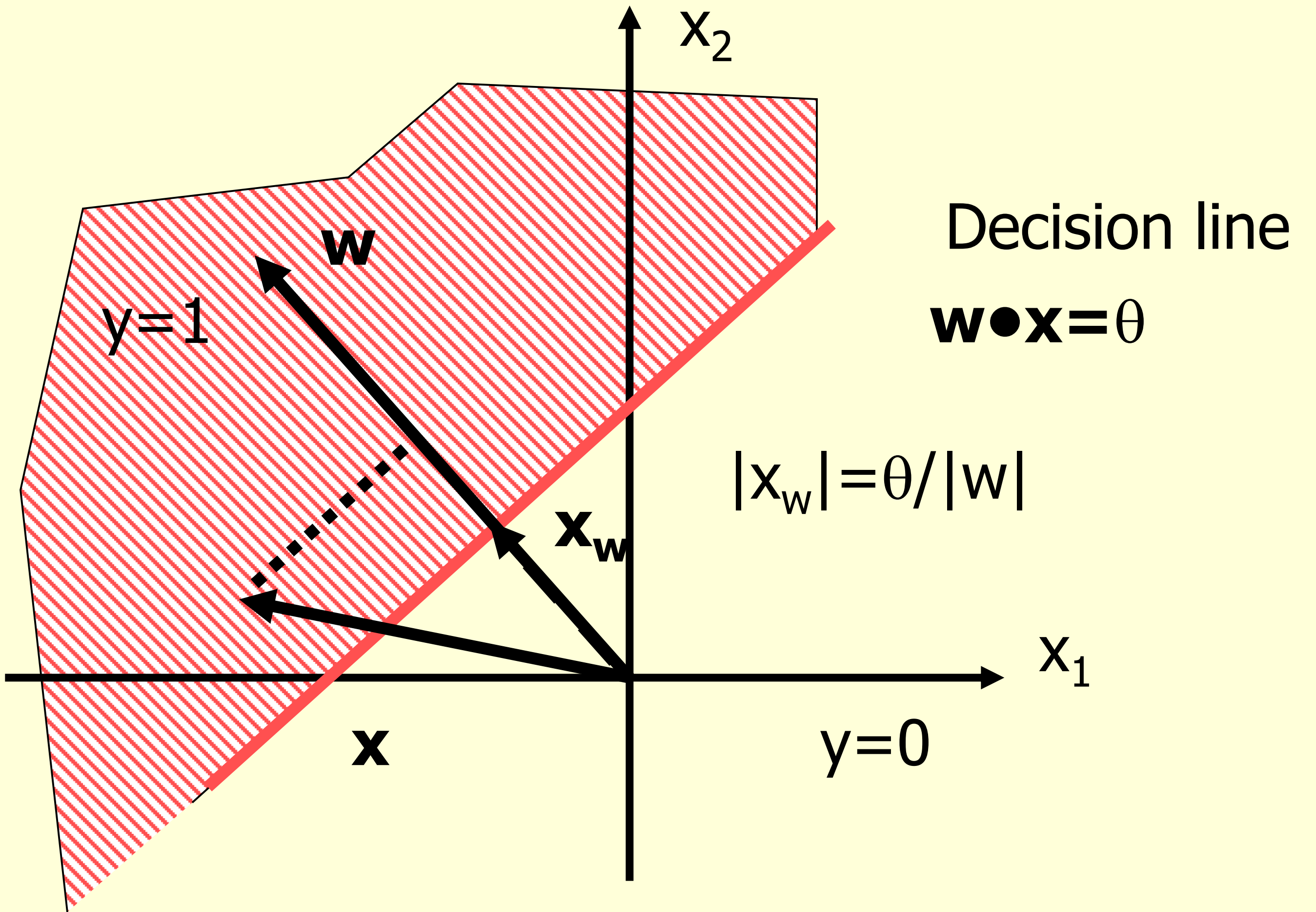


Decision line

$w_1 x_1 + w_2 x_2 = \theta$

Geometric Interpretation

The relation $\mathbf{w}\bullet\mathbf{x}=\theta$ defines the decision line

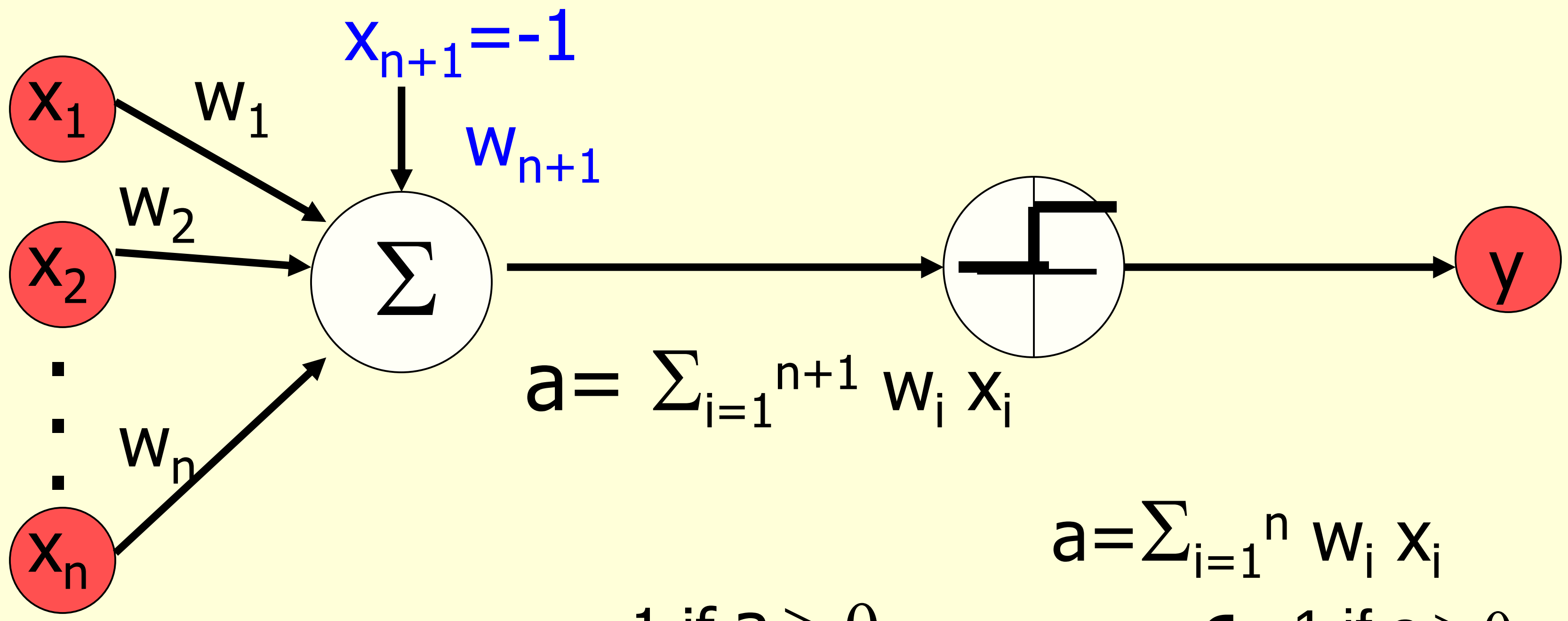


Geometric Interpretation

- In n dimensions the relation **w•x**=θ defines a n-1 dimensional hyper-plane, which is perpendicular to the weight vector **w**.
- On one side of the hyper-plane (**w•x**>θ) all patterns are classified by the TLU as "1", while those that get classified as "0" lie on the other side of the hyper-plane.
- If patterns can be not separated by a hyper-plane then they cannot be correctly classified with a TLU.

