

Prof. Marios Savvides

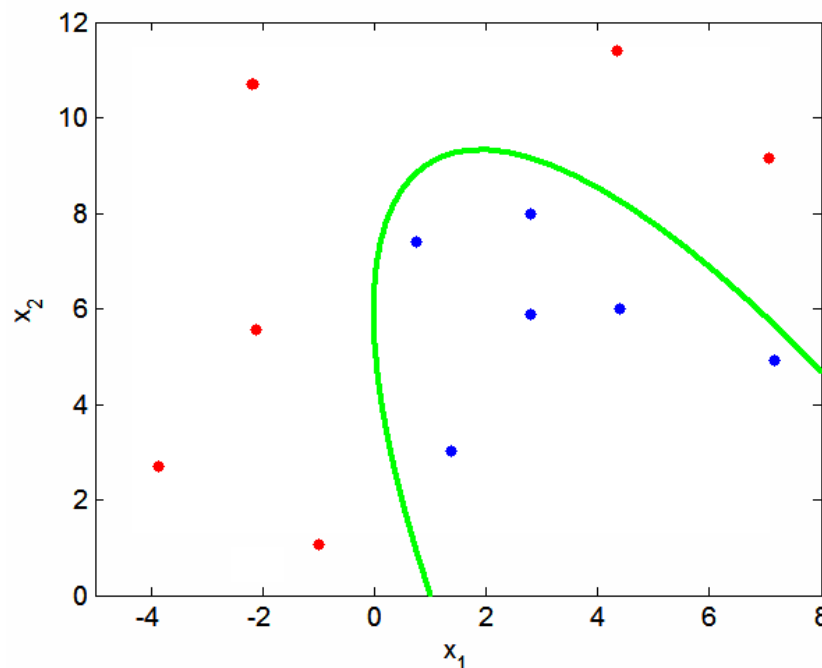
# Pattern Recognition Theory

## Lecture 12 : Perceptron Learning And Neural Networks

All graphics from Pattern Classification, Duda, Hart and Stork, Copyright © John Wiley and Sons, 2001

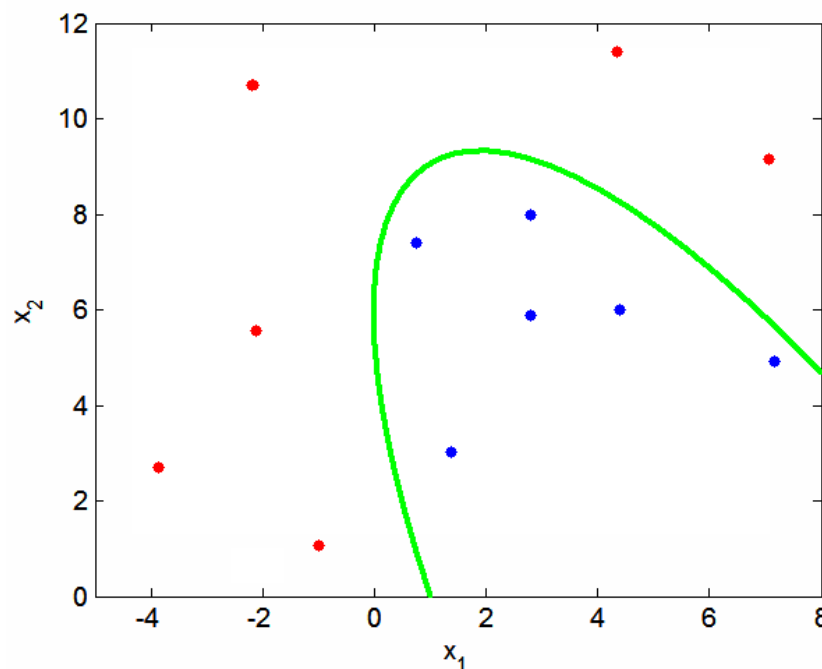
# Discriminant Functions

- A discriminant function is a function of the feature pattern  $x$  that **leads to a classification** rule.
- We can use **any monotonic transformation** of the discriminant function and it will lead to the same decision rule
- In contrast to previous approaches, now the **form of the DF is specified**, and is not imposed by the underlying distribution
  - ➔ Discriminative approach (as opposed to Generative approach)



# Discriminant Function Estimation

- Specify a **parametric form** of the decision boundary ( for ex, linear, quadratic, etc..)
- Find the **“best”** decision boundary of the specified form using a set of training examples.
- This best decision boundary is achieved by **minimizing some criterion function**
  - For example, minimize the *training error*



# Generative vs. Discriminative Methods

- **Generative Methods**

- Model class-conditional pdfs and prior probabilities.
- New data points can be **generated**.
- Examples : Bayesian Minimum Error , Gaussians, Mixture of Gaussians, HMMs...

- **Discriminative Methods**

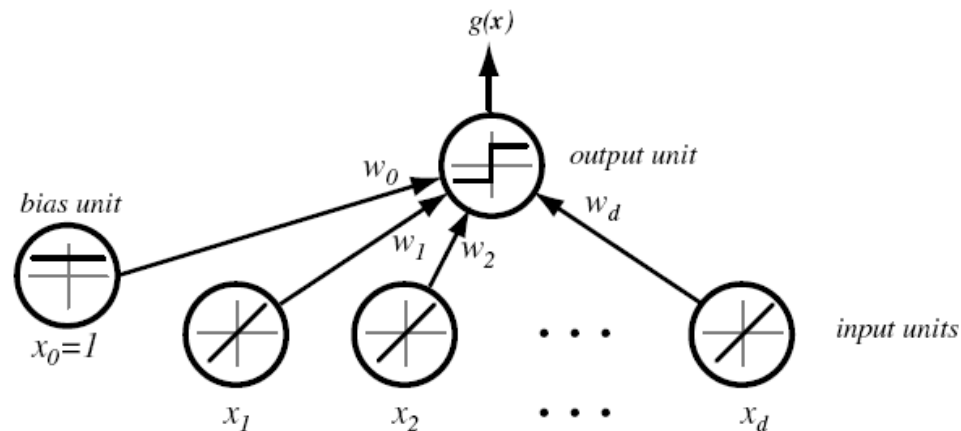
- **Directly** estimate posterior probabilities.
  - No need to model underlying probability distributions.
- Use existing data points
- Examples : *Support Vector Machines. Nearest Neighbor. Neural Networks.*

# Linear Discriminant Functions

- A linear discriminant function is a linear combination of its components:

$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 = \sum_{i=1}^d w_i x_i + w_0$$

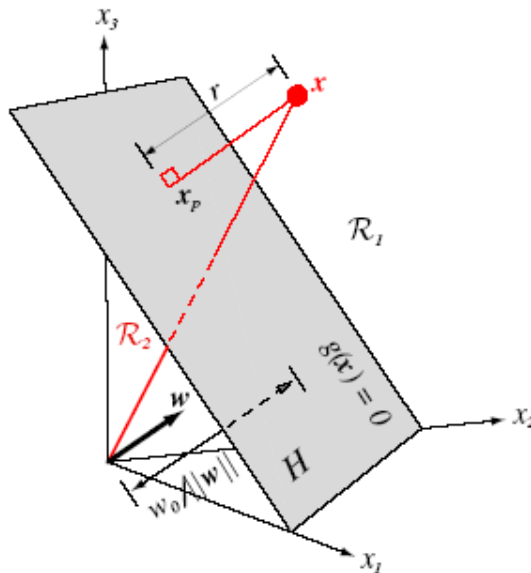
- $\mathbf{w}$  is the weight vector
  - $w_0$  is the bias (or threshold weight)
- Decide  $\omega_1$  if  $g(\mathbf{x}) > 0$  and  $\omega_2$  if  $g(\mathbf{x}) < 0$ 
  - If  $g(\mathbf{x})=0$  then  $\mathbf{x}$  is on the decision boundary and can be assigned to either class
- If  $g(\mathbf{x})$  is linear, the decision boundary is a **hyperplane**.



# Decision Boundary

$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 = \sum_{i=1}^d w_i x_i + w_0$$

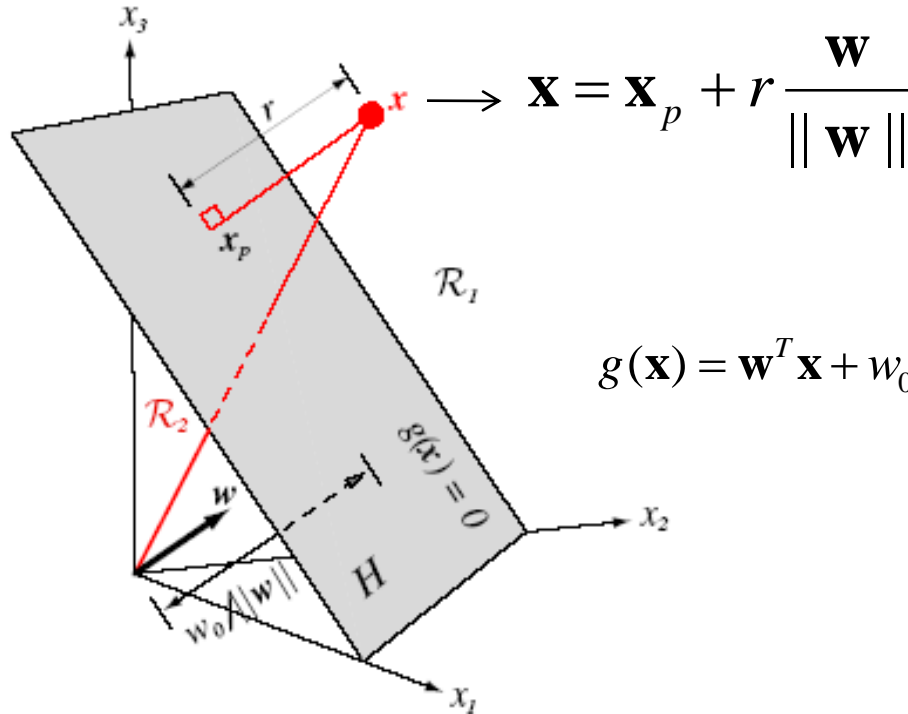
- The orientation of the hyperplane is determined by  $\mathbf{w}$  and its location by  $w_0$ 
  - $\mathbf{w}$  is the normal to the hyperplane
  - If  $w_0=0$ , the hyperplane passes through the origin



- If  $\mathbf{x}_1$  and  $\mathbf{x}_2$  are both on the decision surface, then
 
$$\mathbf{w}^T \mathbf{x}_1 + w_0 = \mathbf{w}^T \mathbf{x}_2 + w_0$$

$$\mathbf{w}^T (\mathbf{x}_1 - \mathbf{x}_2) = 0$$
- $\mathbf{w}$  is normal to any vector lying in the hyperplane.

# Decision Boundary



$$\begin{aligned}
 g(\mathbf{x}) &= \mathbf{w}^T \mathbf{x} + w_0 = \mathbf{w}^T \left( \mathbf{x}_p + r \frac{\mathbf{w}}{\|\mathbf{w}\|} \right) + w_0 = \mathbf{w}^T \mathbf{x}_p + r \frac{\mathbf{w}^T \mathbf{w}}{\|\mathbf{w}\|} + w_0 \\
 &= \underbrace{(\mathbf{w}^T \mathbf{x}_p + w_0)}_{g(\mathbf{x}_p)=0} + r \|\mathbf{w}\| = r \|\mathbf{w}\|
 \end{aligned}$$

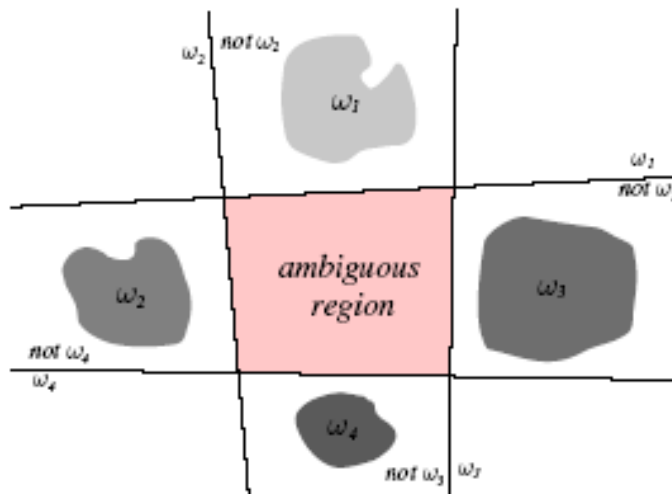
-This gives the distance of  $\mathbf{x}$  from  $H$ :  $r = \frac{g(\mathbf{x})}{\|\mathbf{w}\|}$

- $w_0$  determines the distance of the hyperplane from the origin:

