Data Classification

3.3.1. .A.V.M. .4.

Linear Discriminant Functions

Suppose we wished to decide whether some data

$$\mathbf{x} = (x_1, x_2, \ldots, x_d)$$

belonged to one of two categories

One way to do this is to construct a *Discriminant function.* Let g(**x**) define the categories as:

$$\omega(\mathbf{x}) = \begin{cases} \omega_1, g(\mathbf{x}) > 0 \\ \omega_2, g(\mathbf{x}) \leq 0 \end{cases}$$

If g(x) is linear we can write:

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

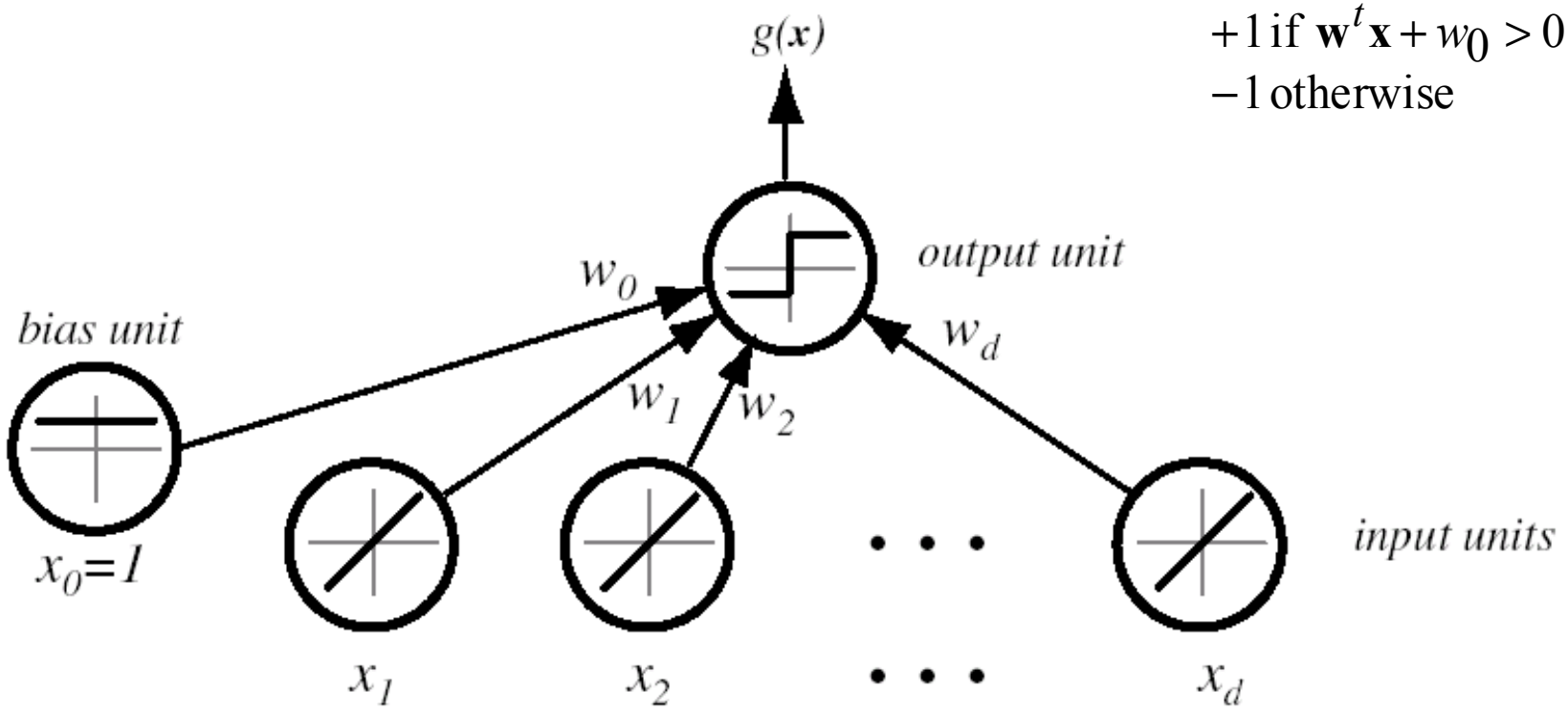
where **w** = {w_1, … , w_d} are called the weights and w_0 is called the bias or threshold weight.

Simple Linear Classifier

Output emits

$+1$ if $\mathbf{w}^t\mathbf{x} + w_0 > 0$

-1 otherwise



Each unit shows its effective input-output function.

Decision surface

g(**x**) = 0 defines a *decision surface* which separates points into ω_1 and ω_2. If g(**x**) is linear, this decision surface is a *hyperplane*.

The hyperplane divides the space into two regions:

R_1 : g(**x**) > 0, hence **x** is in ω_1 and

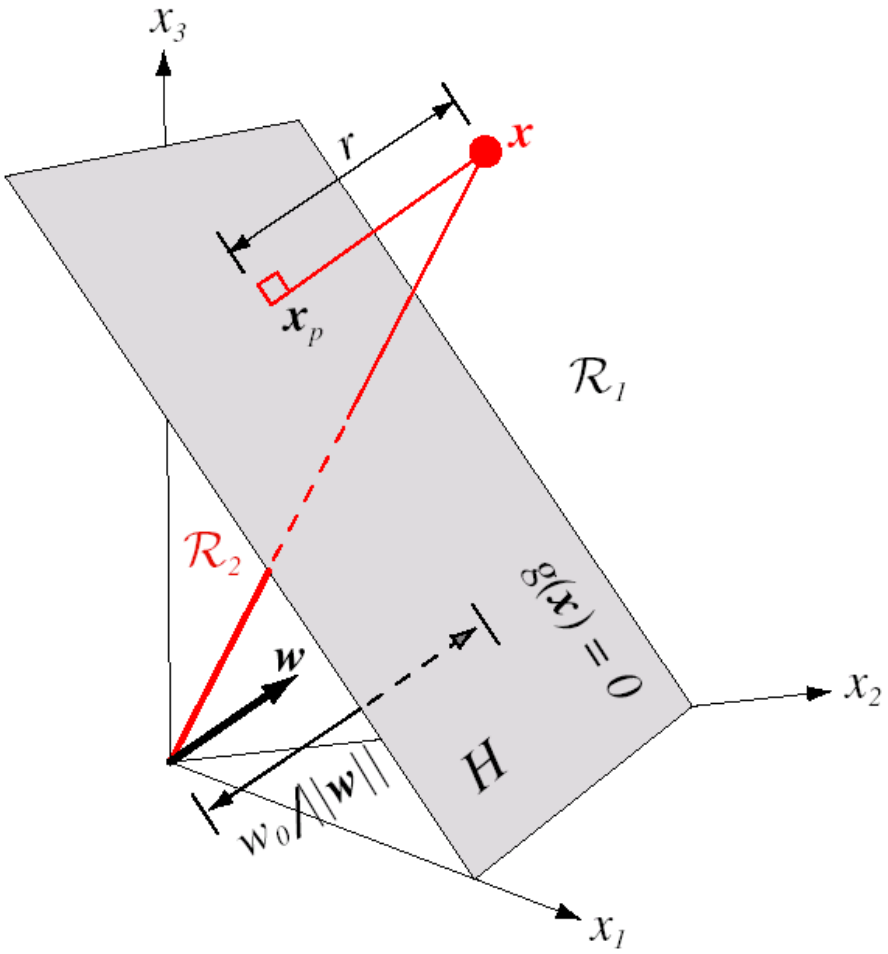
R_2 : g(**x**) ≤ 0, hence **x** is in ω_2