Threshold as Weight

$\theta = w_{n+1}$



$a = \sum_{i=1}^{n+1} w_i x_i$

$a = \sum_{i=1}^{n} w_i x_i$

$$y = \begin{cases} 1 \text{ if } a \geq \theta \\ 0 \text{ if } a < \theta \end{cases} \rightarrow y = \begin{cases} 1 \text{ if } a \geq 0 \\ 0 \text{ if } a < 0 \end{cases}$$

Training ANNs

- Training set S of examples {**x**,**t**}
  - **x** is an input vector and
  - **t** the desired target vector
  - Example: Logical And

    S = {(0,0),0}, {(0,1),0}, {(1,0),0}, {(1,1),1}
- Iterative process
  - Present a training example x , compute network output y , compare output y with target t, adjust weights and thresholds
- Learning rule
  - Specifies how to change the weights w and thresholds θ of the network as a function of the inputs x, output y and target t.

Perceptron Learning Rule

- $\mathbf{w'} = \mathbf{w} + \alpha\,(t-y)\,\mathbf{x}$

Or in components

- $w'_i = w_i + \Delta w_i = w_i + \alpha\,(t-y)\,x_i \quad (i=1..n+1)$

With $w_{n+1} = \theta$ and $x_{n+1}=-1$

- The parameter α is called the *learning rate*. It determines the magnitude of weight updates Δw_i .
- If the output is correct ($t=y$) the weights are not changed ($\Delta w_i = 0$).
- If the output is incorrect ($t \neq y$) the weights w_i are changed such that the output of the TLU for the new weights w'_i is *closer/further* to the input x_i.

Perceptron Training Algorithm

Repeat

for each training vector pair ($\mathbf{x}$,t)

evaluate the output y when $\mathbf{x}$ is the input

if $y \neq t$ then

form a new weight vector $\mathbf{w'}$ according

to $\mathbf{w'} = \mathbf{w} + \alpha\,(t-y)\,\mathbf{x}$

else

do nothing

end if

end for

Until y=t for all training vector pairs

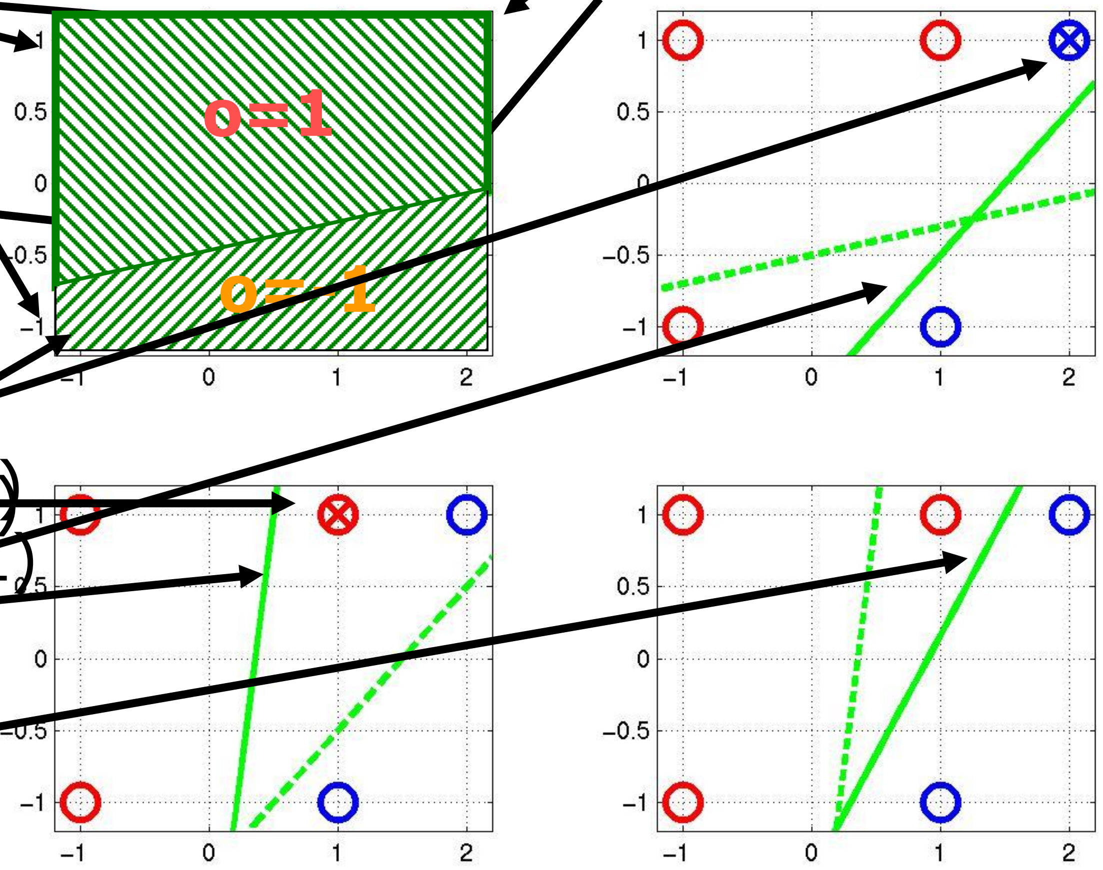
Perceptron Learning Rule

t=1

t=-1

w=[0.25 –0.1 0.5]

$x_2 = 0.2\ x_1 - 0.5$



(x,t)=([2,1],-1)

(x,t)=([-1,-1],1)

o=sgn(0.45-0.6+0.3)

o=sgn(0.25+0.1-0.5)

(x,t)=([1,1],1)

=1

o=sgn(0.25-0.7+0.1)

=-1

Δw=[0.2 –0.2 –0.2]