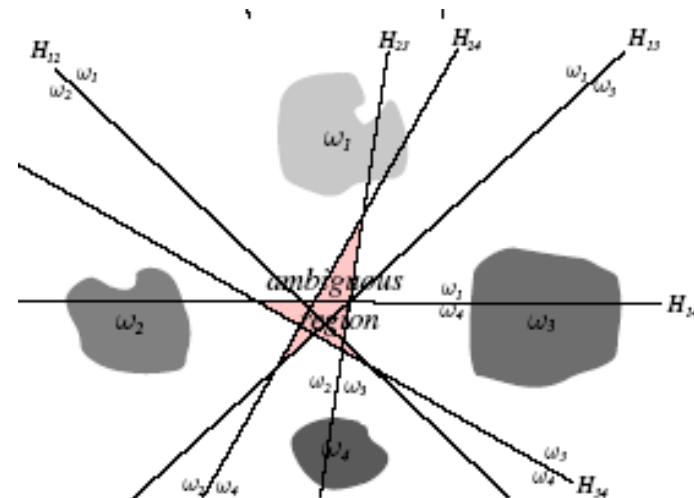
$$\frac{w_0}{\|\mathbf{w}\|}$$

# LDF: Multi-Category Case

- There are several ways to devise multicategory classifiers using LDFs:
  - One against the rest (  $\omega_i$ /not  $\omega_i$  dichotomies)
    - A total of  $c-1$  two-class problems
  - One against another (  $\omega_i$ /  $\omega_j$  dichotomies)
    - A total of  $c(c-1)/2$  pairs of classes
- Both of these methods produce ambiguous regions



**c-1 two class problems**



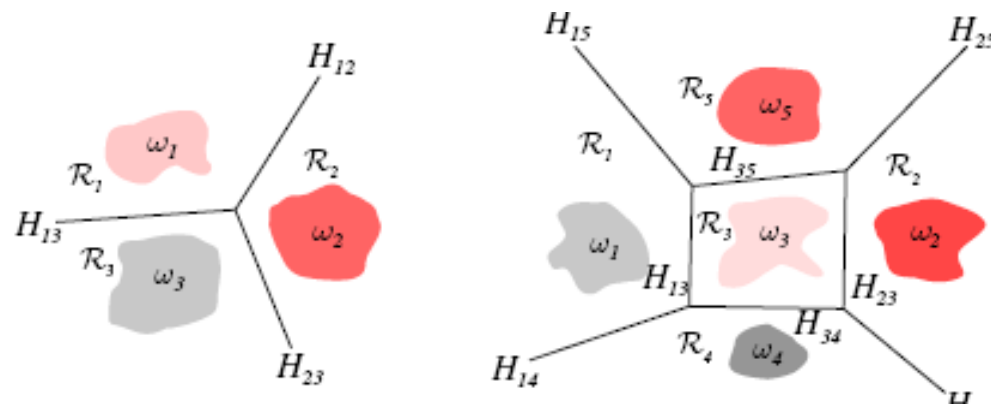
**c(c-1)/2 two class problems**



# LDF: Multi-Category Case

- To avoid the problem of ambiguous regions:
  - Define  $c$  linear discriminant functions
  - Assign  $\mathbf{x}$  to  $\omega_i$  if  $g_i(\mathbf{x}) > g_j(\mathbf{x})$  for  $i \neq j$
- The resulting classifier is called a **linear machine**:

**CONVEX &  
SIMPLY  
CONNECTED  
DECISION  
REGIONS**



- The boundary between two regions is a portion of the hyperplane given by:

$$g_i(\mathbf{x}) = g_j(\mathbf{x}) \quad \text{or}$$

$$(\mathbf{w}_i - \mathbf{w}_j)^t \mathbf{x} + (w_{i0} - w_{j0}) = 0$$

# Higher Order DFs

- High Order DFs can produce more complicated decision boundaries.
  - Quadratic discriminant: obtained by adding terms corresponding to product of pairs of components of  $\mathbf{x}$

$$g(\mathbf{x}) = w_0 + \sum_{i=1}^d w_0 x_i + \sum_{i=1}^d \sum_{j=1}^d x_i x_j w_{ij}$$

- Polynomial discriminant: obtained by adding terms such as  $x_i x_j x_k w_{ijk}$
- The Generalized Linear Discriminant Function:

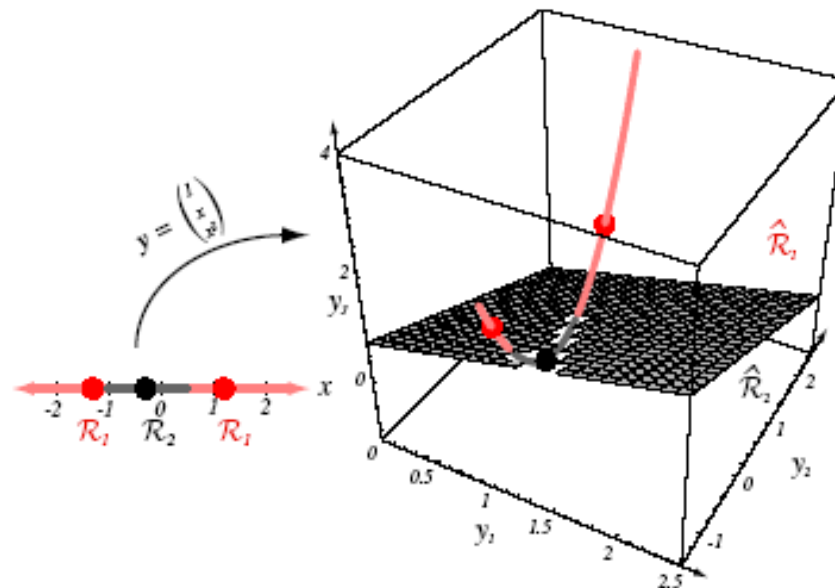
$$g(\mathbf{x}) = \sum_{i=1}^{\hat{d}} a_i y_i(x) = \mathbf{a}^T \mathbf{y}$$

- $\mathbf{A}$  is a  $\hat{d}$  dimensional weight vector
- The functions  $y_i(x)$  map points from the  $d$ -dimensional space to the  $\hat{d}$  dimensional space. (usually  $\hat{d} \gg d$ )
- The resulting DF is **not linear in  $\mathbf{x}$  but linear in  $\mathbf{y}$**
- The generalized discriminant separates points in the transformed space

# Generalized DF Example:

$$g(x) = -1 + x + 2x^2 \quad \mathbf{a} = [-1 \quad 1 \quad 2]^T, \mathbf{y} = [1 \quad x \quad x^2]^T$$

- Maps a line in  $\mathbf{x}$ -space to a parabola in  $\mathbf{y}$ -space
- The plane  $\mathbf{a}^T \mathbf{y} = 0$  divides the  $\mathbf{y}$ -space in two decision regions
- The corresponding decision regions in the original  $\mathbf{x}$ -space are not simply connected.



# Alternative Notation: Augmented Feature/Weight Vectors

- Exploit the convenience of writing  $g(\mathbf{x})$  in the homogeneous form

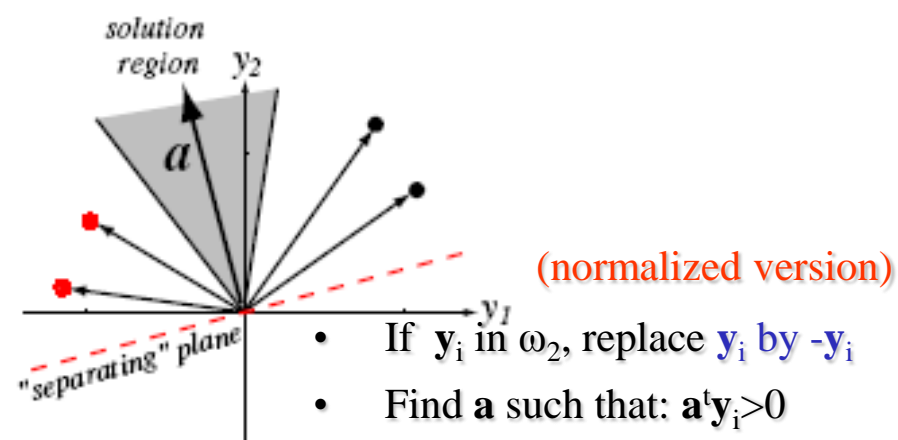
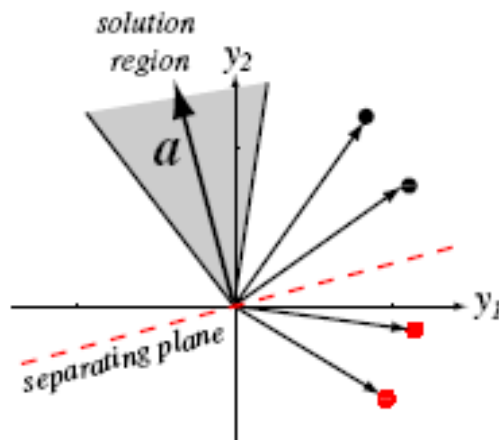
$$g(\mathbf{x}) = \mathbf{w}^t \mathbf{x} + w_0 = \sum_{i=1}^d w_i x_i + x_0 w_0 = \sum_{i=0}^d w_i x_i = \mathbf{a}^t \mathbf{y}$$

$$\mathbf{a} = \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_d \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 1 \\ x_1 \\ \dots \\ x_d \end{bmatrix} \quad x_0 = 1$$

- Decision hyperplane passes through origin in  $\mathbf{y}$ -space.

# Two-Class, Linearly Separable

- Given a linear DF  $g(\mathbf{x})=\mathbf{a}^T\mathbf{y}$ , the goal is to learn the weights using a set of  $n$  labeled samples.
- Classification rule:
  - If  $\mathbf{a}^T\mathbf{y} > 0$ , assign  $y_i$  to  $\omega_1$  else if  $\mathbf{a}^T\mathbf{y} < 0$ , assign  $y_i$  to  $\omega_2$
- Every training sample places a constraint on the weight vector  $\mathbf{a}$
- Given  $n$  samples, the solution must lie in the intersection of  $n$  half-spaces.
- Solution vector is usually not unique! Impose constraints to enforce uniqueness.



# Perceptron Learning 1

- Now consider the problem of learning a binary classification problem with a linear discriminant function.

- Remember that our objective is to find a vector  $\mathbf{a}$  such that

$$g(\mathbf{x}) = \mathbf{a}^T \mathbf{y} = \begin{cases} > 0 & \mathbf{x} \in \omega_1 \\ < 0 & \mathbf{x} \in \omega_2 \end{cases}$$

- To simplify the derivation, we will normalize the training set by replacing all samples from class  $\omega_2$  by their negative:

$$\mathbf{y} \leftarrow [-\mathbf{y}] \quad \forall \mathbf{y} \in \omega_2$$

- Now we can ignore the class labels and look for a weight vector such that

$$g(\mathbf{x}) = \mathbf{a}^T \mathbf{y} > 0$$

- To find the solution we must define the criterion function  $J(\mathbf{a})$

- One choice is known as the **Perceptron criterion function**  $J_P(\mathbf{a}) = \sum_{\mathbf{y} \in Y_M} -\mathbf{a}^T \mathbf{y}$
- $Y_M$  is the set of misclassified samples by  $\mathbf{a}$
- Note that  $J_P(\mathbf{a})$  is **non-negative** since  $\mathbf{a}^T \mathbf{y} < 0$  for the misclassified

# Perceptron Learning 2

- To find the minimum of this criterion function we use gradient descent.

$$J_P(\mathbf{a}) = \sum_{\mathbf{y} \in Y_M} -\mathbf{a}^T \mathbf{y}$$

$$\nabla_{\mathbf{a}} J_P(\mathbf{a}) = \sum_{\mathbf{y} \in Y_M} -\mathbf{y}$$

$Y_M$  is the set of misclassified samples by  $\mathbf{a}$

- The gradient descent update rule becomes:

$$\mathbf{a}(k+1) = \mathbf{a}(k) + \eta \sum_{\mathbf{y} \in Y_M(k)} \mathbf{y}$$

**Perceptron rule**

- This is known as the Perceptron batch update rule
- If the classes are linearly separable, the Perceptron rule is guaranteed to converge to a valid solution.
- However, if the 2 classes are not linearly separable, the Perceptron rule will not converge.

» Since no weight vector  $\mathbf{a}$  can correctly classify every sample, the corrections in the Perceptron rule will never diminish.

# Iterative Optimization

- To find the solution to the set of inequalities  $\mathbf{a}^t \mathbf{y}_i > 0$
- Define a criterion function  $J(\mathbf{a})$  that is minimized if  $\mathbf{a}$  is a solution vector
- Gradient descent is the general method for function minimization
  - Recall that the minimum of a function  $J(\mathbf{x})$  is defined by the zeros of the gradient.
$$\hat{\mathbf{x}} = \arg \min [J(\mathbf{x})] \rightarrow \nabla_{\mathbf{x}} J(\mathbf{x}) = 0$$
  - Only in very specific cases that this minimization function has a closed form solution.
  - In some other cases a closed form solution may exist but can be numerically unstable or too expensive/impractical.
- Gradient descent finds the minimum in an iterative way by moving in the direction of steepest descent.



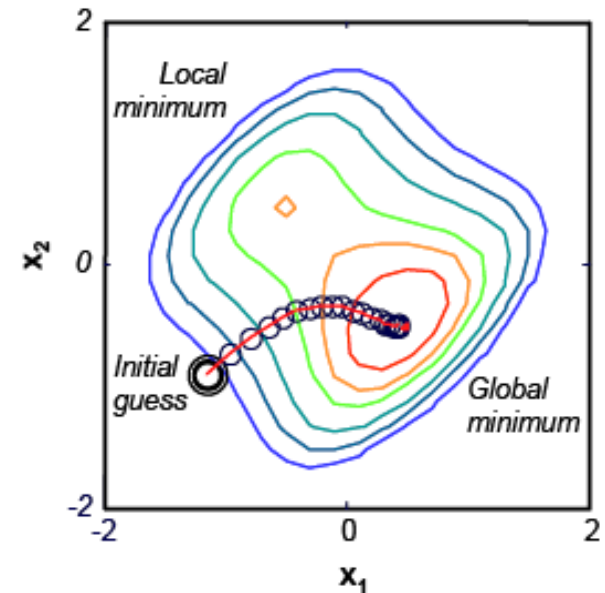
# Gradient Descent

- Gradient descent finds the minimum in an iterative way by moving in the direction of steepest descent.