Suppose x_1 and x_2 are both on the decision surface. Then:

$$\mathbf{w}^t\mathbf{x}_1 + w_0 = \mathbf{w}^t\mathbf{x}_2 + w_0 \qquad \text{i.e.} \qquad \mathbf{w}^t(\mathbf{x}_1 - \mathbf{x}_2) = 0.$$

Therefore **w** is normal (orthogonal) to the hyperplane.

Hyperplane decision surface



Let us write

$$\mathbf{x} = \mathbf{x}_p + r\frac{\mathbf{w}}{\|\mathbf{w}\|}$$

where $\mathbf{x}_p$ is normal projection

of $\mathbf{x} \rightarrow H$ and r is distance from H

($r > 0$ on + ve side, $r < 0$ on - ve)

Since $g(\mathbf{x}_p) = 0$,

$$g(\mathbf{x}) = \mathbf{w}^t\mathbf{x} + w_0 = r\|\mathbf{w}\|$$

$$\Rightarrow r = g(\mathbf{x})/\|\mathbf{w}\|$$

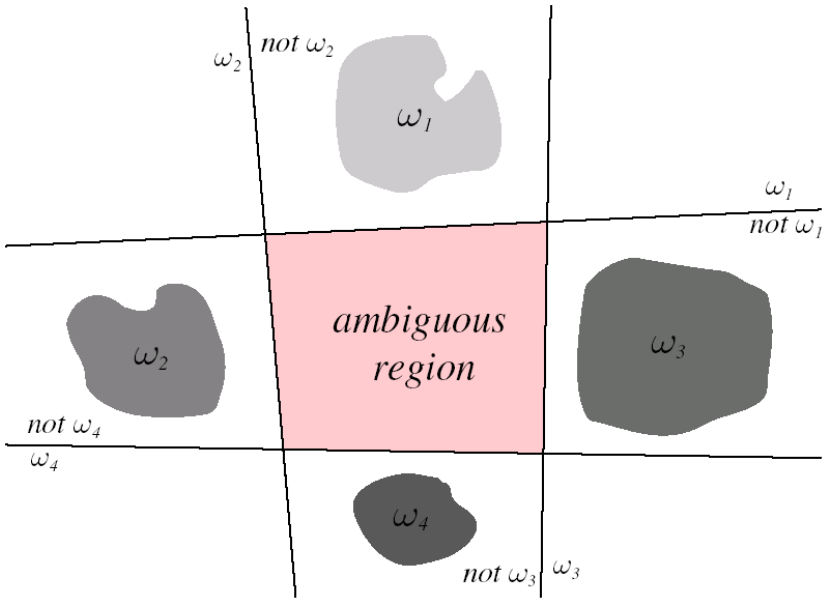
$g(\mathbf{x})$ measures dist from $\mathbf{x}$ to H.

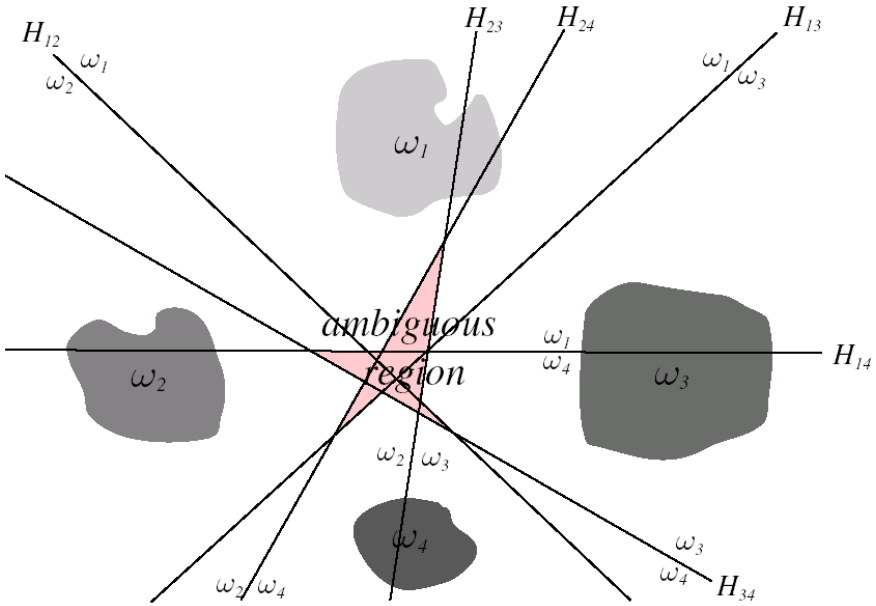
Multicategory Case

Several ways to make classifiers for $c>2$ categories, e.g.