Δw=[-0.2 –0.4 –0.2]

Δw=[0.2 0.2 0.2]

Perceptron Convergence Theorem

The algorithm converges to the correct classification

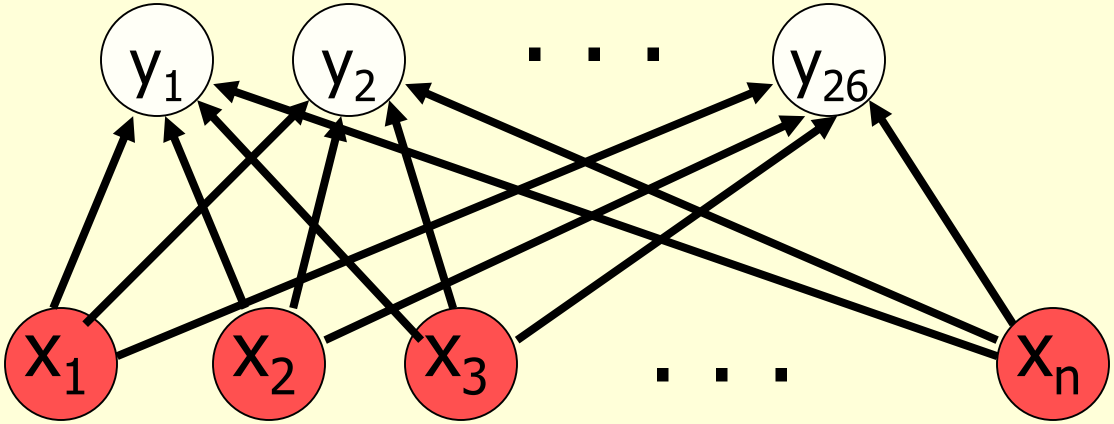
- if the training data is linearly separable
- and η is sufficiently small

- If two classes of vectors X_1 and X_2 are linearly separable, the application of the perceptron training algorithm will eventually result in a weight vector $\mathbf{w_0}$, such that $\mathbf{w_0}$ defines a TLU whose decision hyper-plane separates X_1 and X_2 (Rosenblatt 1962).
- Solution $\mathbf{w_0}$ is not unique, since if $\mathbf{w_0}\ \mathbf{x} = 0$ defines a hyper-plane, so does $\mathbf{w'_0} = k\ \mathbf{w_0}$.

Multiple TLUs

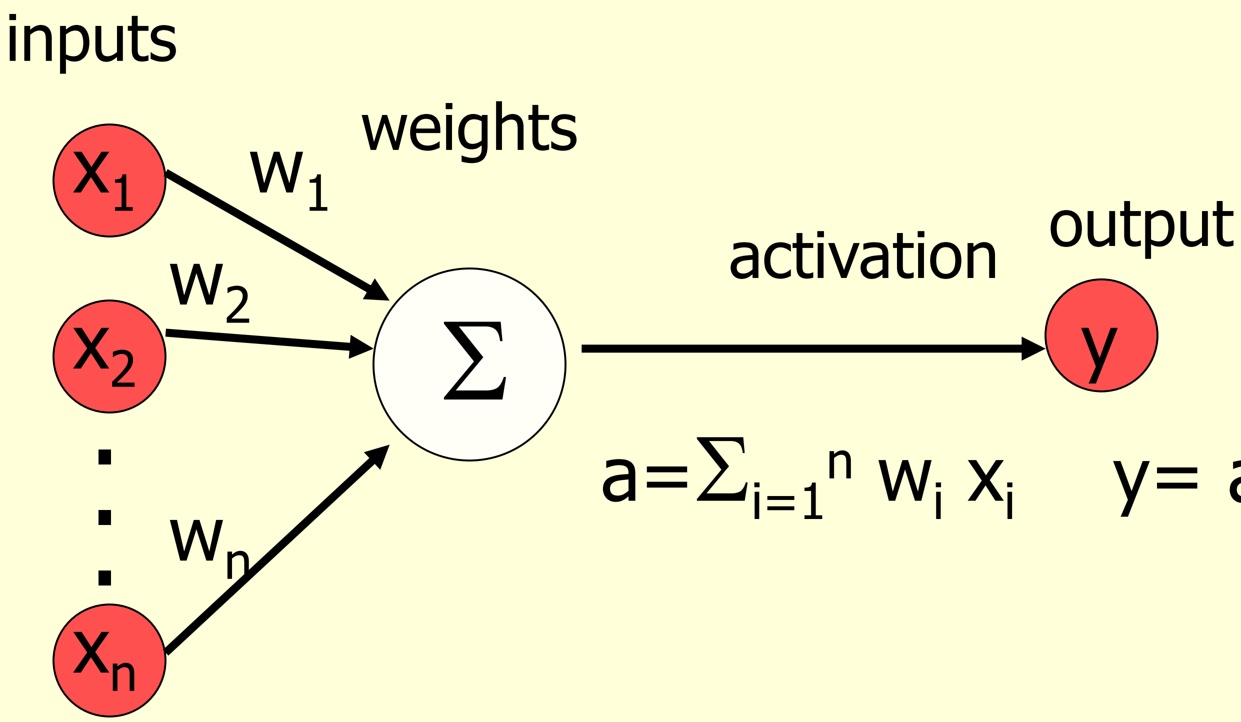
- Handwritten alphabetic character recognition
  - 26 classes : A,B,C…,Z
  - First TLU distinguishes between “A”s and “non-A”s, second TLU between “B”s and “non-B”s etc.

w_{ji} connects x_i with y_j



$$w'_{ji} = w_{ji} + \alpha\,(t_j - y_j)\,x_i$$

Linear Unit



$a=\sum_{i=1}^{n} w_i x_i$ $\quad$ $y= a = \sum_{i=1}^{n} w_i x_i$

Gradient Descent Learning Rule

- Consider linear unit without threshold and continuous output o (not just –1,1)
  - $o = w_0 + w_1 x_1 + \ldots + w_n x_n$
- Train the w_i's such that they minimize the squared error
  - $E[w_0, w_1, \ldots, w_n] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$

    where D is the set of training examples

Gradient Descent

$$E[w_0, w_1, \ldots, w_n] = \tfrac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$D=\{<(1,1),1>,<(-1,-1),1>,$
$<(1,-1),-1>,<(-1,1),-1>\}$

Gradient:

$\nabla E[w]=[\partial E/\partial w_0, \ldots \partial E/\partial w_n]$

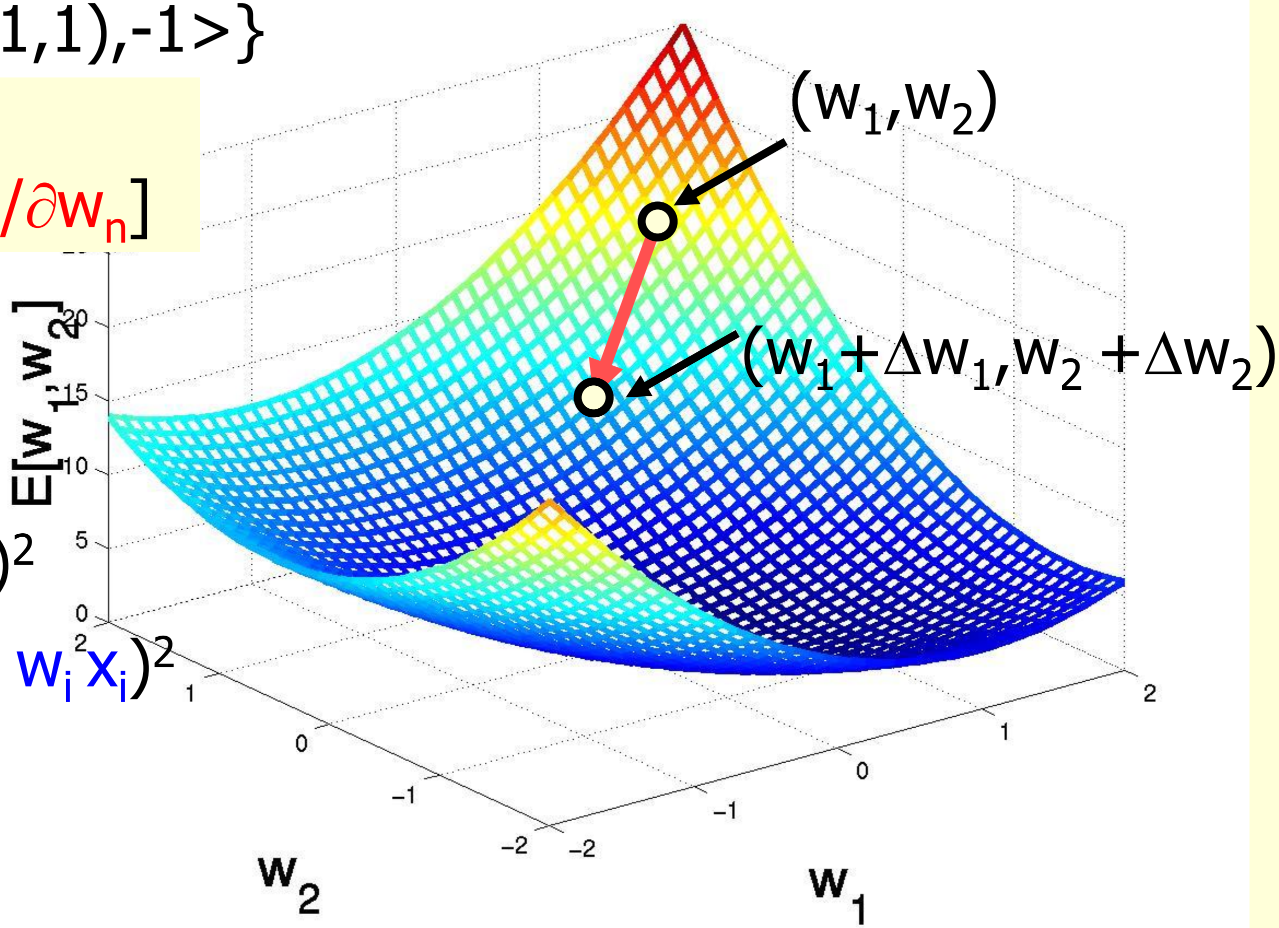
$\Delta w = -\eta \nabla E[w]$

$\Delta w_i = -\eta \, \partial E/\partial w_i$

$= \partial/\partial w_i \; 1/2 \sum_d (t_d - o_d)^2$

$= \partial/\partial w_i \; 1/2 \sum_d (t_d - \sum_i w_i x_i)^2$

$= \sum_d (t_d - o_d)(-x_i)$



Incremental Stochastic Gradient Descent

- Batch mode : gradient descent

  $w = w - \eta \nabla E_D[w]$ over the entire data D

  $E_D[w] = 1/2 \sum_d (t_d - o_d)^2$

- Incremental mode: gradient descent

  $w = w - \eta \nabla E_d[w]$ over individual training examples d

  $E_d[w] = 1/2\ (t_d - o_d)^2$

Incremental Gradient Descent can approximate Batch Gradient Descent arbitrarily closely if η is small enough

Perceptron vs. Gradient Descent Rule

- perceptron rule

  $w'_i = w_i + \alpha\ (t^p - y^p)\ x_i^p$

  derived from manipulation of decision surface.

- gradient descent rule

  $w'_i = w_i + \alpha\ (t^p - y^p)\ x_i^p$

  derived from minimization of error function

  $E[w_1,\ldots,w_n] = \frac{1}{2} \sum_p (t^p - y^p)^2$

  by means of gradient descent.

Where is the big difference?