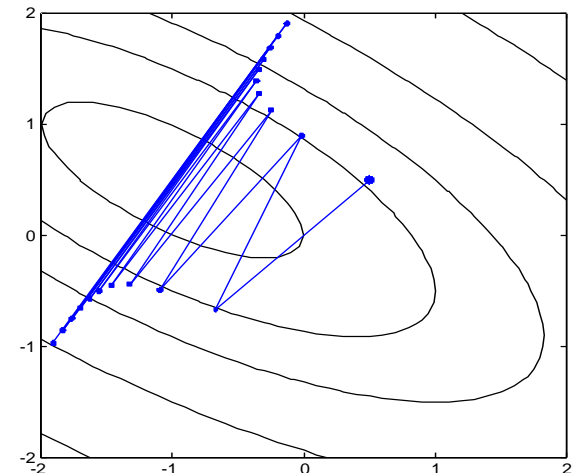
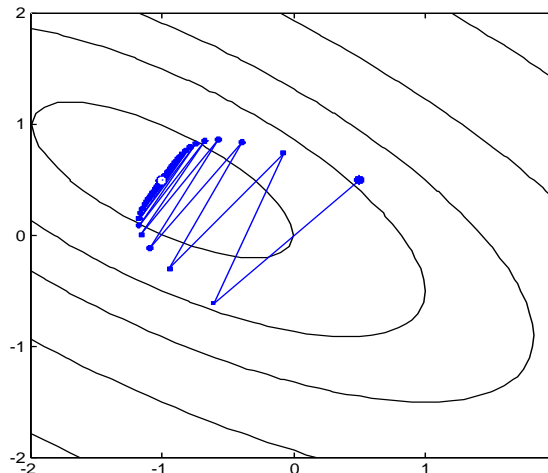
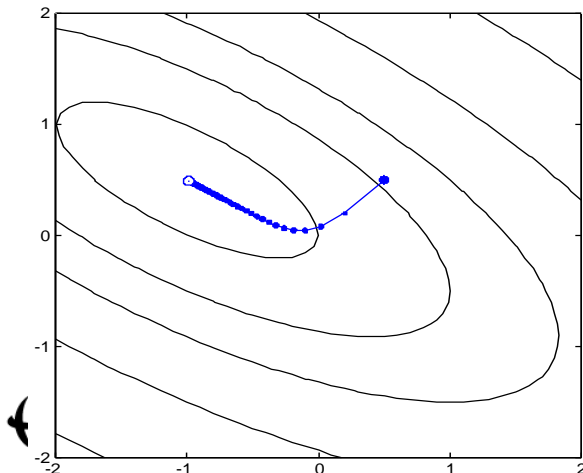
- Start with an arbitrary initial solution (guess)  $\mathbf{x}(0)$
- Compute the gradient  $\nabla_{\mathbf{x}} J(\mathbf{x}(k))$
- Move in the direction of steepest descent

$$\mathbf{x}(k+1) = \mathbf{x}(k) - \eta \nabla_{\mathbf{x}} J(\mathbf{x}(k))$$

- Repeat until convergence.



- Effect of the learning rate on convergence:



# Linearly Non-Separable

- **Linearly separable**
  - The Perceptron procedure
- **Linearly non-separable**
  - The Minimum Squared Error procedure
  - The Least Mean Squared Error procedure
  - The Ho-Kashyap procedure

# Minimum Squared Error Solution

- The traditional Minimum Squared Error (MSE) criterion provides an alternative to the Perceptron rule.
  - The Perceptron rule seeks a weight vector  $\mathbf{a}$  that satisfies  $\mathbf{a}^T \mathbf{y}_i > 0$ 
    - The perceptron rule only considers misclassified samples, since these are the only ones that violate the inequality
  - Instead, the MSE criterion looks for a solution to the equality  $\mathbf{a}^T \mathbf{y}_i = b_i$  where  $b$  are some pre-specified target values ( for example class labels).
    - The MSE solution uses ALL the samples in the training set.
- The system of equations solved by MSE is

$$\begin{bmatrix} y_0^1 & y_1^1 & \cdots & y_d^1 \\ y_0^2 & y_1^2 & & y_d^2 \\ \vdots & & \ddots & \vdots \\ y_0^n & y_1^n & \cdots & y_d^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} = \begin{bmatrix} b^1 \\ b^2 \\ \vdots \\ b^n \end{bmatrix} \Leftrightarrow \mathbf{Y}\mathbf{a} = \mathbf{b}$$

- where  $\mathbf{a}$  is the weight vector, each row in  $\mathbf{Y}$  is a training sample, and each row in  $\mathbf{b}$  is the corresponding class label.

# MSE Solution: Pseudo Inverse

- An exact solution to  $\mathbf{Y}\mathbf{a} = \mathbf{b}$  can be found sometimes.
  - If the number of equations ( $n$ ) is equal to the number of unknowns ( $d+1$ ), the exact solution is defined by

$$\mathbf{a} = \mathbf{Y}^{-1}\mathbf{b}$$

- In practice,  $\mathbf{Y}$  will be singular and the inverse does not exist.
  - $\mathbf{Y}$  will have more rows (samples) than columns (dimensions) which yields an over-determined system.
- The solution in this case is to find a weight vector that minimizes some function of the error between  $\mathbf{a}\mathbf{Y}$  and the desired output  $\mathbf{b}$ .

$$J_{MSE}(\mathbf{a}) = \sum_{i=1}^n (\mathbf{a}^T \mathbf{y}^i - \mathbf{b}^i)^2 = \|\mathbf{Y}\mathbf{a} - \mathbf{b}\|^2$$

- The gradient of  $J$  is given by  $\nabla_{\mathbf{a}} J_{MSE}(\mathbf{a}) = 2\mathbf{Y}^T (\mathbf{Y}\mathbf{a} - \mathbf{b}) = 0$
- If  $\mathbf{Y}^T\mathbf{Y}$  is nonsingular, the MSE solution becomes:

$$\mathbf{a} = (\mathbf{Y}^T\mathbf{Y})^{-1} \mathbf{Y}^T\mathbf{b} = \mathbf{Y}^\dagger\mathbf{b}$$

Pseudoinverse solution

Known as the normal equations

# MSE Numerical Example

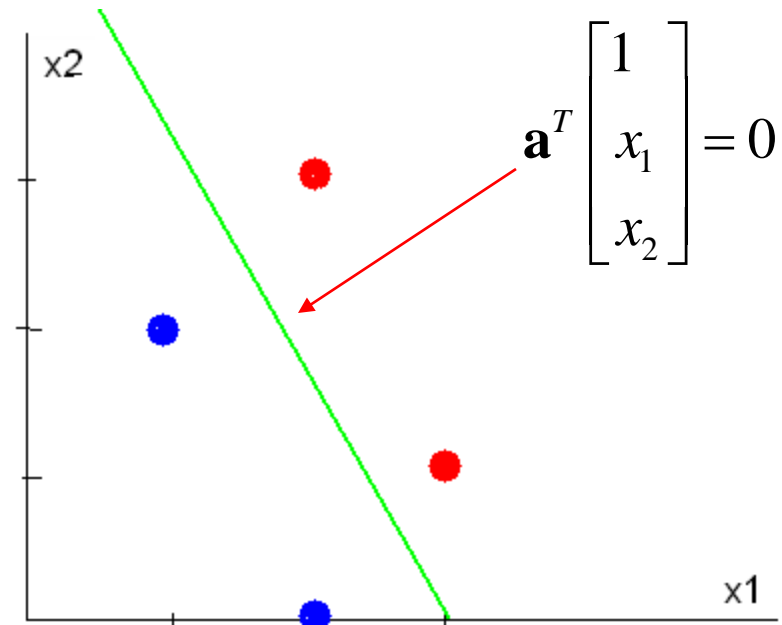
- Consider the following samples:

- Class 1: [1, 2] and [2, 0]
- Class 2: [3, 1], and [2, 3]

- Sample matrix ( $d = 1+2$ ,  $n = 4$ )

$$\mathbf{Y} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 0 \\ -1 & -3 & -1 \\ -1 & -2 & -3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\mathbf{y} \leftarrow [-\mathbf{y}] \quad \forall \mathbf{y} \in \omega_2$$



$$\mathbf{Y}^\dagger = (\mathbf{Y}^T \mathbf{Y})^{-1} \mathbf{Y}^T = \begin{bmatrix} 5/4 & 13/12 & 3/4 & 7/12 \\ -1/2 & -1/6 & -1/2 & -1/6 \\ 0 & -1/3 & 0 & -1/3 \end{bmatrix}$$

$$\mathbf{a} = \mathbf{Y}^\dagger \mathbf{b} = \begin{bmatrix} 11/3 \\ -4/3 \\ -2/3 \end{bmatrix}$$

# Perceptron vs MSE Example

- Consider the following dataset:

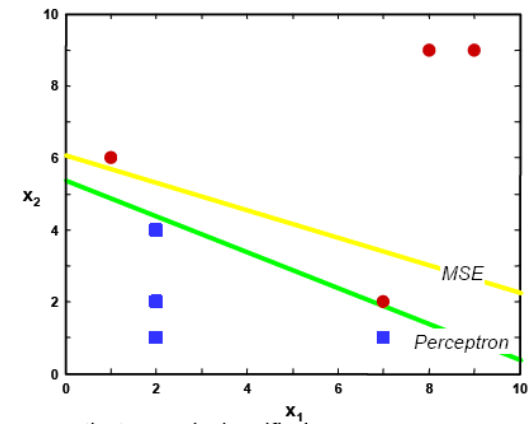
- $X1 = \{(1,6), (7,2), (8,9), (9,9)\}$
- $X2 = \{(2,1), (2,2), (2,4), (7,1)\}$

- The Perceptron learning approach:

- Assume  $\eta=0.1$  and  $a(0)=[0.1, 0.1, 0.1]$
- Using an online update rule:

- Normalize the data
- Iterate through all the examples and update  $a(k)$  on the ones that were misclassified
  - $Y(1) \rightarrow [1 \ 1 \ 6] * [0.1 \ 0.1 \ 0.1]^t > 0 \rightarrow$  no update
  - $Y(2) \rightarrow [1 \ 7 \ 2] * [0.1 \ 0.1 \ 0.1]^t > 0 \rightarrow$  no update
  - ...
  - $Y(5) \rightarrow [-1 \ -2 \ -1] * [0.1 \ 0.1 \ 0.1]^t < 0 \rightarrow$  update  $a(1) = [0.1 \ 0.1 \ 0.1] + \eta [-1 \ 2 \ -1] = [0 \ -0.1 \ 0]$
  - $Y(6) \rightarrow [-1 \ -2 \ -2] * [0 \ -0.1 \ 0]^t > 0 \rightarrow$  no update
  - ...
  - $Y(1) \rightarrow [1 \ 1 \ 6] * [0 \ -0.1 \ 0]^t < 0 \rightarrow$  update  $a(2) = [0 \ -0.1 \ 0] + \eta [1 \ 1 \ 6] = [0.1 \ 0 \ 0.6]$
- After 175 iterations, the resulting  $a = [-3.5 \ 0.3 \ 0.7]$

$$Y = \begin{bmatrix} 1 & 1 & 6 \\ 1 & 7 & 2 \\ 1 & 8 & 9 \\ 1 & 9 & 9 \\ -1 & -2 & -1 \\ -1 & -2 & -2 \\ -1 & -2 & -4 \\ -1 & -7 & -1 \end{bmatrix}$$



- MSE approach

- The MSE solution found is  $a = (Y^t Y)^{-1} Y^t b = [-1.18 \ 0.07 \ 0.19]$

# Least Mean Squares Solution

- The criterion function of the MSE  $J_{MSE}(\mathbf{a}) = \sum_{i=1}^n (\mathbf{a}^T \mathbf{y}^i - \mathbf{b}^i)^2 = \|\mathbf{Y}\mathbf{a} - \mathbf{b}\|^2$  can also be found using a gradient descent procedure

- This avoids inverting large matrices
- It is especially useful when  $\mathbf{Y}^T \mathbf{Y}$  is singular

- The update rule now becomes

$$\mathbf{a}(k+1) = \mathbf{a}(k) - \eta(k) \mathbf{Y}^T (\mathbf{b} - \mathbf{Y}\mathbf{a}(k))$$

**LMS rule – Batch update**

- It can be shown that if  $\eta(k) = \eta(1)/k$  where  $\eta(1)$  is any positive constant, this rule generates a sequences of weight vectors that converge to a solution to  $\mathbf{Y}^T(\mathbf{Y}\mathbf{a} - \mathbf{b}) = 0$

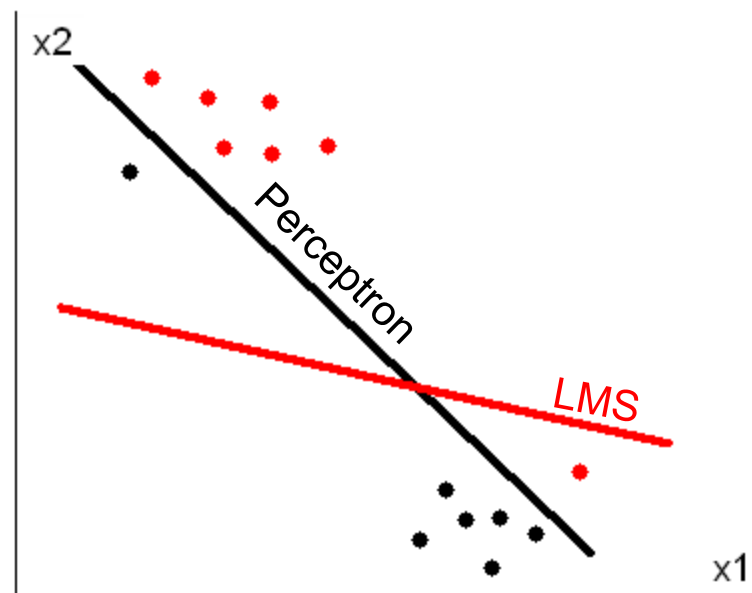
- Another advantage of this method is that the storage requirement can be further reduced by considering each sample sequentially

$$\mathbf{a}(k+1) = \mathbf{a}(k) - \eta(k) (b^i - y^i \mathbf{a}(k)) \mathbf{y}^i$$

**LMS rule – Sequential update**

# Perceptron Rule vs MSE Rule

- Perceptron rule:
  - The perceptron rule always finds a solution if the classes are linearly separable but does not converge in case of non-separability.
- MSE/LMS rule
  - The MSE/LMS solution guarantees convergence, but it may not find a separating hyperplane if the classes are linearly separable.
    - It just tries to minimize the sum of the square of the distances of the samples to the hyperplane, as opposed to finding the actual hyperplane.