- Reduce to c 2-class problems (in w_i vs not in w_i).
- Use $c(c\text{-}1)/2$ discriminants, one for each pair of classes

These both lead to areas of ambiguity:





Linear Machine

We define c linear discriminant functions:

$$g_i(\mathbf{x}) = \mathbf{w}_i^t \mathbf{x} + w_{i0} \qquad i = 1, \ldots, c$$

And assign **x** to ω_i if $g_i(\mathbf{x}) > g_j(\mathbf{x})$ for all $j \neq i$.

This classifier is called a *linear machine*, dividing the feature space into c regions with $g_i(\mathbf{x})$ largest is region R_i.

The boundary hyperplane H_{ij} between regions is defined by:

$$g_i(\mathbf{x}) = g_j(\mathbf{x}) \quad \text{i.e.} \quad (\mathbf{w}_i - \mathbf{w}_j)^t \mathbf{x} + (w_{i0} - w_{j0}) = 0.$$

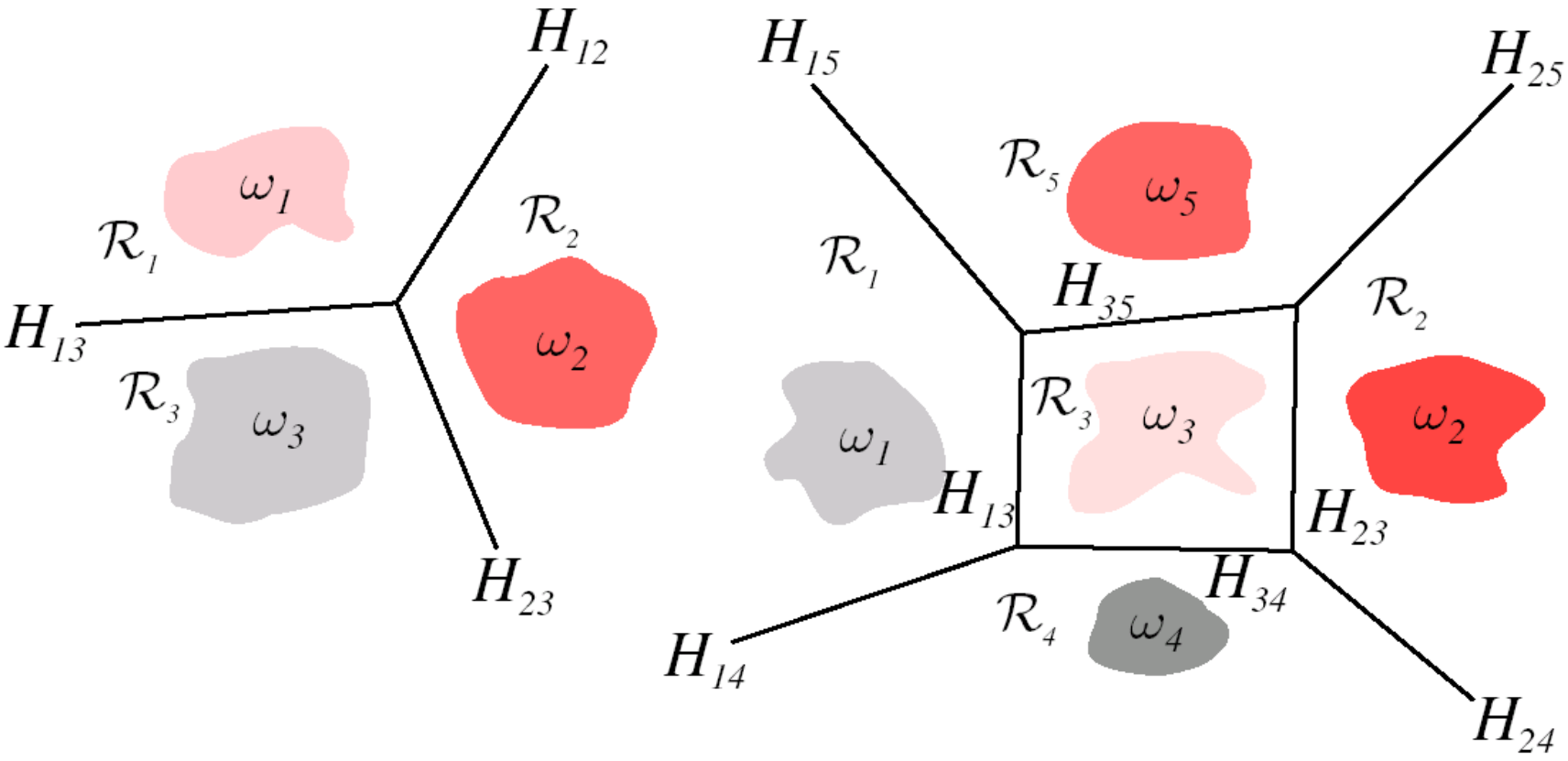
So the weight difference $\mathbf{w}_i$-$\mathbf{w}_j$ is normal to H_{ij} and the distance from the hyperplane is:

$$(g_i(\mathbf{x}) - g_j(\mathbf{x})) / \| \mathbf{w}_i - \mathbf{w}_j \|$$

Decision boundaries for Linear Machine

3 categories

5 categories



Generalized Linear Discriminants

We can generalize g(**x**) by adding terms $x_i x_j$ to give a quadratic (nonlinear) discriminant function:

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

Can generalize to cubic, etc.

We can view this as a linear discriminant function in a new space.

Let $y_i(\mathbf{x})$ define a new variable as a (nonlinear) function of **x**.