Perceptron vs. Gradient Descent Rule

Perceptron learning rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

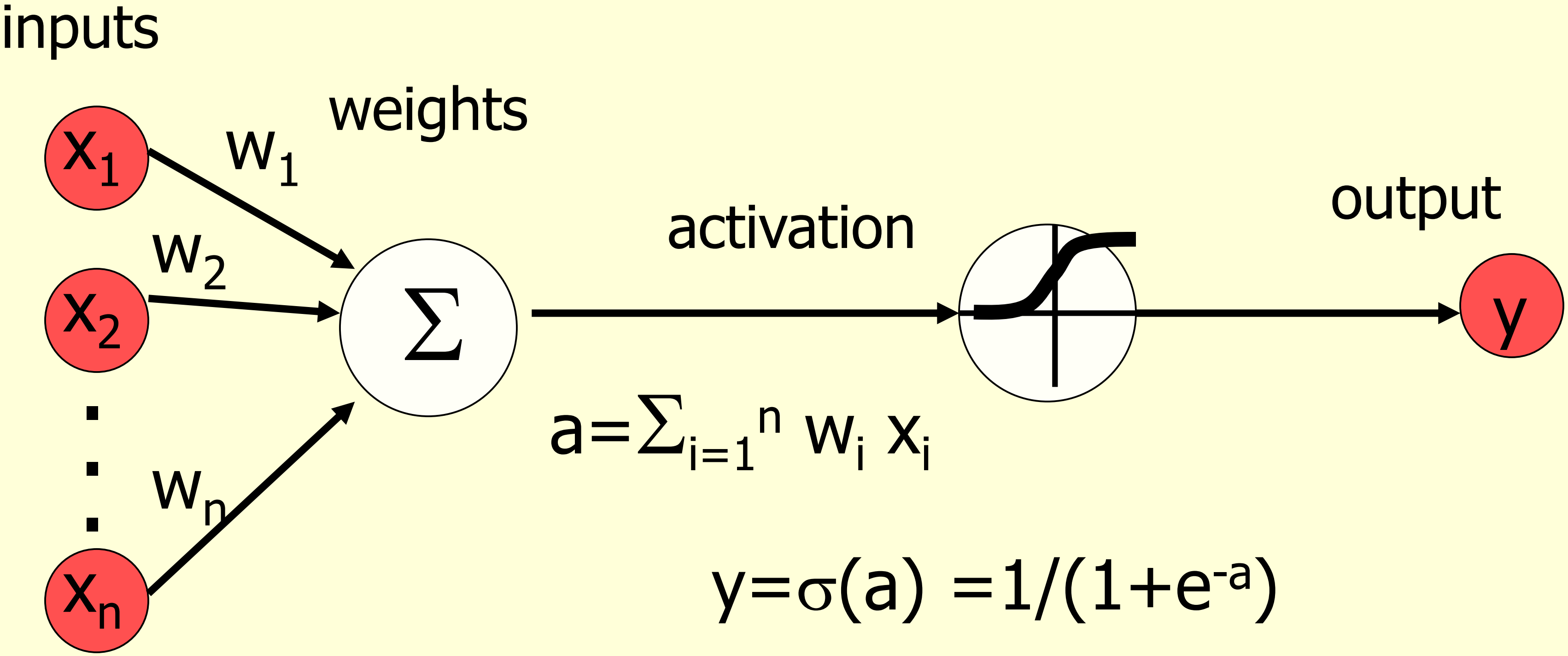
Linear unit training rules uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Presentation of Training Examples

- Presenting all training examples once to the ANN is called an *epoch*.
- In incremental stochastic gradient descent training examples can be presented in
  - Fixed order (1,2,3…,M)
  - Randomly permutated order (5,2,7,…,3)
  - Completely random (4,1,7,1,5,4,……)

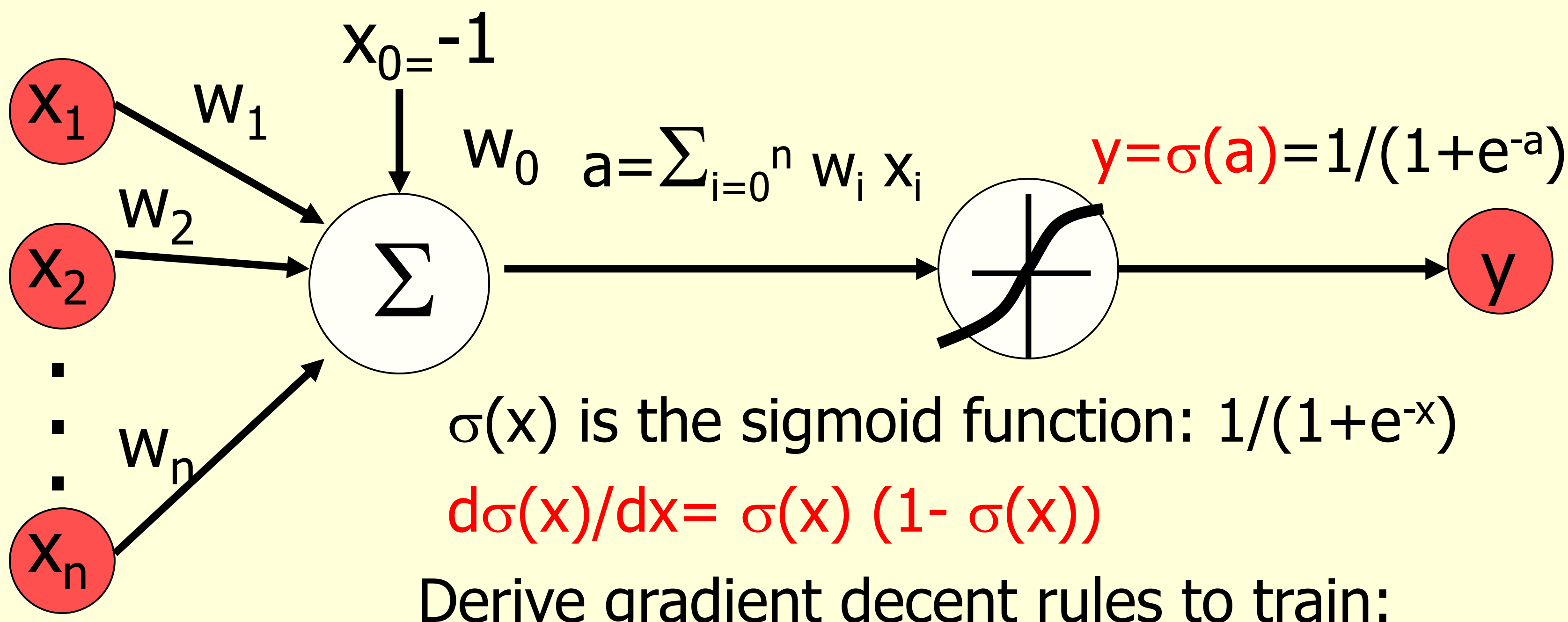
Neuron with Sigmoid-Function



$$a=\sum_{i=1}^{n} w_i x_i$$

$$y=\sigma(a) = 1/(1+e^{-a})$$

Sigmoid Unit



$\sigma(x)$ is the sigmoid function: $1/(1+e^{-x})$

$d\sigma(x)/dx = \sigma(x)\,(1-\sigma(x))$

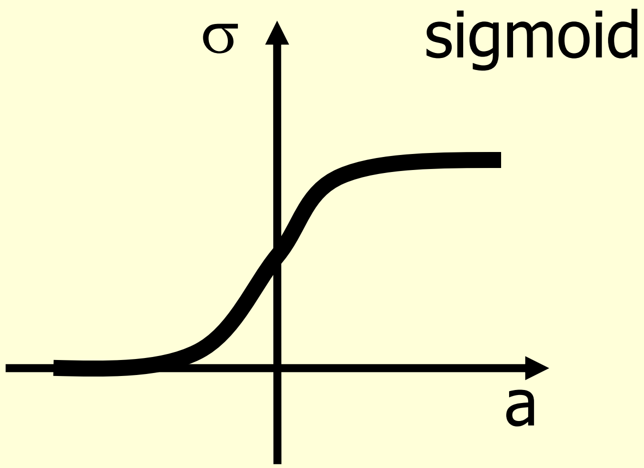
Derive gradient decent rules to train:

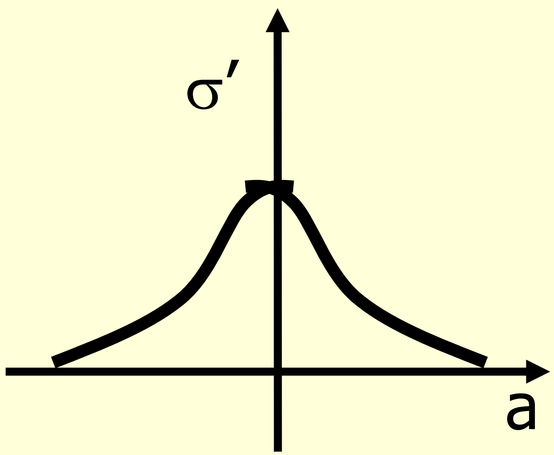
- one sigmoid function

  $\partial E/\partial w_i = -\sum_p (t^p-y)\, y\, (1-y)\, x_i^p$

- Multilayer networks of sigmoid units backpropagation:

Gradient Descent Rule for Sigmoid Output Function





$$E^p[w_0, w_1, \ldots, w_n] = \tfrac{1}{2}\,(t^p - y^p)^2$$

$$\partial E^p / \partial w_i = \partial / \partial w_i \; \underline{\tfrac{1}{2}\,(t^p - y^p)^2}$$

$$= \partial / \partial w_i \; \underline{\tfrac{1}{2}\,(t^p - \sigma(\textstyle\sum_i w_i\, x_i^p))^2}$$

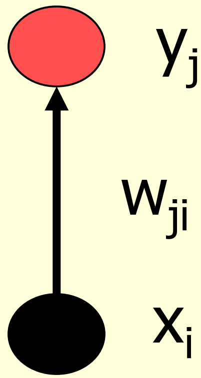
$$= \underline{(t^p - y^p)\; \sigma'(\textstyle\sum_i w_i\, x_i^p)\; (-x_i^p)}$$

for $y = \sigma(a) = 1/(1+e^{-a})$

$\sigma'(a) = e^{-a}/(1+e^{-a})^2 = \sigma(a)\,(1-\sigma(a))$

$$w'_i = w_i + \Delta w_i = w_i + \alpha\, y(1-y)(t^p - y^p)\, x_i^p$$

Gradient Descent Learning Rule



$$\Delta w_i = \alpha \;\; y_j^p(1-y_j^p) \;\; (t_j^p-y_j^p) \;\;\; x_i^p$$

learning rate

derivative of
activation function

error δ_j of
post-synaptic neuron

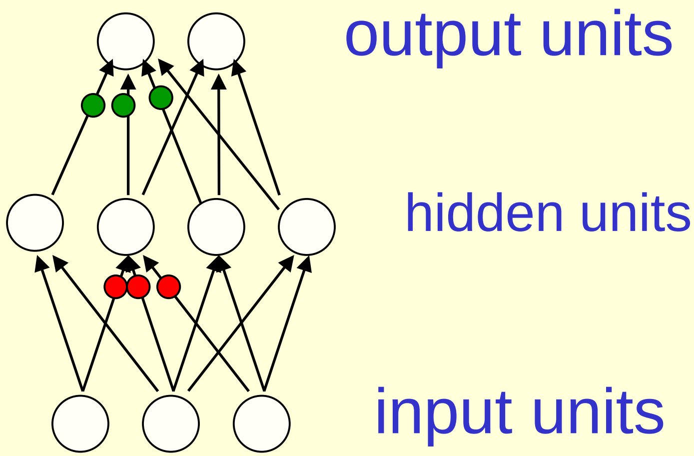
activation of
pre-synaptic neuron

Learning with hidden units

- Networks without hidden units are very limited in the input-output mappings they can model.
  - More layers of linear units do not help. Its still linear.
  - Fixed output non-linearities are not enough
- We need multiple layers of adaptive non-linear hidden units. This gives us a universal approximate. But how can we train such nets?
  - We need an efficient way of adapting all the weights, not just the last layer. This is hard. Learning the weights going into hidden units is equivalent to learning features.
  - Nobody is telling us directly what hidden units should do.

Learning by perturbing weights

- Randomly perturb one weight and see if it improves performance. If so, save the change.
  - Very inefficient. We need to do multiple forward passes on a representative set of training data just to change one weight.
  - Towards the end of learning, large weight perturbations will nearly always make things worse.
- We could randomly perturb all the weights in parallel and correlate the performance gain with the weight changes.
  - Not any better because we need lots of trials to "see" the effect of changing one weight through the noise created by all the others.