# The Ho-Kashyap Procedure

- The main issue with the MSE criterion is the lack of guarantees that a separating hyperplane will be found in the linearly separable case
  - The MSE rule just minimizes  $J_{\text{mse}} = ||Y\mathbf{a} - \mathbf{b}||^2$ . Whether it will find a separating hyperplane or not depends on how the outputs  $\mathbf{b}$  are selected.
- If the two classes are linearly separable, there must be vectors  $\mathbf{a}^*$  and  $\mathbf{b}^*$  such that  $Y\mathbf{a} = \mathbf{b} > 0$ 
  - If  $\mathbf{b}^*$  were known, one could simply use the MSE solution to compute the separating hyperplane.
  - However, **since  $\mathbf{b}$  is also unknown**, one must then **solve for BOTH  $\mathbf{a}$  &  $\mathbf{b}$** .
  - We need to find the good labeling ( $\mathbf{b}$ ) .
- This gives an alternative training algorithm for LDFs known as the Ho-Kashyap procedure
  - Find the target values  $\mathbf{b}$  through gradient descent
  - Compute the weight vector  $\mathbf{a}$  from the MSE solution

# The Ho-Kashyap Procedure

- The gradients are defined by:  $J_{MSE}(\mathbf{a}, \mathbf{b}) = \|\mathbf{Y}\mathbf{a} - \mathbf{b}\|^2$

$$\nabla_{\mathbf{a}} J_{MSE}(\mathbf{a}, \mathbf{b}) = -2\mathbf{Y}^t (\mathbf{Y}\mathbf{a} - \mathbf{b}) \quad \nabla_{\mathbf{b}} J_{MSE}(\mathbf{a}, \mathbf{b}) = -2(\mathbf{Y}\mathbf{a} - \mathbf{b})$$

- For any value of  $\mathbf{b}$ , we can always use the pseudoinverse solution

$$\mathbf{a} = \mathbf{Y}^\dagger \mathbf{b}$$

- Constraints on  $\mathbf{b}$ :

- $\mathbf{b} > 0$

- We don't want  $\mathbf{b}(\mathbf{k}) \rightarrow 0$ . So start with a  $\mathbf{b} > 0$  and refuse to reduce any of its components.

$$\mathbf{b}(k+1) = \mathbf{b}(k) - \eta(k) \nabla_{\mathbf{b}} J_{MSE} = \mathbf{b}(k) - 2\eta(k)(\mathbf{Y}\mathbf{a} - \mathbf{b})$$

- The final iterative procedure is given by:

$$\mathbf{b}(k+1) = \mathbf{b}(k) - 2\eta(k) \mathbf{e}^+(k)$$

$$\mathbf{a}(k+1) = \mathbf{Y}^\dagger \mathbf{b}(k+1)$$

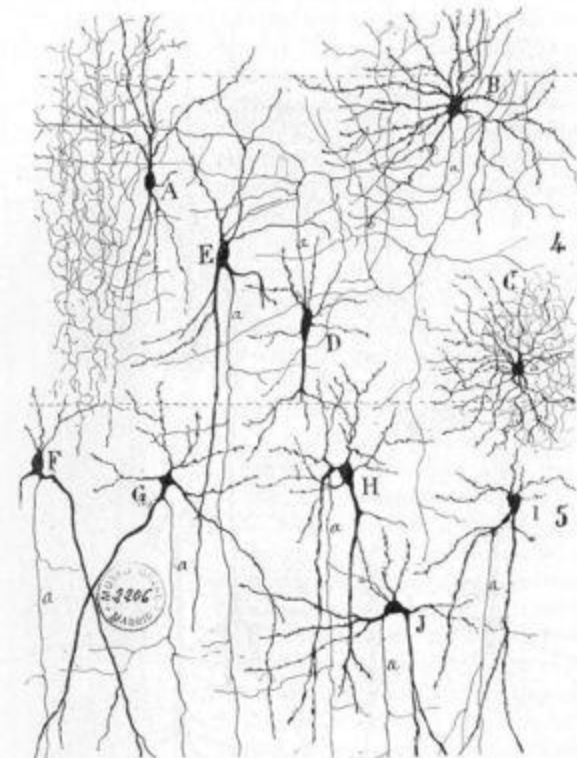
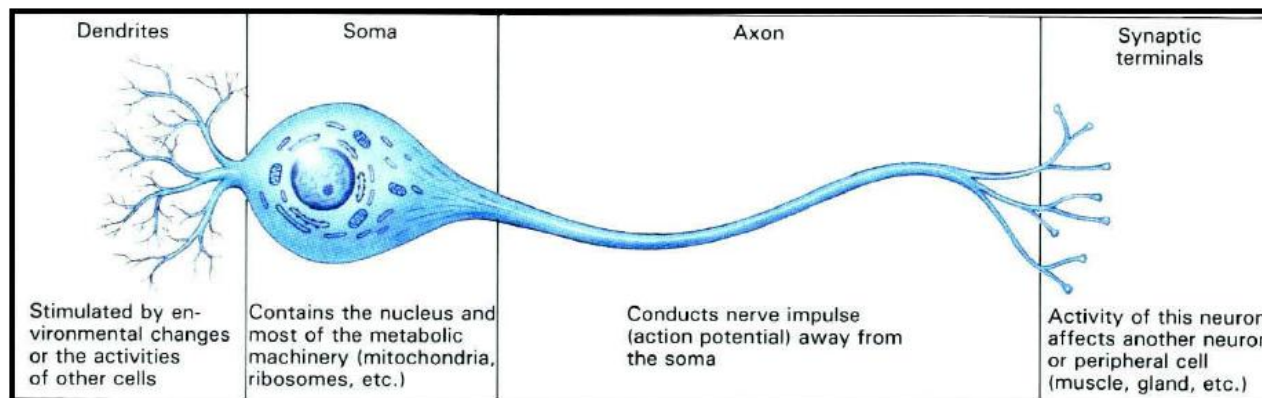
Ho-Kashyap  
procedure

Error vector  $\longrightarrow \mathbf{e}(k) = \mathbf{Y}\mathbf{a}(k) - \mathbf{b}(k)$

Positive part of the Error vector  $\longrightarrow \mathbf{e}^+(k) = \frac{1}{2}(\mathbf{e}(k) + |\mathbf{e}(k)|)$

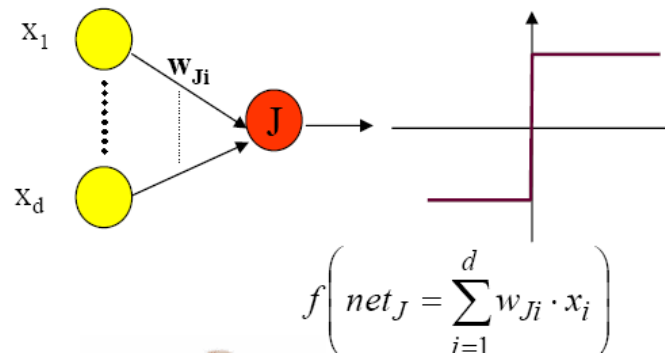
# Artificial Neural Networks

- An **artificial neural network** (ANN), often just called a "neural network" (NN), is a mathematical model or computational model, based on biological neural networks.
- It can be looked upon as consisting of an interconnected group of artificial neurons
- During the learning phase, this structure undergoes changes. The final structure is used for the final classification problem



# History Overview 1

- McCulloch & Pitts, 1943
- Hebb, 1949
- Rosenblatt, 1958
  - Introduced the Perceptron
    - A single neuron with adjustable synaptic weights and a threshold activation function
    - Also developed an error-correction rule to adapt the weights (Perceptron learning rule)
    - Proved that if 2 classes are linearly separable, the algorithm would converge (Perceptron convergence theorem).

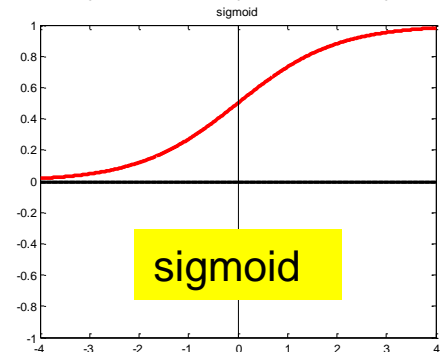
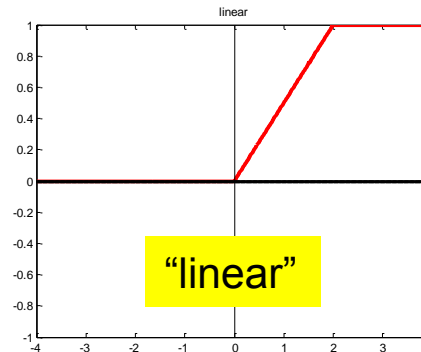
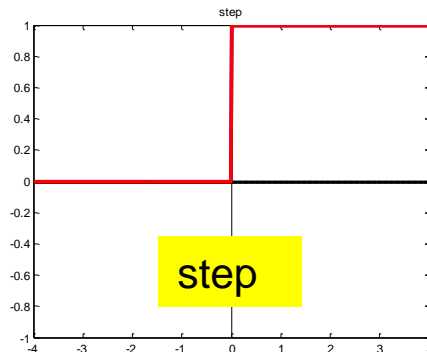
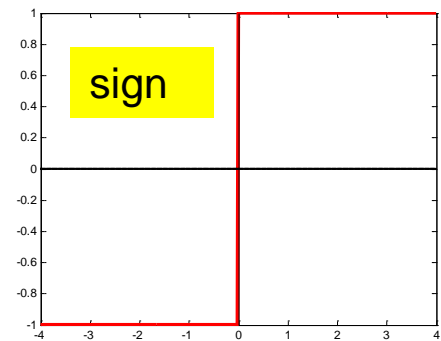
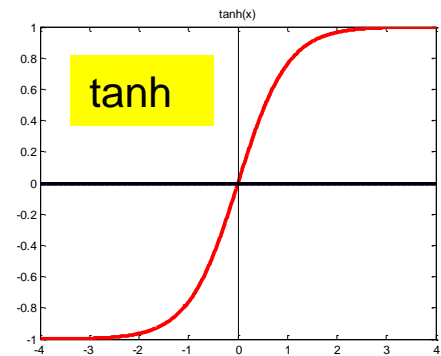
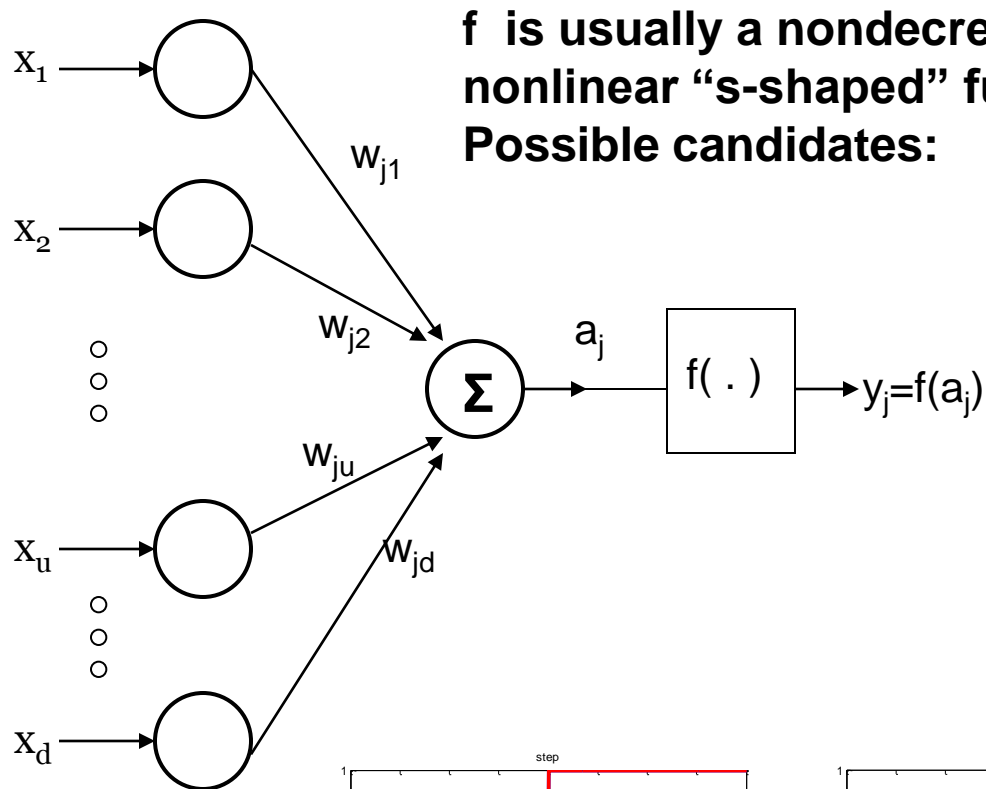