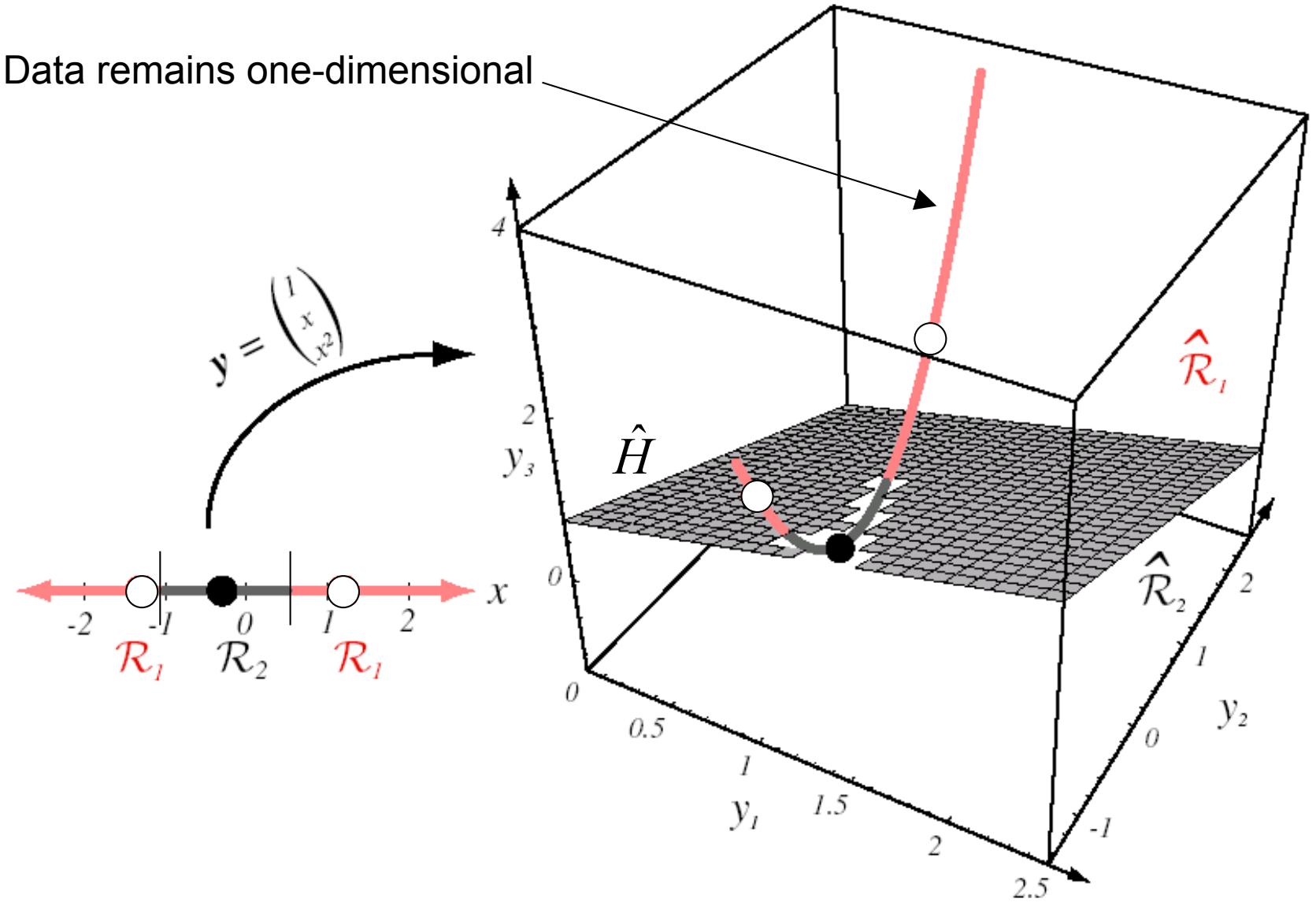
$$g(\mathbf{x}) = \sum_{i=1}^{\hat{d}} a_i y_i(\mathbf{x})$$

Note we have absorbed the bias weight in this formulation - A process called *augmentation*.

e.g. $g(\mathbf{x}) = w_0 + w_1 x + w_{11} x^2 = a_1 y_1 + a_2 y_2 + a_3 y_3$

with $\mathbf{y} = (1, x, x^2)$, $\mathbf{a} = (w_0, w_1, w_{11})$.