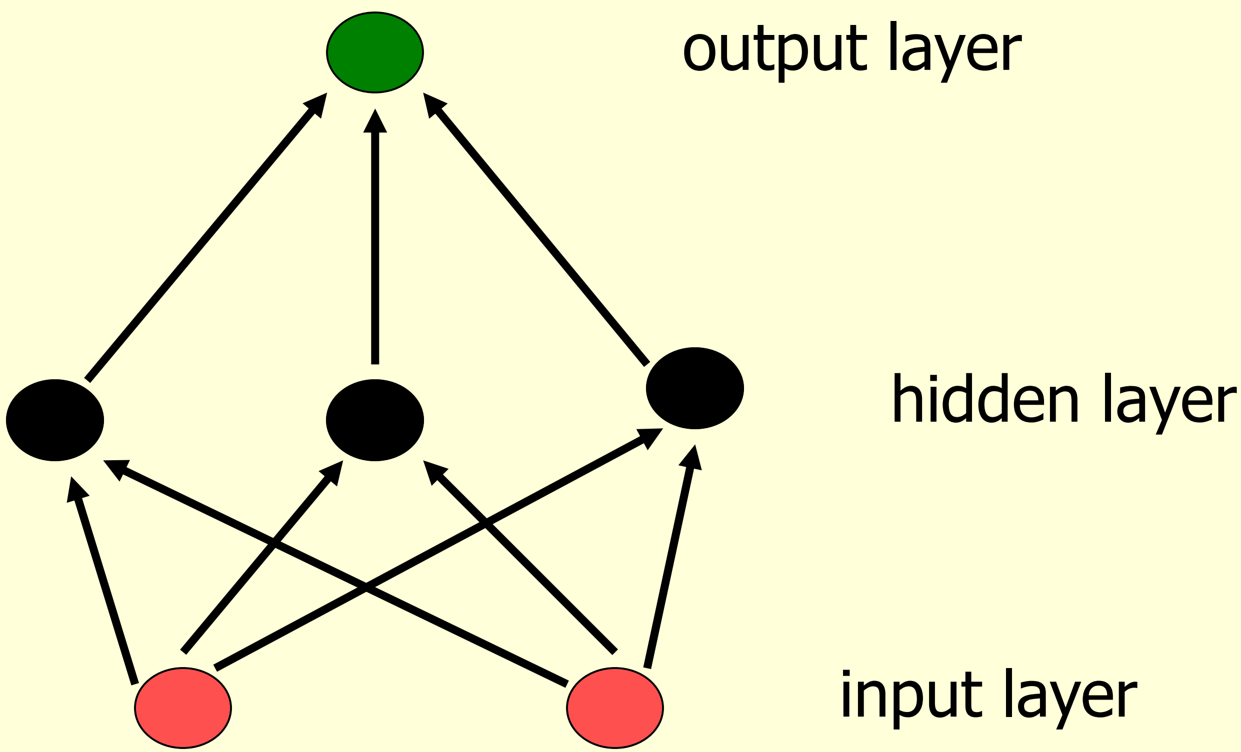


Learning the hidden to output weights is easy. Learning the input to hidden weights is hard.

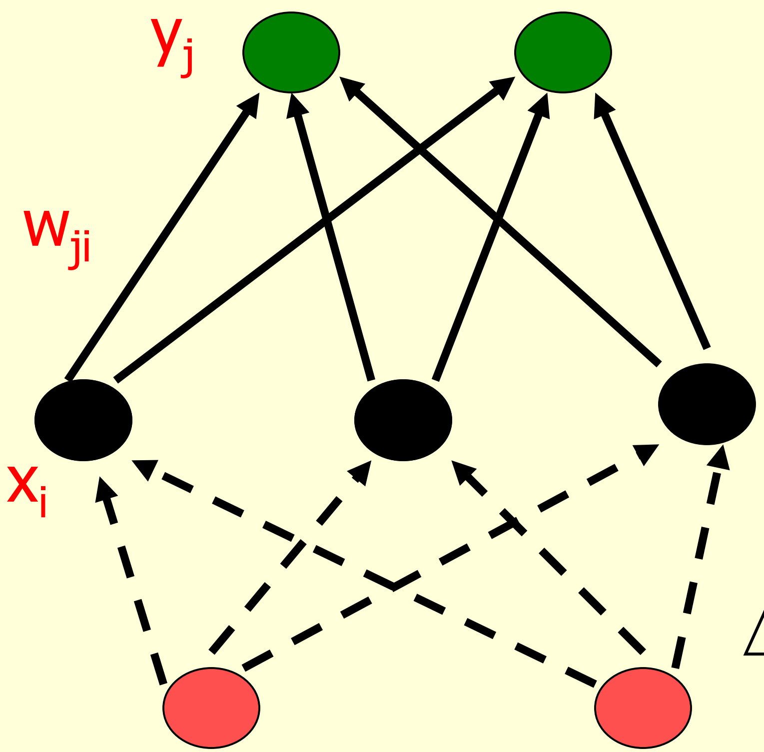
The idea behind backpropagation

- We don't know what the hidden units ought to do, but we can compute how fast the error changes as we change a hidden activity.
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities.
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.
  - We can compute error derivatives for all the hidden units efficiently.
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit.

Multi-Layer Networks



Training-Rule for Weights to the Output Layer



$$E^p[w_{ij}] = \tfrac{1}{2} \sum_j (t_j^p - y_j^p)^2$$

$$\partial E^p / \partial w_{ji} = \partial / \partial w_{ji} \; \tfrac{1}{2} \sum_j (t_j^p - y_j^p)^2$$

$$= \ldots$$

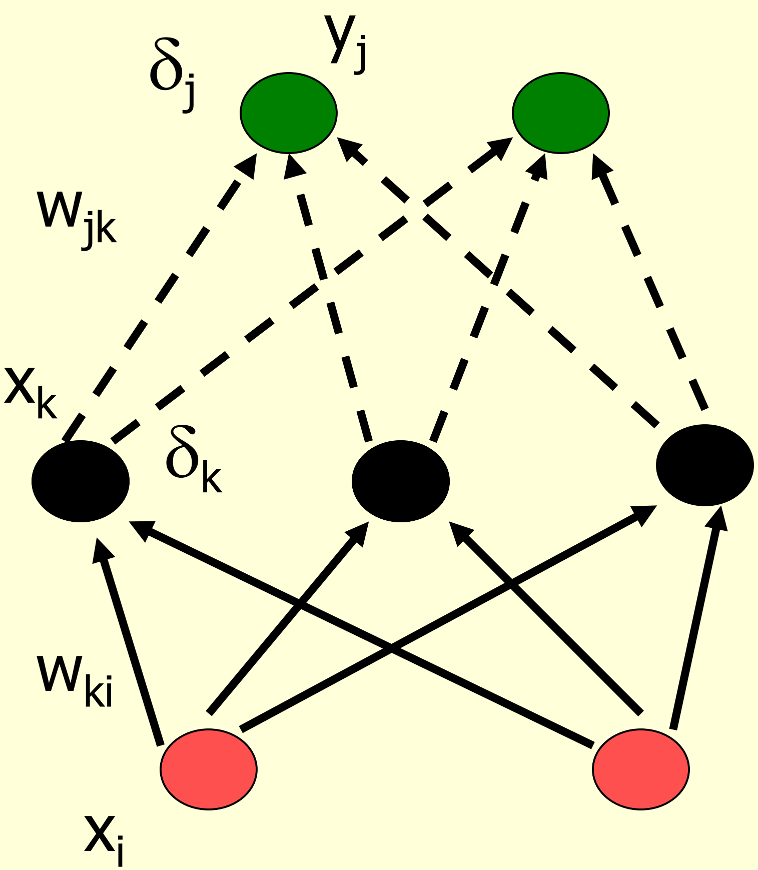
$$= - y_j^p (1 - y^p_j)(t^p_j - y^p_j) \;\; x_i^p$$