- Widrow & Hoff, 1960
  - Introduced the LMS algorithm
  - Hoff is also credited with the invention of the microprocessor

# History Overview 2

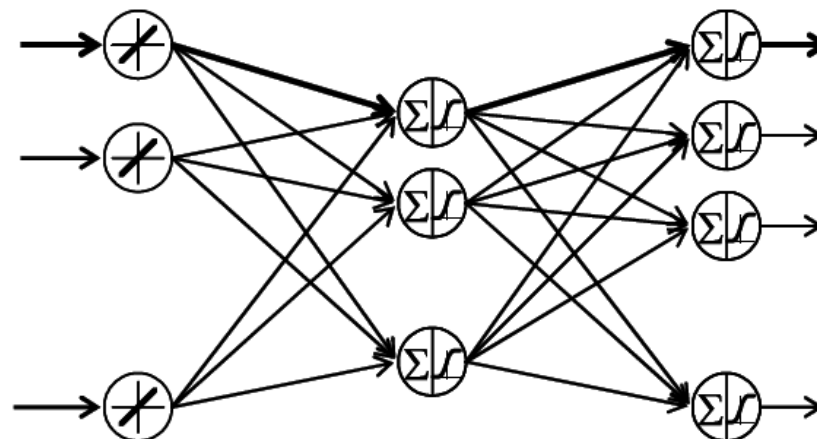
- Minsky & Papert, 1969
  - Pioneers of AI
  - Proved the limitations of the perceptron:
    - can't solve the XOR problem
- Rumelhart, Hinton & Williams, 1986
  - Developed a simple algorithm for training multilayer networks, learning nonlinearly separable decision boundaries through backward propagation of errors, a generalization of the LMS algorithm.
  - Overcame the limitations of the Perceptron.
  - Original inventor of the backpropagation algorithm was Paul Werbos (Ph.D. thesis Harvard 1974)

# Main Perceptron Unit



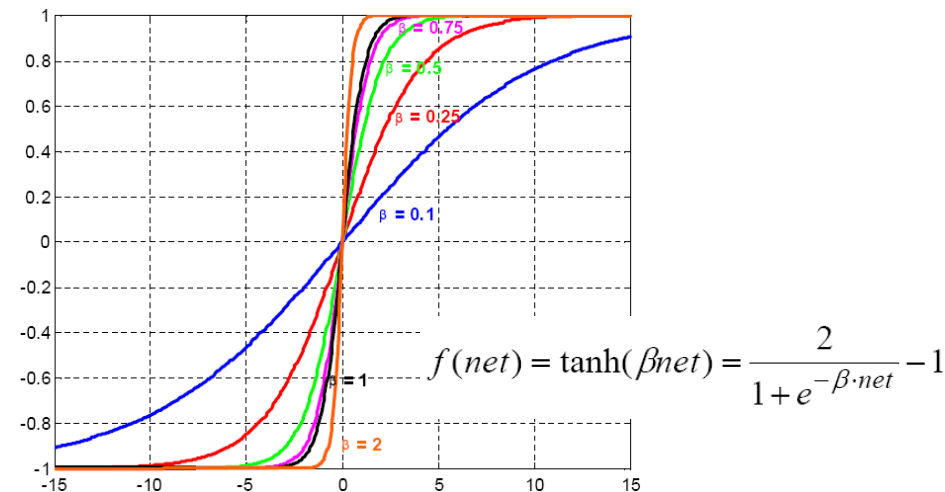
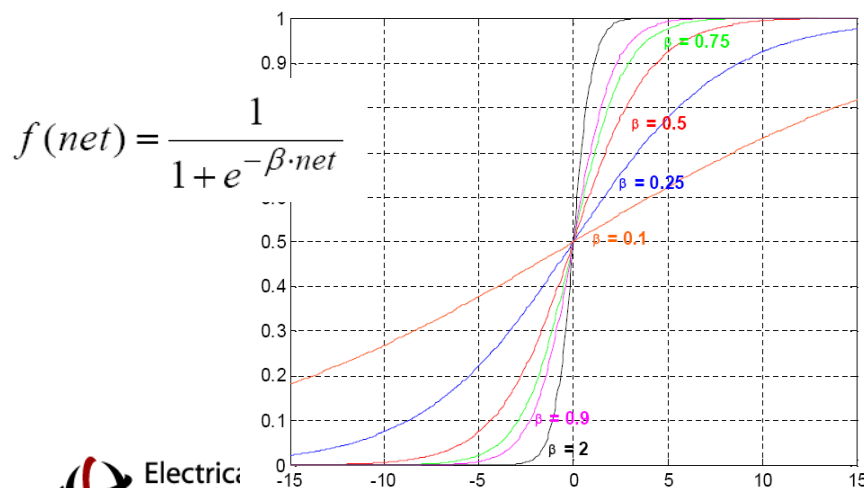
# Multilayer Perceptrons

- MLPs are feed-forward networks of simple processing units with at least ONE hidden layer.
- What truly separates an MLP from a regular simple Perceptron is the non-linear threshold function, also known as the activation function.
  - If a linear thresholding function is used, the MLP can be replaced with a series of simple Perceptrons which can only solve linearly separable problems.



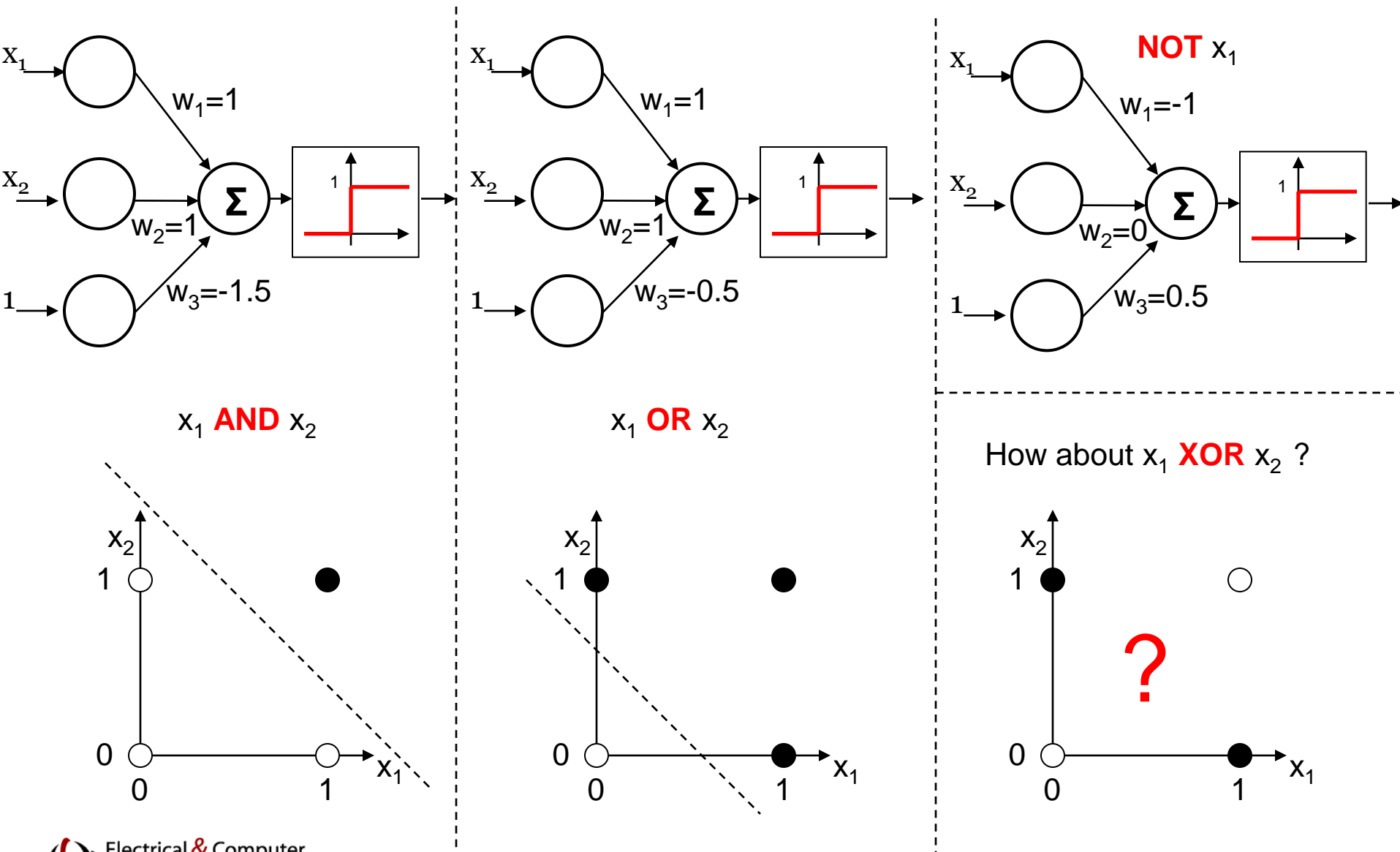
# Activation Functions

- Some desirable attributes of an activation function
  - Nonlinearity: gives the MLP the power of generating nonlinear decision boundaries
  - Saturation (for classification problems) so that the outputs can be limited between a min and a max limit ( e.g.  $\rightarrow -1$  &  $1$ ).
  - Continuity and smoothness – so we can take its derivative
  - Monotonicity – so that the activation function itself does not introduce additional local minima
- The logarithmic sigmoid and the tangential sigmoid satisfy all attributes