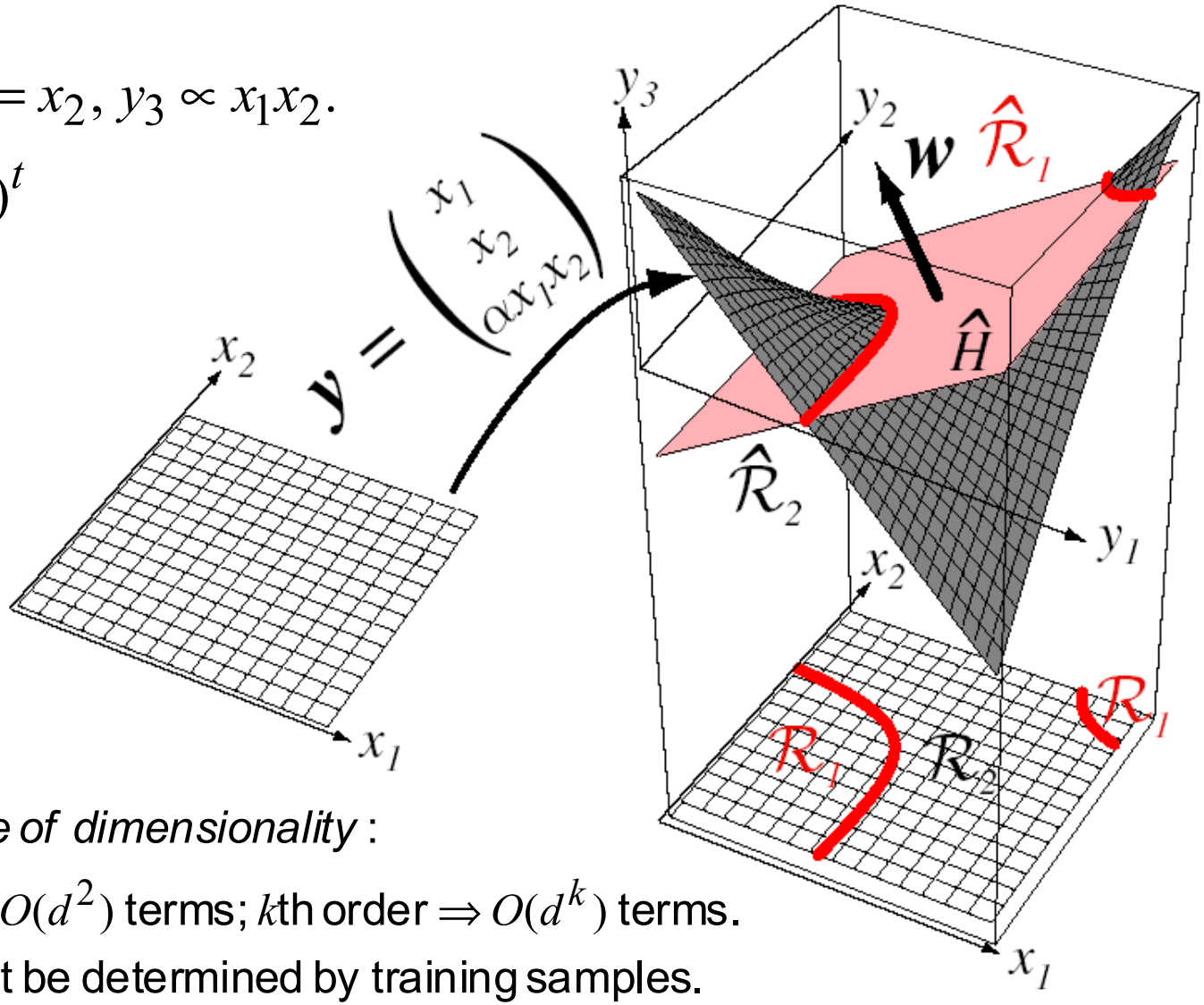
Quadratic discriminant: 1-d case



Quadratic discriminant: 2-d case

$y_1 = x_1, y_2 = x_2, y_3 \propto x_1 x_2.$

$\mathbf{a} = (-1, 1, 2)^t$



Beware *curse of dimensionality* :

Quadratic $\Rightarrow O(d^2)$ terms; kth order $\Rightarrow O(d^k)$ terms.

Weights must be determined by training samples.

Linearly Separable Case (2 category)

Suppose we have n samples $y_1,\ldots y_n$, each labelled either ω_1 or ω_2 and we wish to *learn* a discriminant function $g(\mathbf{y}) = \mathbf{a}^T\mathbf{y}$, that correctly classifies the data.

Sample $\mathbf{y}_i$ is correctly classified if

$$\mathbf{a}^T\mathbf{y}_i > 0 \text{ when } \mathbf{y}_i \text{ is labelled } \omega_1$$

or

$$\mathbf{a}^T\mathbf{y}_i < 0 \text{ when } \mathbf{y}_i \text{ is labelled } \omega_2$$

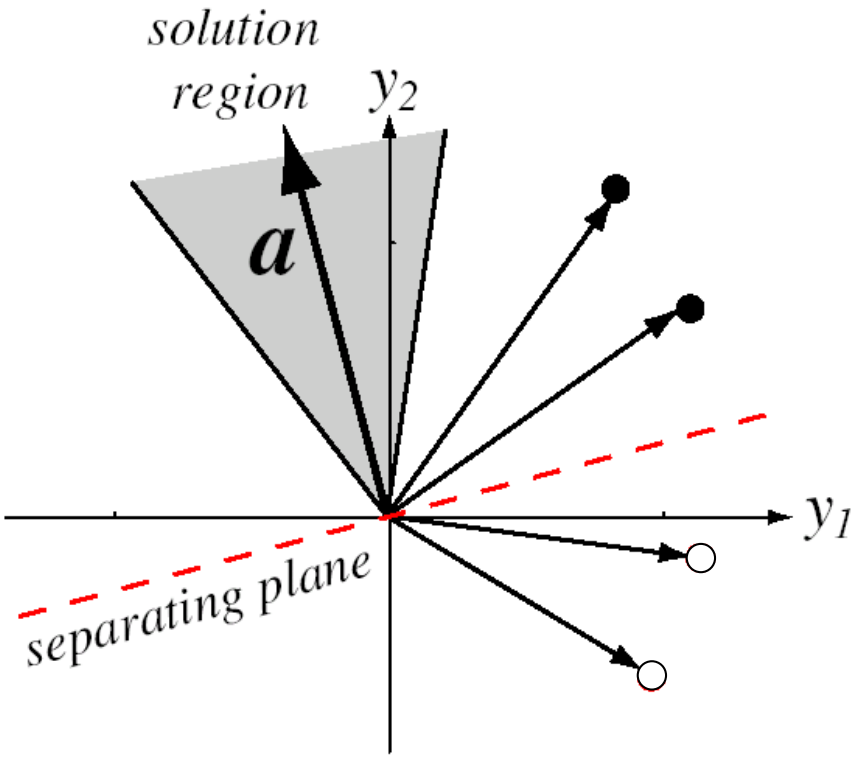
A data set is called *linearly separable* if there exists a vector a which correctly classifies all samples. This is called a *separating vector* or *solution vector.*

Includes the case of $g(\mathbf{y}) = \mathbf{a}^T\mathbf{y}+a_0$ by augmentation: $\mathbf{y}' \equiv [\mathbf{y},1]^T, \mathbf{a}' \equiv [\mathbf{a}, a_0]^T$

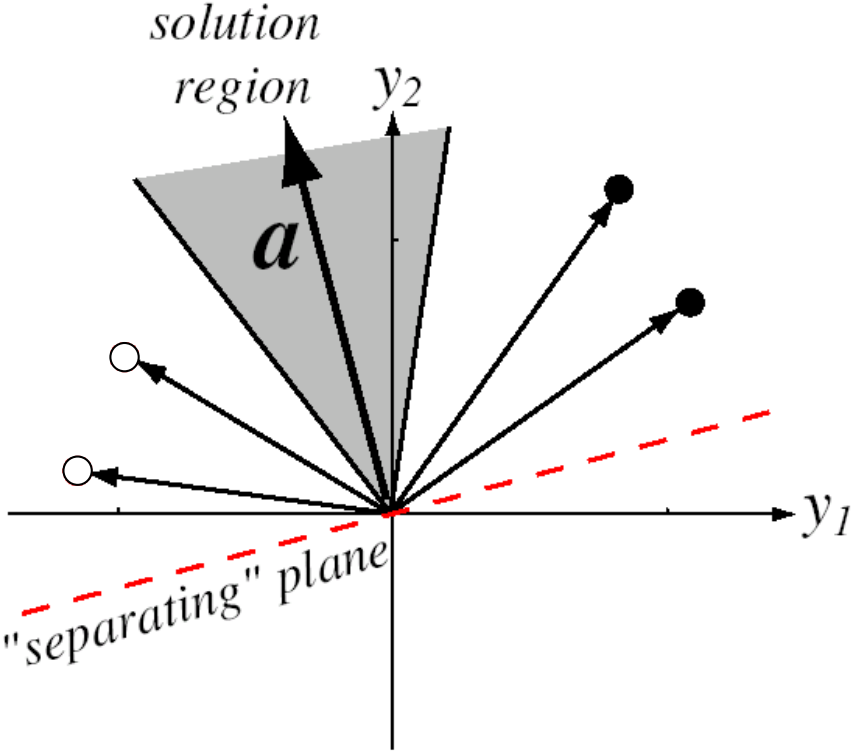
Normalization (2 category)