$$\Delta w_{ji} = \alpha \;\; y_j^p (1 - y_j^p)\,(t^p_j - y_j^p)\, x_i^p$$

$$= \alpha \;\; \delta_j^p \, x_i^p$$

with $\delta_j^p := y_j^p (1 - y_j^p)\,(t^p_j - y_j^p)$

Training-Rule for Weights to the Hidden Layer



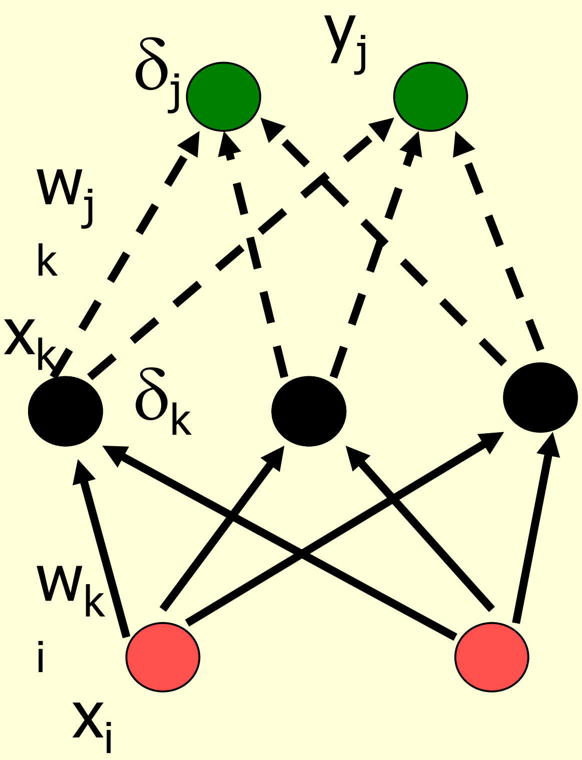
Credit assignment problem:
No target values t for hidden layer units.

Error for hidden units?

$$\delta_k = \Sigma_j \, w_{jk} \, \delta_j \, y_j \, (1-y_j)$$

$$\Delta w_{ki} = \alpha \; x_k^p(1-x_k^p) \, \delta_k^p \, x_i^p$$

Training-Rule for Weights to the Hidden Layer



$$E^p[w_{ki}] = \tfrac{1}{2} \Sigma_j (t_j^p - y_j^p)^2$$

$$\partial E^p / \partial w_{ki} = \partial / \partial w_{ki} \tfrac{1}{2} \Sigma_j (t_j^p - y_j^p)^2$$

$$= \partial / \partial w_{ki} \tfrac{1}{2} \Sigma_j (t_j^p - \sigma(\Sigma_k w_{jk} x_k^p))^2$$

$$= \partial / \partial w_{ki} \tfrac{1}{2} \Sigma_j (t_j^p - \sigma(\Sigma_k w_{jk} \sigma(\Sigma_i w_{ki} x_i^p)))^2$$

$$= -\Sigma_j (t_j^p - y_j^p) \sigma'_j(a) w_{jk} \sigma'_k(a) x_i^p$$

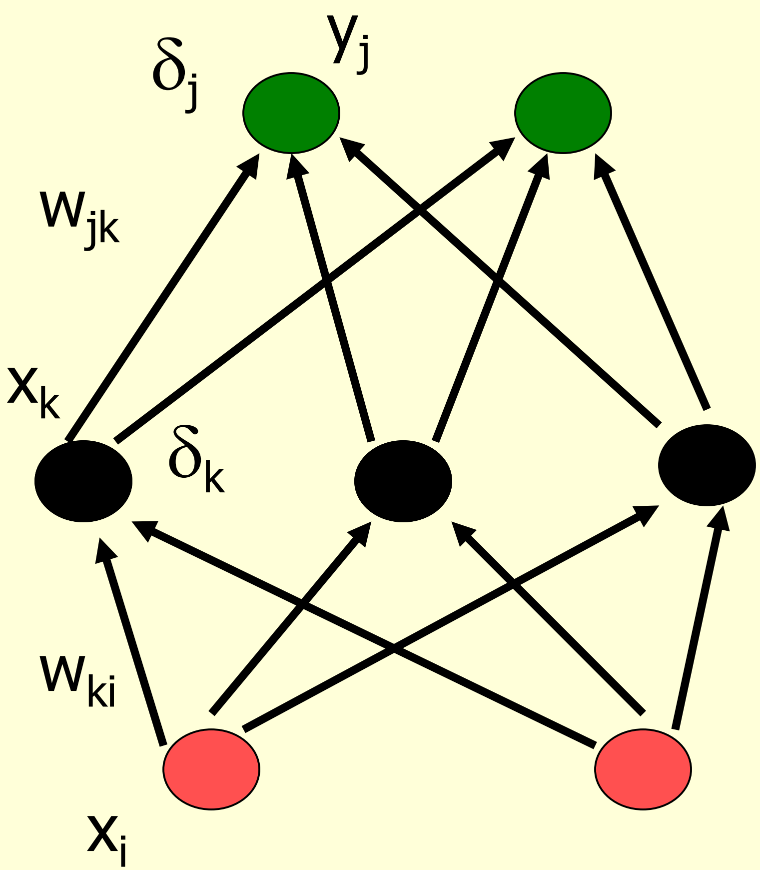
$$= -\Sigma_j \delta_j w_{jk} \sigma'_k(a) x_i^p$$

$$= -\Sigma_j \delta_j w_{jk} x_k (1 - x_k) x_i^p$$

$$\Delta w_{ki} = \alpha \delta_k x_i^p$$

with $\delta_k = \Sigma_j \delta_j w_{jk} x_k(1-x_k)$

Backpropagation



Backward step:
propagate errors from output to hidden layer

Forward step:
Propagate activation from input to output layer

Backpropagation Algorithm

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - For each training example $<(x_1,\ldots x_n),t>$ Do
    - Input the instance $(x_1,\ldots,x_n)$ to the network and compute the network outputs y_k
    - For each output unit k
      - $\delta_k = y_k(1-y_k)(t_k-y_k)$
    - For each hidden unit h
      - $\delta_h = y_h(1-y_h) \sum_k w_{h,k} \delta_k$
    - For each network weight $w_{i,j}$ Do
    - $w_{i,j} = w_{i,j} + \Delta w_{i,j}$ where

      $\Delta w_{i,j} = \eta \, \delta_j \, x_{i,j}$

MLP Exercise (Due June 29)

- Become familiar with the Neural Network Toolbox in Matlab
- Construct a single hidden layer, feed forward network with sigmoidal units. to output. The network should have n hidden units n=3 to 6.
- Construct two more networks of same nature with n-1 and n+1 hidden units respectively.
- Initial random weights are from ~ $N(\mu,\sigma^2)$
- The dimensionality of the input data is d

MLP Exercise (Cnt'd)

- Constructing the a train and test set of size M
- For simplicity, choose two distributions N(-1, σ^2) and N(1, σ^2). Choose M/2 samples of d dimensions from the first distribution and M/2 from the second. This way you get a set of M vectors in d dimensions. Give the first set a class label of 0 and the second set a class label of 1.
- Repeat this again for the construction of the test set.

Actual Training

- Train 5 networks with the same training data (each network has different initial conditions)
- Construct a classification error graph for both train and test data taken at different time steps (mean and std over 5 nets)
- Repeat for n=3-6 using both n+1 and n-1
- **Discuss the results, justify with graphs and provide clear understanding**
- you may try other setups to test your understanding