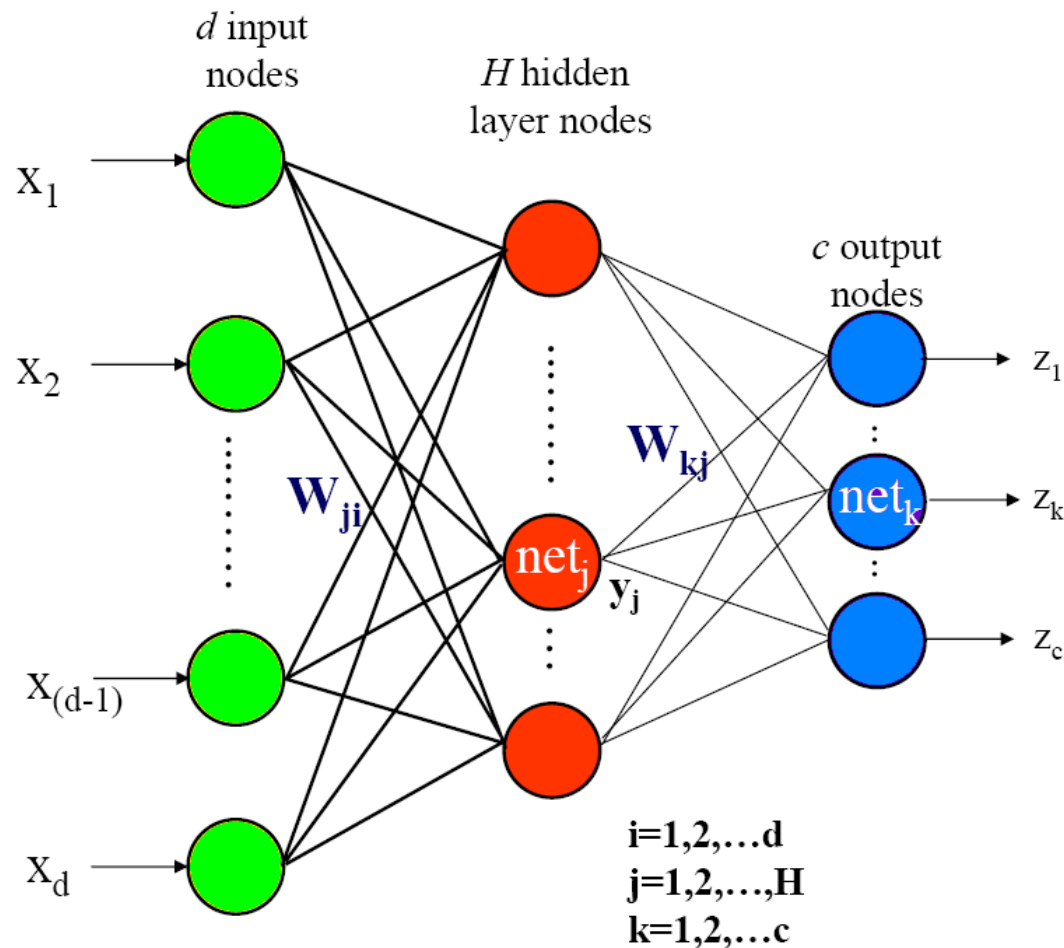




# What Can Perceptrons Represent?



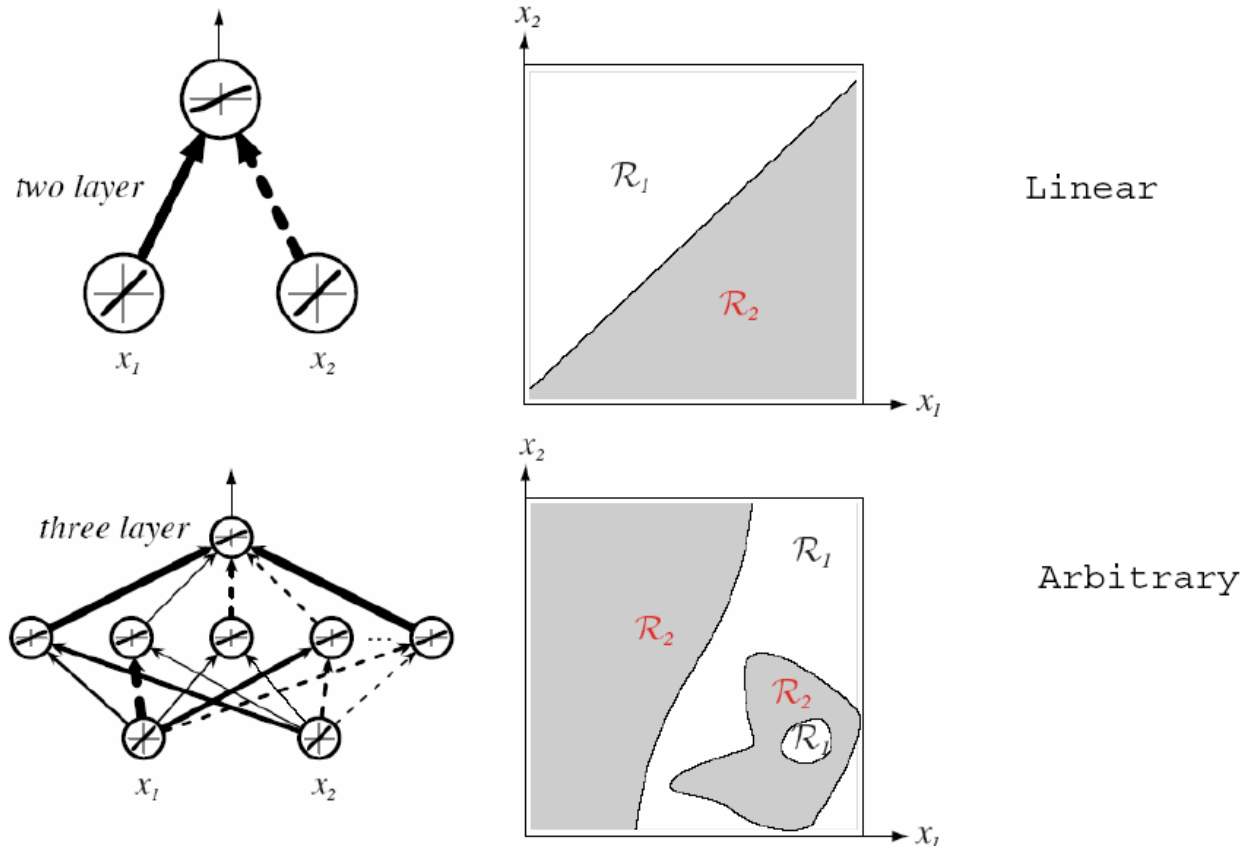
# Multilayer Perceptrons – Feedforward Operation



$$y_j = f(net_j) = f\left(\underbrace{\sum_{i=1}^d w_{ji}x_i}_{net_j}\right)$$

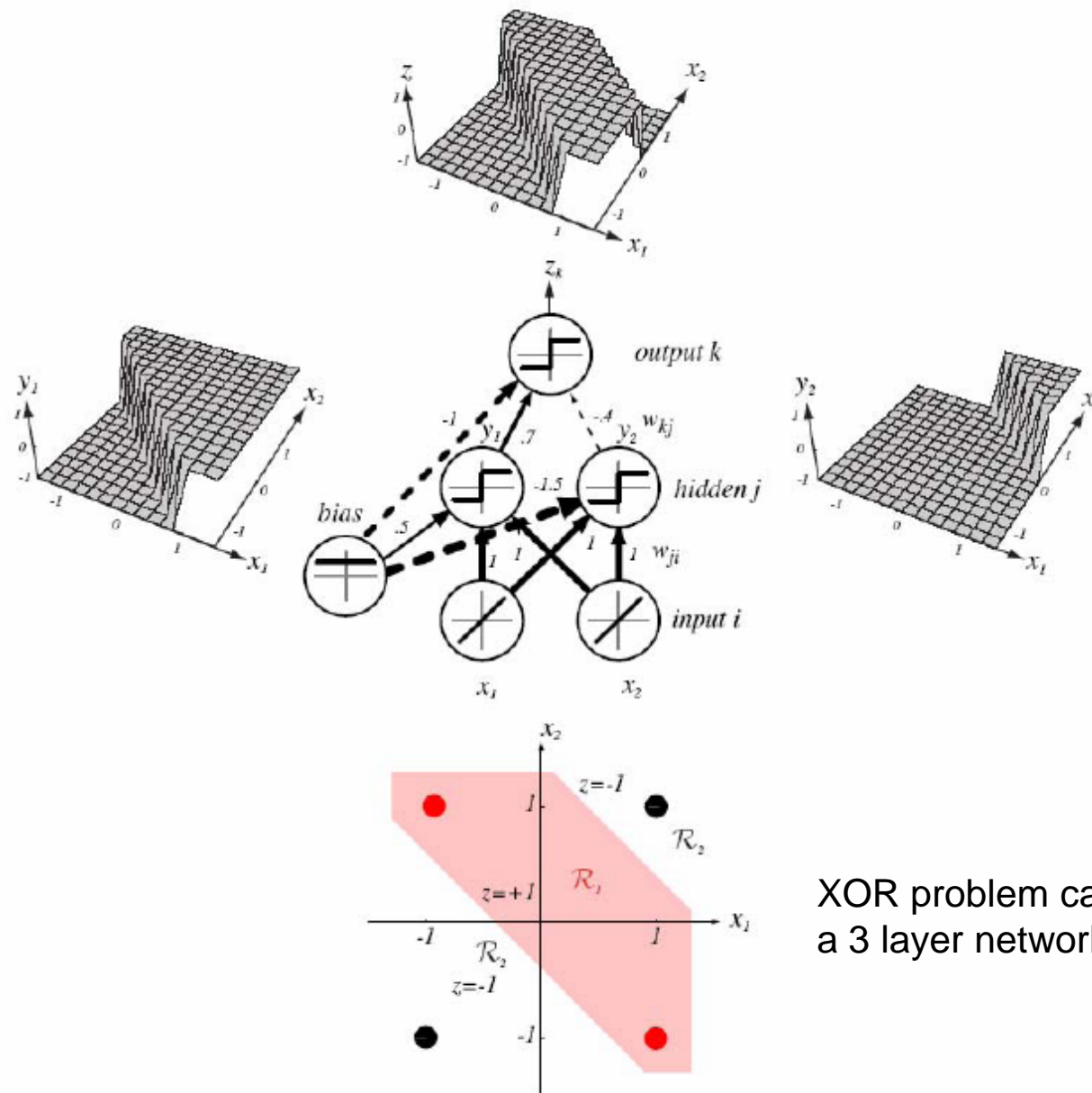
$$z_k = f(net_k) = f\left(\underbrace{\sum_{j=1}^H w_{kj}y_j}_{net_k}\right)$$

# Decisions Boundaries



A two layer network classifier can only implement a linear decision boundary. But given an adequate number of hidden units, higher-layer networks can implement arbitrary decision boundaries.

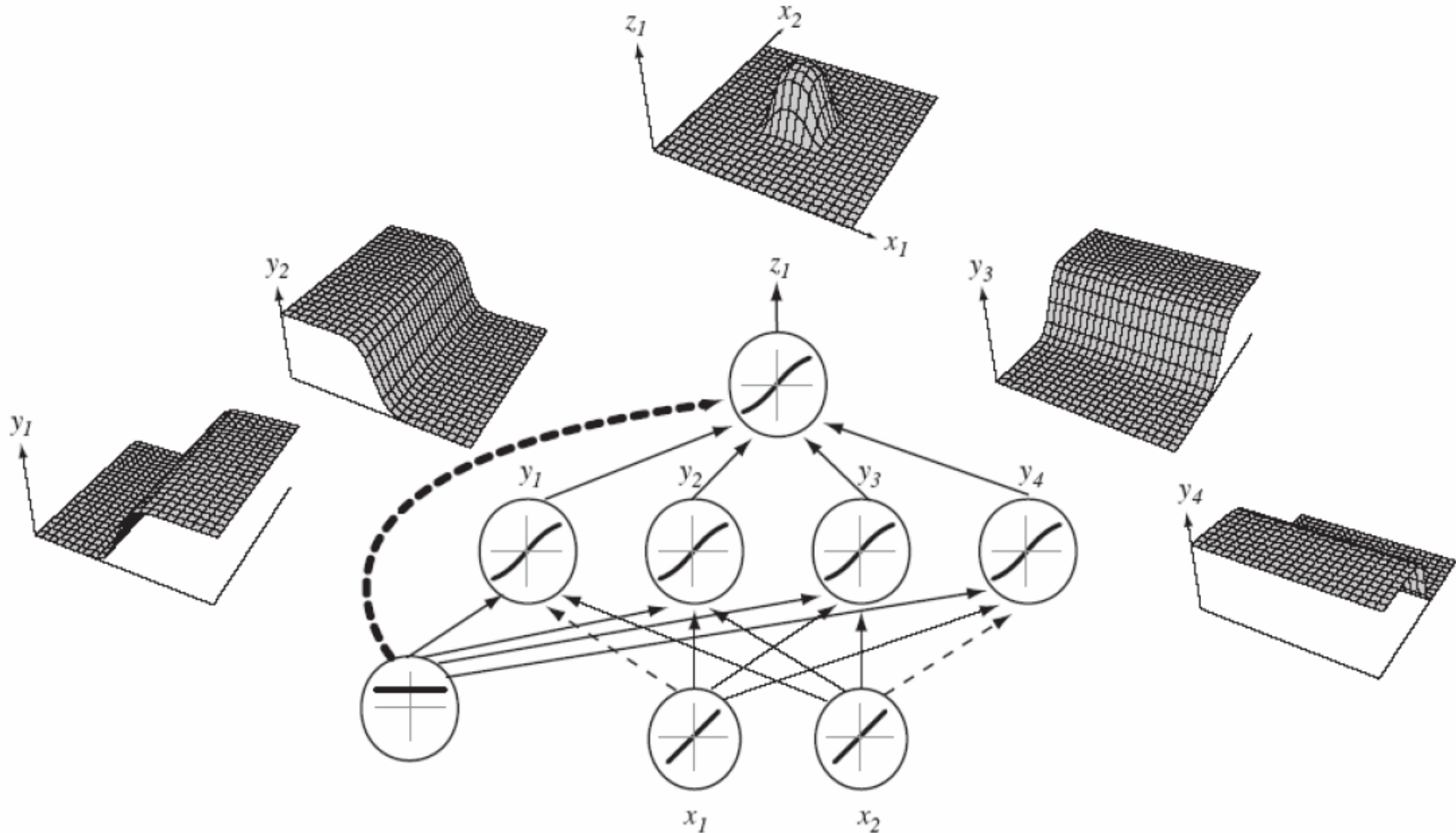
# XOR: Simple Three-Layer ANN



XOR problem can be solved by a 3 layer network.

# 2-4-1 ANN

Given a sufficiently large number of hidden units, any continuous function from input to output can be approximated arbitrarily well by such a network. Hence a MLP can solve any non-linearly separable classification problem.