In the 2-category problem it is possible to *normalize* the data: negate all $\mathbf{y}_i$s labelled as ω_2. Then we require: $\mathbf{a}^T\mathbf{y}_i > 0$ for all samples

Original training samples



Normalized training samples

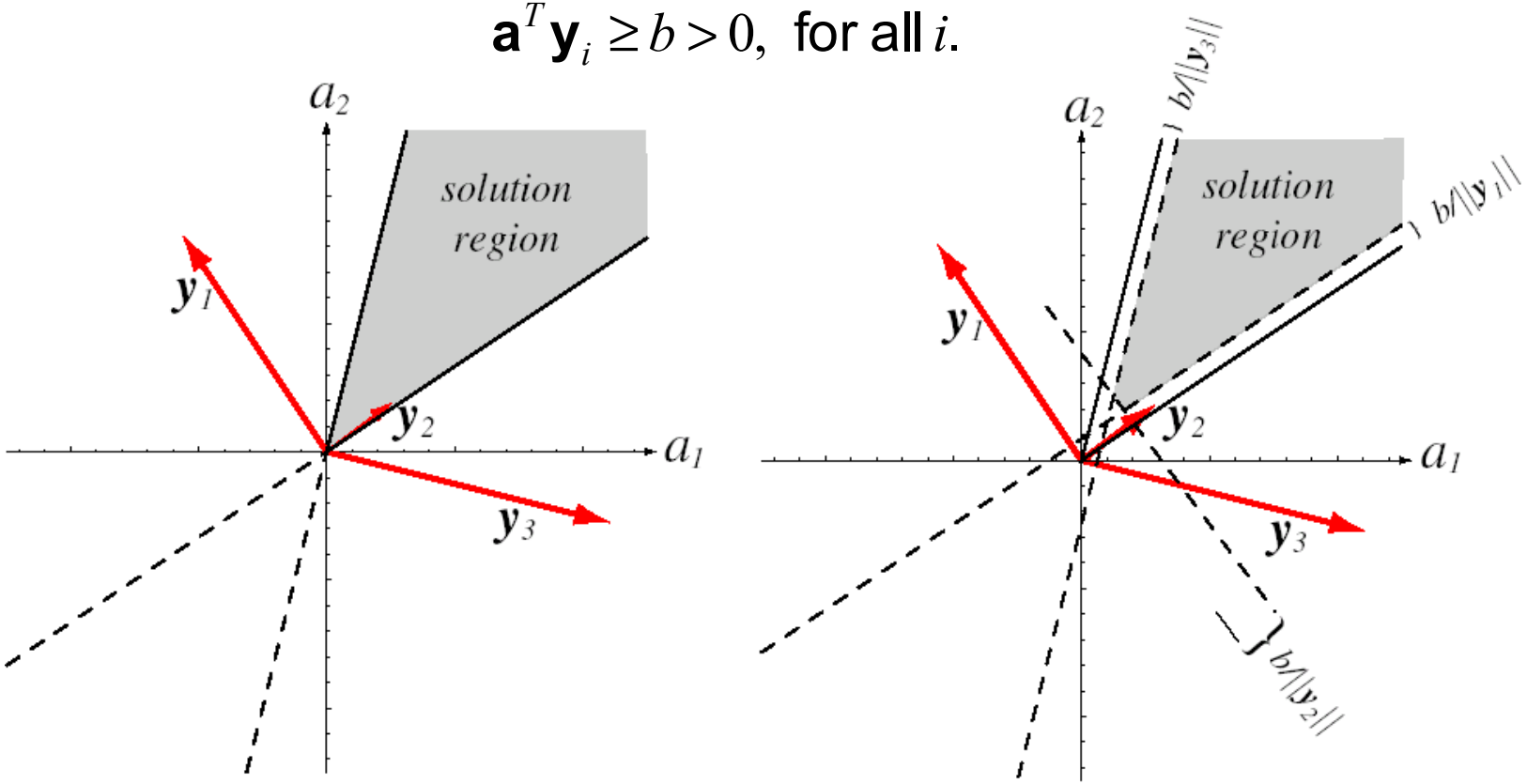


Vector $\mathbf{a}$ specifies a point in *weight space*.

Margin

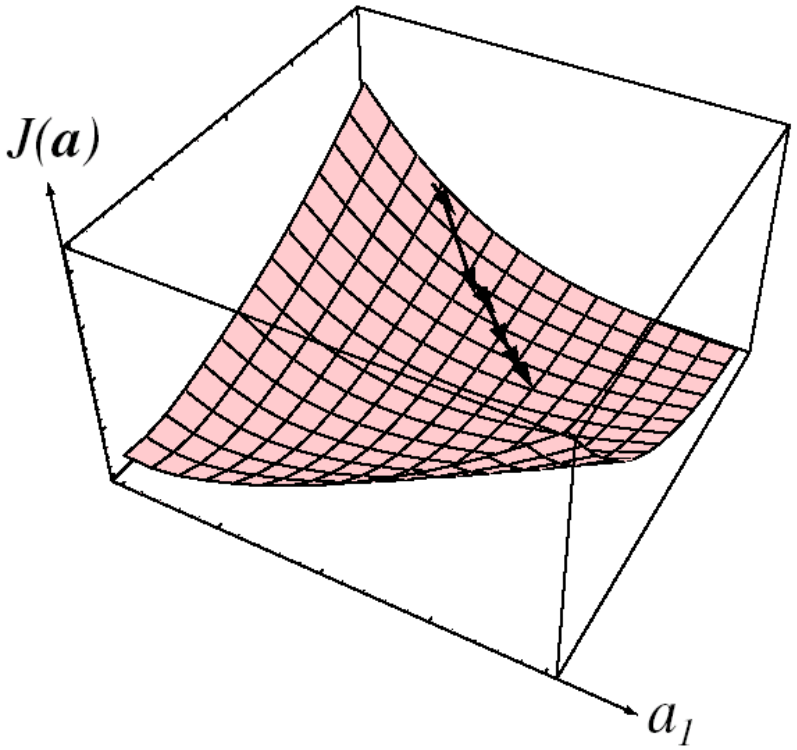
The solution vector may not be unique. However some may be 'close' to making the wrong decision. On approach is introduce a *margin*, *b*. i.e. require

$$\mathbf{a}^T \mathbf{y}_i \geq b > 0, \text{ for all } i.$$



Gradient Descent

Define a criterion (cost) function *J(**a**)* in terms of the data $\mathbf{x}_n$ which is minimized if a is a solution vector. Learning a classifier is then reduced to a minimization problem and can be solved by e.g. gradient descent.