# Notation

- $x_i$  is the  $i$ th input to the network
- $w_{ij}^{<1>}$  is the weight connection the  $i$ th input to the  $j$ th hidden neuron
- $net_j^{<1>}$  is the dot product at the  $j$ th hidden neuron
- $y_j$  is the output of the  $j$ th hidden neuron
- $w_{jk}^{<2>}$  is the weight connecting the  $j$ th hidden neuron to the  $k$ th output
- $net_k^{<2>}$  is the dot product at the  $k$ th output neuron
- $t_k$  is the target or desired output at the  $k$ th output neuron

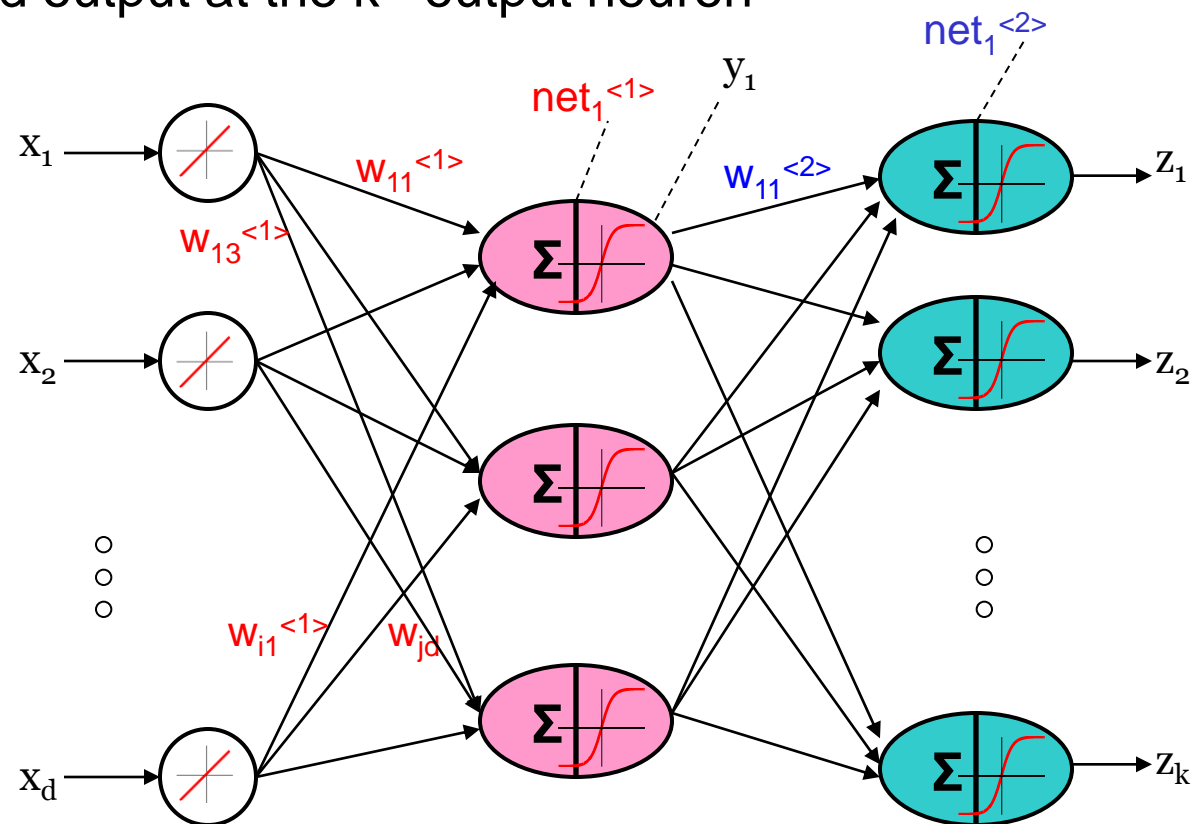
Example:

$$net_j^{<1>} = \sum_{i=1}^{N_{input}} x_i w_{ij}^{<1>}$$

$$y_j^{<1>} = f(net_j^{<1>}) = \frac{1}{1 + \exp(-net_j^{<1>})}$$

$$net_k^{<2>} = \sum_{j=1}^{N_{hidden}} y_j w_{jk}^{<2>}$$

$$y_k^{<2>} = f(net_k^{<2>}) = \frac{1}{1 + \exp(-net_k^{<2>})}$$



# Backpropagation Algorithm 1

- The learning problem is finding the weight vector  $\mathbf{w}$  that captures the input/output mapping implicit in a set of training examples
  - If we use the sum-squared error at the output as our criterion

$$J(\mathbf{w}) = \sum_{n=1}^{N_{\text{samples}}} \sum_{k=1}^{N_{\text{output}}} \frac{1}{2} \left( t_k^{(n)} - z_k^{(n)} \right)^2$$

- where  $t_k^{(n)}$  is the desired target of the  $k^{\text{th}}$  output neuro for the  $n^{\text{th}}$  sample
- where  $z_k^{(n)}$  is the output of the  $k^{\text{th}}$  output neuron for the  $n^{\text{th}}$  sample
- $\mathbf{w} = \{w_{ij}^{<1>}, w_{jk}^{<2>}, \dots\}$  is the set of all weights.

- Backprop learns the weights through gradient descent

$$w = w + \Delta w = w - \eta \frac{\partial J(\mathbf{w})}{\partial w}$$

- For simplicity, we first assume that the number of samples is 1

$$J(\mathbf{w}) = \sum_{k=1}^{N_{\text{output}}} \frac{1}{2} \left( t_k^{(n)} - z_k \right)^2$$

# Backpropagation Algorithm 2

- Computing  $dJ/dw$  for **hidden-to-output** (HO) weights
  - Using the **chain rule**, the derivative of  $J(W)$  with respect to a HO weight is

$$\frac{\partial J(\mathbf{w})}{\partial w_{jk}^{<2>}} = \frac{\partial J(\mathbf{w})}{\partial z_k} \frac{\partial z_k}{\partial net_k^{<2>}} \frac{\partial net_k^{<2>}}{\partial w_{jk}^{<2>}}$$

- Compute each one separately

$$- 1. \quad \frac{\partial J(\mathbf{w})}{\partial z_k} = \frac{\partial}{\partial z_k} \left[ \sum_{k=1}^{N_{output}} \frac{1}{2} \left( t_k^{(n)} - z_k^{(n)} \right)^2 \right] = z_k - t_k^{(n)}$$

$$- 2. \quad \frac{\partial z_k}{\partial net_k^{<2>}} = \frac{\partial}{\partial net_k^{<2>}} \left[ \frac{1}{1 + \exp(-net_k^{<2>})} \right]$$

$$= \frac{\exp(-net_k^{<2>})}{\left(1 + \exp(-net_k^{<2>})\right)^2} = \left[ \frac{\exp(-net_k^{<2>}) + 1 - 1}{1 + \exp(-net_k^{<2>})} \right] \left[ \frac{1}{1 + \exp(-net_k^{<2>})} \right] = (1 - z_k) z_k$$

$$- 3. \quad \frac{\partial net_k^{<2>}}{\partial w_{jk}^{<2>}} = \frac{\partial}{\partial w_{jk}^{<2>}} \left[ \sum_{n=1}^{N_{hidden}} w_{nk}^{<2>} y_n \right] = y_j$$

- Final Result for HO is

$$\frac{\partial J(\mathbf{w})}{\partial w_{jk}^{<2>}} = \left( z_k - t_k^{(n)} \right) (1 - z_k) z_k y_j$$



# Backpropagation Algorithm 3

- Computing  $dJ/dw$  for **input-to-hidden** (IH) weights
  - Using the **chain rule**, the derivative of  $J(W)$  with respect to a IH weight is
 
$$\frac{\partial J(\mathbf{w})}{\partial w_{ij}^{<1>}} = \frac{\partial J(\mathbf{w})}{\partial y_j} \frac{\partial y_j}{\partial net_j^{<1>}} \frac{\partial net_j^{<1>}}{\partial w_{ij}^{<1>}}$$
  - Following similar approach as before, we can easily compute the last 2 terms:
    - 1.  $\frac{\partial y_j^{<1>}}{\partial net_j^{<1>}} = (1 - y_j) y_j$
    - 2.  $\frac{\partial net_j^{<1>}}{\partial w_{ij}^{<1>}} = x_i$
  - The first term poses a problem and is not straightforward since we ignore what the outputs of the hidden neurons will be
    - This is known as the “credit assignment problem” [Minsky, 1961], and confused scientists for 2 decades
  - The trick to solve it was to realize that the hidden neurons do not make errors but only contribute to the errors of the output nodes.
    - The derivative of the error with respect to a hidden node’s output is the sum of that hidden node’s contribution to the errors of all the output neurons

$$\frac{\partial J(\mathbf{w})}{\partial y_j} = \sum_{n=1}^{N_{output}} \frac{\partial J(\mathbf{w})}{\partial z_n} \frac{\partial z_n}{\partial net_n^{<2>}} \frac{\partial net_n^{<2>}}{\partial y_j}$$

# Backpropagation Algorithm 4

- The derivative of the error with respect to a hidden node's output is the sum of that hidden node's contribution to the errors of all the output neurons

$$\frac{\partial J(\mathbf{w})}{\partial y_j} = \sum_{n=1}^{N_{\text{output}}} \frac{\partial J(\mathbf{w})}{\partial z_n} \frac{\partial z_n}{\partial \text{net}_n^{<2>}} \frac{\partial \text{net}_n^{<2>}}{\partial y_j}$$

- Using similar techniques to the previous step, we can get the first 2 terms:

$$\frac{\partial J(\mathbf{w})}{\partial z_n} = z_n - t_n^{(n)} \qquad \frac{\partial z_n}{\partial \text{net}_n^{<2>}} = (1 - z_n) z_n$$

- The last term is given by:

$$\frac{\partial \text{net}_n^{<2>}}{\partial y_j} = w_{jn}^{<2>}$$

- Merging all terms together, we obtain:

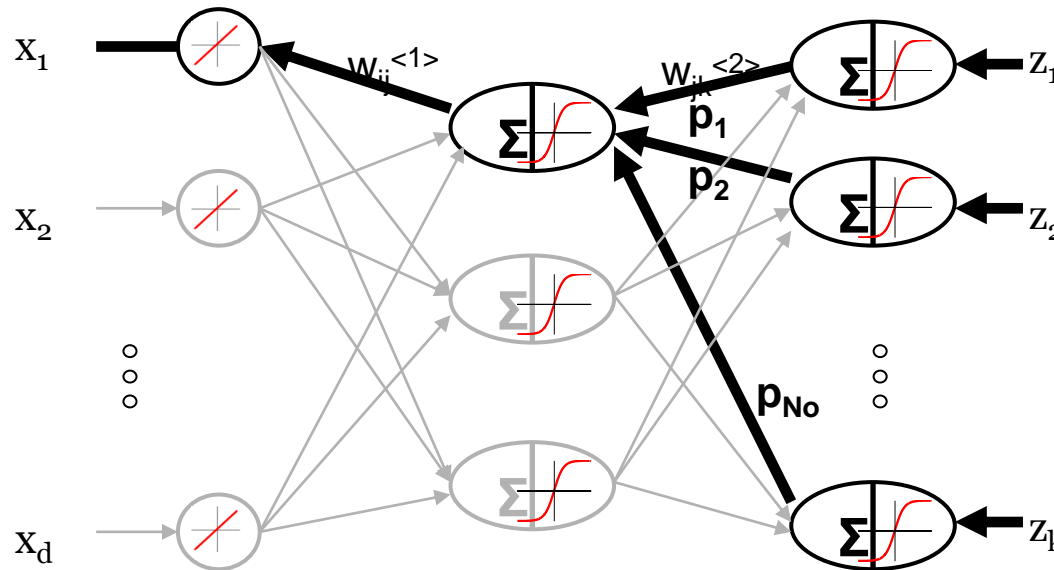
$$\frac{\partial J(\mathbf{w})}{\partial y_j} = \sum_{n=1}^{N_{\text{output}}} \frac{\partial J(\mathbf{w})}{\partial z_n} \frac{\partial z_n}{\partial \text{net}_n^{<2>}} \frac{\partial \text{net}_n^{<2>}}{\partial y_j} = \sum_{n=1}^{N_{\text{output}}} (z_n - t_n^{(n)}) (1 - z_n) z_n w_{jn}^{<2>}$$

# Backpropagation Algorithm 5

- The derivative of the error with respect to a hidden node is given by:

$$\frac{\partial J(\mathbf{w})}{\partial y_j} = \sum_{n=1}^{N_{\text{output}}} \underbrace{(z_n - t_n^{(n)})(1 - z_n) z_n}_{p_n} w_{jn}^{<2>}$$

- What we are actually doing is propagating the error term  $p_n$  backwards through the hidden-to-output weights



- The combined expression for  $dJ(\mathbf{w})/d\mathbf{w}$  for input-to-hidden weights is

$$\frac{\partial J(\mathbf{w})}{\partial w_{ij}^{<1>}} = \left[ \sum_{n=1}^{N_{\text{output}}} (z_n - t_n^{(n)})(1 - z_n) z_n w_{jn}^{<2>} \right] (1 - y_j) y_j x_j$$


# Backprop: Recap 1

- The weights are determined through a gradient descent error minimization of a criterion function  $J(\mathbf{w})$

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t) \Rightarrow \Delta\mathbf{w} = -\eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$$

- We need to express  $J(\mathbf{w})$  in terms of  $w$  for both output and hidden layer nodes. Output nodes are easy since we know the functional representation of  $J$  with respect to  $w$  through the chain rule:

$$\frac{\partial J(\mathbf{w})}{\partial w_{jk}} = \frac{\partial J(\mathbf{w})}{\partial net_k} \frac{\partial net_k}{\partial w_{jk}} = \frac{\partial J(\mathbf{w})}{\partial z_k} \frac{\partial z_k}{\partial net_k} \frac{\partial net_k}{\partial w_{jk}} = - \underbrace{(t_k - z_k) f'(net_k)}_{\delta_k} y_j$$


 Output node sensitivity

For output layer weights

$$\Delta w_{jk} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j$$

# Backprop: Recap 2

- For the hidden layer, things are more complicated because we do not know the desired values of the hidden layer node outputs. However, by the appropriate use of the chain rule we obtain:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_{ij}} = \frac{\partial J(\mathbf{w})}{\partial y_j} \frac{\partial y_j}{\partial w_{kj}} = \frac{\partial J(\mathbf{w})}{\partial y_j} \frac{\partial y_j}{\partial \mathbf{net}_j} \frac{\partial \mathbf{net}_j}{\partial w_{ij}} = \frac{\partial J(\mathbf{w})}{\partial y_j} f'(\mathbf{net}_j) x_i$$

$$\frac{\partial J(\mathbf{w})}{\partial y_j} = \frac{\partial}{\partial y_j} \left[ \frac{1}{2} \sum_{n=1}^{N_{\text{output}}} (t_k - z_k)^2 \right] = - \sum_{n=1}^{N_{\text{output}}} (t_k - z_k) = - \sum_{n=1}^{N_{\text{output}}} (t_k - z_k) \frac{\partial z_k}{\partial y_j}$$

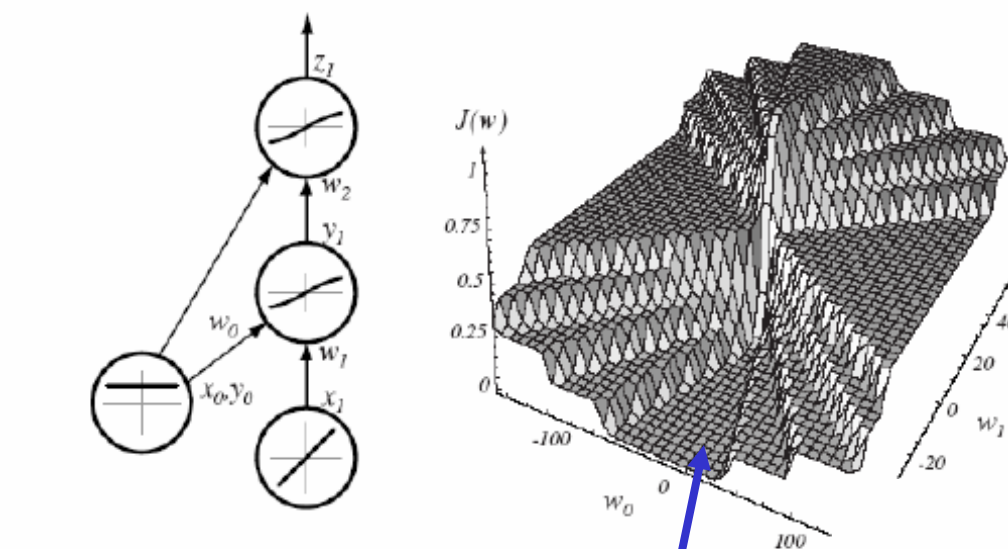
For hidden layer weights

$$= - \sum_{n=1}^{N_{\text{output}}} \underbrace{(t_k - z_k) f'(\mathbf{net}_k)}_{\delta_k} w_{jk}$$

$$\Delta \mathbf{w}_{ij} = -\eta \frac{\partial J}{\partial \mathbf{w}_{ij}} = \eta \sum_{k=1}^{N_{\text{output}}} \delta_k w_{jk} f'(\mathbf{net}_j) x_i = \eta \delta_j x_i$$

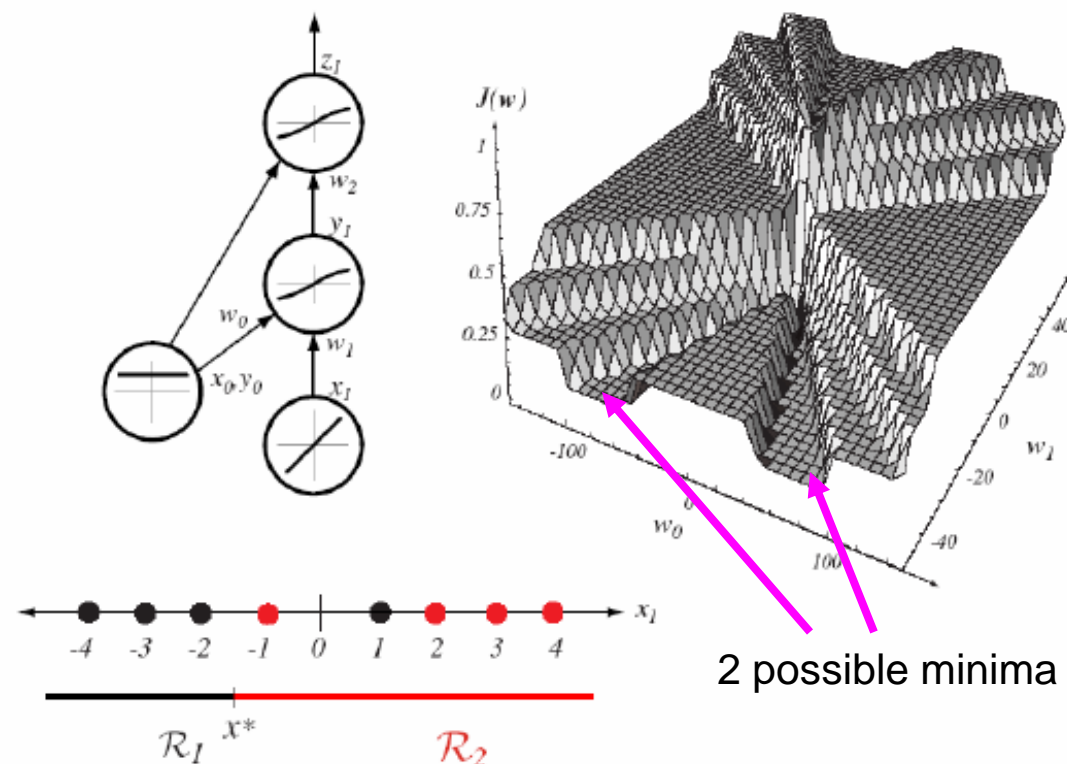
The parameter  $\zeta$  represents the sensitivity of the criterion function (error) with respect to a hidden/output layer node. Note that the sensitivity of a hidden layer node is a weighted sum of the output sensitivities, scaled by  $f(\mathbf{net}_j)$ , where the weights are the hidden-to-output layer weights.

# Error Surfaces



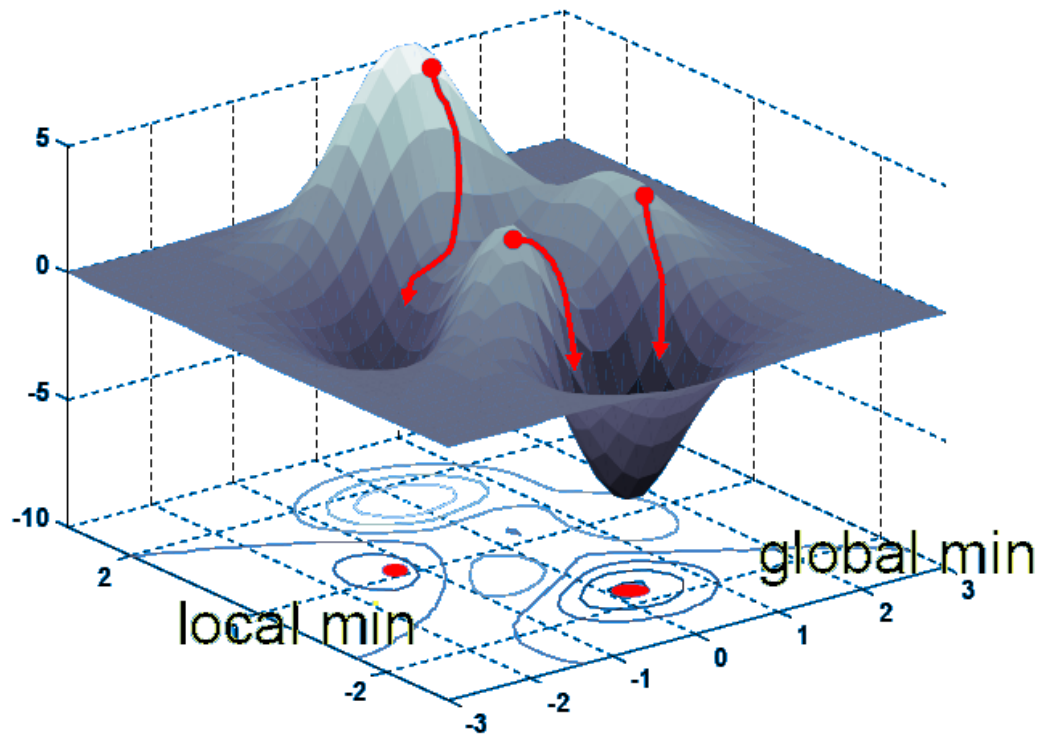
Non-Linearly Separable case:  
The error surface is slightly higher and two forms of minimum error solution exist.

Linearly Separable case: A low error solution exists which leads to a decision boundary separating the sample points



# General Multimodal Cost Surface

- One of the main problem of backprop (and gradient descent in general) is the local minima which creates a suboptimal solution

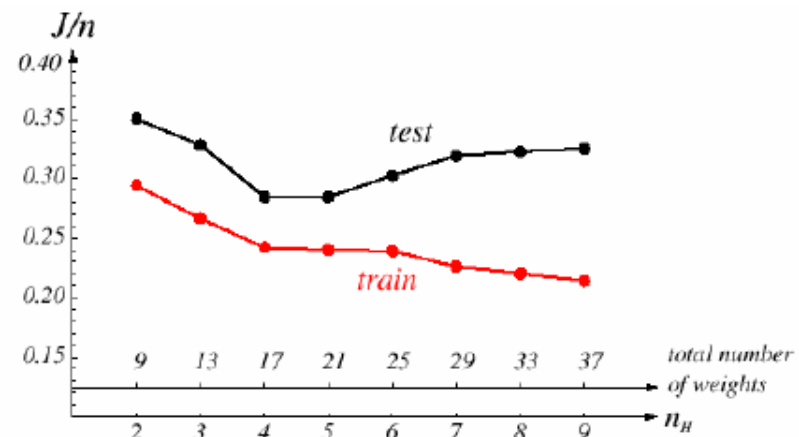


# Implementation Issues 1

- Choosing an activation function
- Input normalization
  - If two features vary orders of magnitude in their values, network cannot learn effectively. For stability reasons, individual features need to be in the same order of magnitude.
- Target Values
  - Usually 0/1 for sigmoid function and -1/1 for tanh.
- Number of Hidden Units
  - It defines the *expressive power* of the network.
    - Too many units cause overfitting.
    - Too few may not be able to solve a complicated problem.
    - A rule of thumb is to choose number of hidden units such that the total number of weights remains less than  $N/10$  where  $N$  is the total number of training samples.

Network trained with 90 2D features from 2 categories  $\rightarrow n = 180$

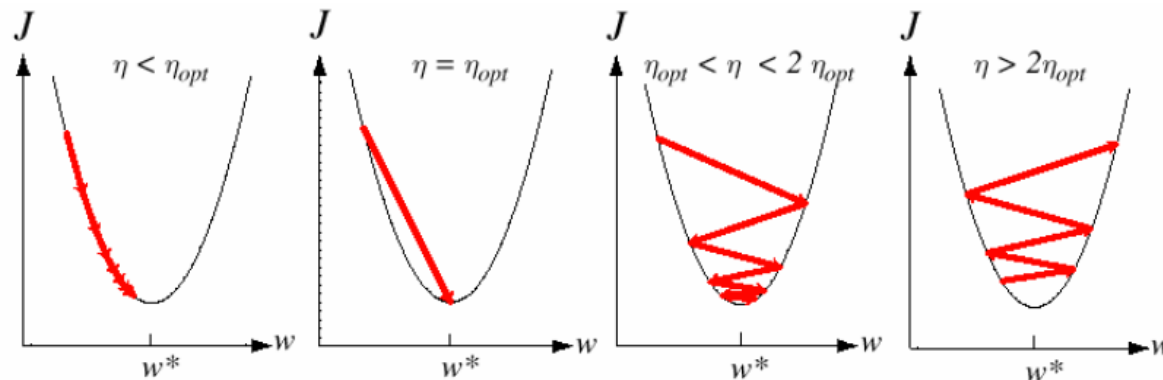
The minimum of the test error occurs in the range  $N \rightarrow [17 \ 21]$  corresponding to number of hidden units  $\rightarrow [4 \ 5]$





# Implementation Issues 2

- Initializing Weights
  - Determines the final solution in the case of a complex error surface.
  - A rule of thumb is to randomly choose the weights from a uniform distribution:
- Learning Rates
  - In theory only affects convergence time, but in practice can lead to divergence:



- Momentum
  - Adding a momentum term can control the speed of learning
- Stopping Criterion
  - The algorithm should be stopped before it reaches its minimum error, because too small error causes the noise in the data to be learned at the expense of the general pattern.
- Regularization
  - Is the smoothing of the error curve so that the optimum solution can be found.

# Recap

- Discriminant Functions
- Perceptron Learning
- Gradient Descent
- Minimum Square Error Solution
- Least Mean Square Solution
- Ho-Kashyap Procedure
- Artificial Neural Networks
- Activation Functions and Multilayer Perceptrons
- Backpropagation Algorithm
- Implementation Issues