Update : $\mathbf{a}(k+1) = \mathbf{a}(k) - \eta(k)\nabla J(\mathbf{a}(k))$
Learning rate $\eta(k)$
Gradient vector $\nabla J(\cdot)$

Basic gradient descent algorithm :
1. Initialize $k \leftarrow 0, \mathbf{a}(1), \theta, \eta(.)$
2. $k \leftarrow k+1$
3. $\mathbf{a} \leftarrow \mathbf{a} - \eta(k)\nabla J(\mathbf{a})$
4. If $|\eta(k)\nabla J(\mathbf{a})| > \theta$, repeat from 2.

Perceptron Criterion

Could J = number of samples misclassified?

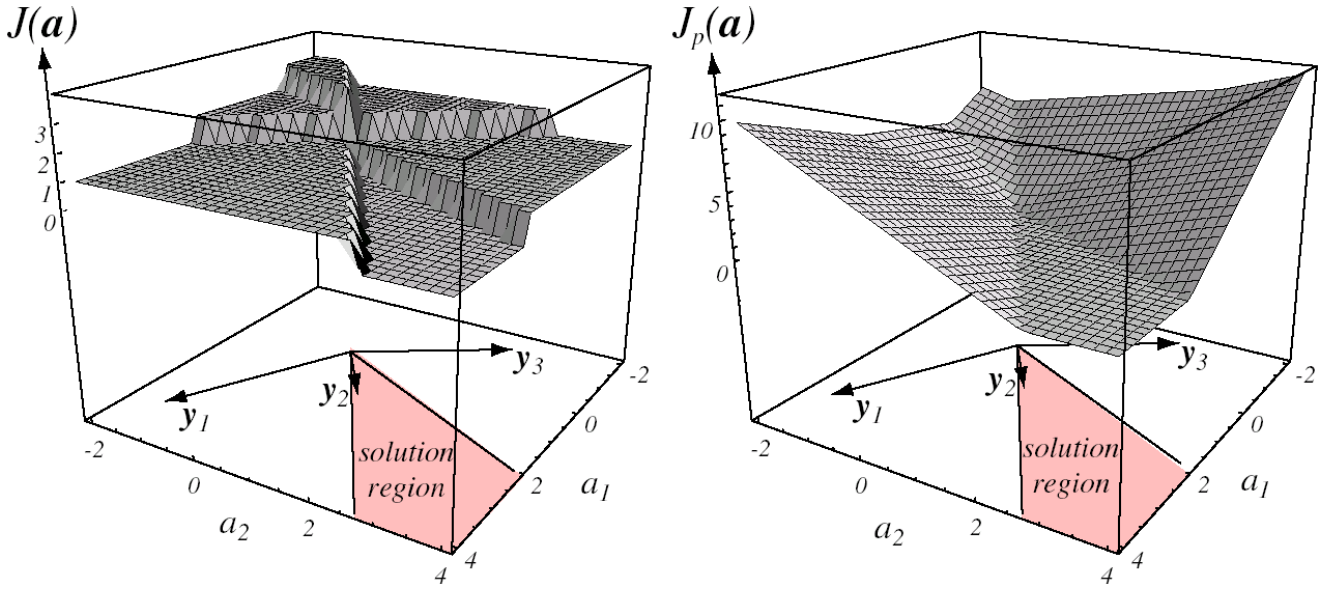
No – the function is piecewise constant, hence impossible to search.

Instead use:

$$J_p(\mathbf{a}) = \sum_{\mathbf{y} \in Y_m} (-\mathbf{a}^T \mathbf{y})$$

warning: Data are assumed normalized

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



Perceptron Algorithm

$$[\nabla J_p(\mathbf{a})]_i = \partial J_p(\mathbf{a})/\partial a_i = \sum_{\mathbf{y}\in Y_m}(-y_i)$$

warning: Data are assumed normalized

i.e. $\nabla J_p(\mathbf{a}) = \sum_{\mathbf{y}\in Y_m}(-\mathbf{y})$ so gradient descent rule is :

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

where Y_m is set of samples misclassified by $\mathbf{a}(k)$.

Batch Perceptron Algorithm

1. Initialize $\mathbf{a}, \eta(\cdot)$, stopping criterion $\theta, k \leftarrow 0$
2. $k \leftarrow k+1$
3. $\mathbf{a} \leftarrow \mathbf{a} + \eta(k)\sum_{\mathbf{y}\in Y_m}\mathbf{y}$
4. If $|\eta(k)\sum_{\mathbf{y}\in Y_m}\mathbf{y}| > \theta$, repeat from 2.

η(k) is the learning rate or step size

Perceptron Convergence

It is possible to prove the following:

Perceptron Convergence Theorem (Rosenblatt, 1958)

If training samples are linearly separable, then the sequence of weight vectors given by the “Perceptron Algorithm” (Single-Sample Perceptron Algorithm) will terminate at a solution vector. See Duda, Hart and Stork.

(If the training samples are not linearly separable, the Perceptron Algorithm will not terminate).

Similarly can show that the Batch Perceptron Algorithm always converges.

Nonseparable Samples

A general dataset is likely to be *not* linearly separable. This means:

- there is no “correct” separating hyperplane
- the Perceptron algorithm will never finish

But – we still want to find something: a sort of “least worst” solution

Hence we introduce a different criterion:

Minimum Squared Error (MSE)

Minimum Squared Error

The perceptron criterion focussed on ***misclassified*** samples. Here we will consider a cost function involving ***all*** samples.

Let b_i denote a desired margins (e.g. $b_i = \pm 1$). We then try to get the solution as close as possible to these margins. That is we want:

$$\mathbf{a}^T \mathbf{y}_i = b_i \quad \text{for all } i$$

Writing this is matrix vector form we want **Ya**=**b** or

$$\begin{pmatrix} y_{10} & \cdots & y_{1d} \\ \vdots & \ddots & \vdots \\ y_{n0} & \cdots & y_{nd} \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_d \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

If Y is square and invertible we can solve for **a** by: $\mathbf{a} = \mathbf{Y}^{-1}\mathbf{b}$. When Y is rectangular (number of samples > dim a) it is called *overdetermined*: an exact solution may not be possible.

Sum squared error

Sum squared error criterion:

$$J_s(\mathbf{a}) = \| \mathbf{e} \|^2 = \| \mathbf{Ya} - \mathbf{b} \|^2 = \sum_{i=1}^{n} (\mathbf{a}^t \mathbf{y}_i - b_i)^2$$

We wish to find the a that minimizes $J(\mathbf{a})$. This is a classical optimization problem. As with the perceptron criterion we could calculate the gradient

$$\nabla J_s = \sum_{i=1}^{n} 2(\mathbf{a}^t \mathbf{y}_i - b_i)\mathbf{y}_i = 2\mathbf{Y}^t(\mathbf{Ya} - \mathbf{b})$$

and use gradient descent.

However, in this case, there is also a direct solution.

Pseudoinverse

Equating the gradient to zero

$$\nabla J_s \equiv 2\mathbf{Y}^T(\mathbf{Ya}-\mathbf{b}) = 0$$

and solving we have: $\mathbf{Y}^t\mathbf{Ya} = \mathbf{Y}^t\mathbf{b}$ and

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

The matrix $\mathbf{Y}^T\mathbf{Y}$ is square and typically invertible. The matrix

$$\mathbf{Y}^+ \equiv (\mathbf{Y}^t\mathbf{Y})^{-1}\mathbf{Y}^T$$

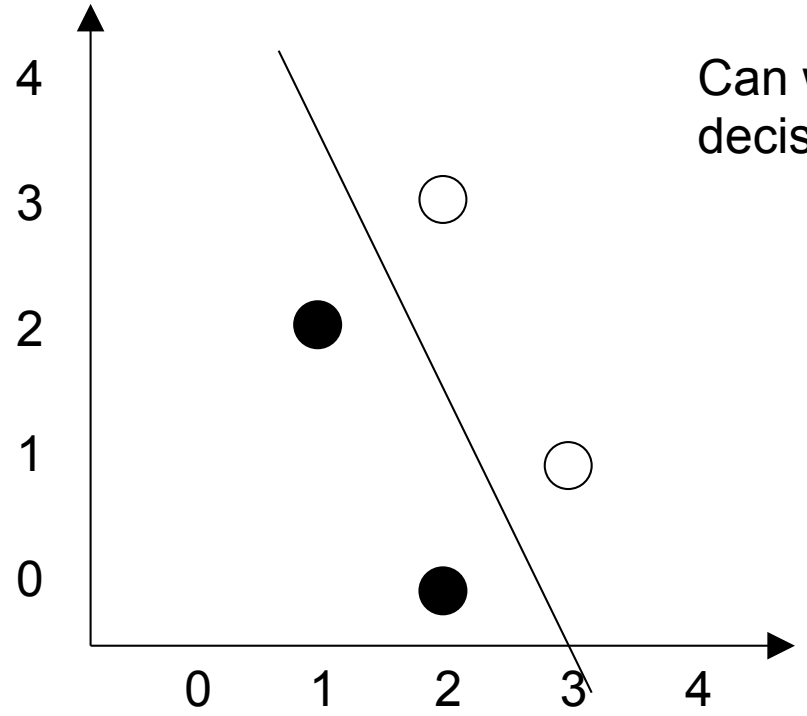
is called the *pseudoinverse* of Y.

Example

Suppose we have these 2D points for two categories:

ω_1: $(1,2)^t$ $(2,0)^t$

ω_2: $(3,1)^t$ $(2,3)^t$



Can we find the “obvious” decision boundary?

Example (cont)

Step1. *Augment* the data points : $\hat{\mathbf{y}}_i = (1, x_{i1}, x_{i2})$

ω_1 : $(1,2)^t \quad (2,0)^t \quad \Rightarrow \quad \hat{\mathbf{y}}_1 = (1,1,2)^t, \quad \hat{\mathbf{y}}_2 = (1,2,0)^t$

ω_2 : $(3,1)^t \quad (2,3)^t \quad \Rightarrow \quad \hat{\mathbf{y}}_3 = (1,3,1)^t, \quad \hat{\mathbf{y}}_4 = (1,2,3)^t$

Step 2. Normalize the data (negate $\mathbf{y}_i$s for class ω_2) :

$$\mathbf{y}_1 = (1,1,2)^t, \qquad \mathbf{y}_2 = (1,2,0)^t,$$

$$\mathbf{y}_3 = (-1,-3,-1)^t, \quad \mathbf{y}_4 = (-1,-2,-3)^t$$

giving

$$\mathbf{Y} = \begin{pmatrix} 1 & 1 & 2 \\ 1 & 2 & 0 \\ -1 & -3 & -1 \\ -1 & -2 & -3 \end{pmatrix}$$

Example (cont)

Step 3. Calculate the pseudoinverse :

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

[Can do this by hand - calculate $\mathbf{Y}^{T}\mathbf{Y}$, then invert it

using $\mathbf{M}^{-1} = Adj(\mathbf{M}) / |\mathbf{M}|$

where $Adj(\mathbf{M})$ is adjoint and $|\mathbf{M}|$ is determinant.]

Step 4. Multiply out $\mathbf{a} = \mathbf{Y}^{+}\mathbf{b}$:

In this case we use $\mathbf{b} = (1, 1, 1, 1)^{T}$.

Multiplying out gives us $\mathbf{a} = \mathbf{Y}^{+}\mathbf{b} = (11/3, -4/3, -2/3)^{T}$

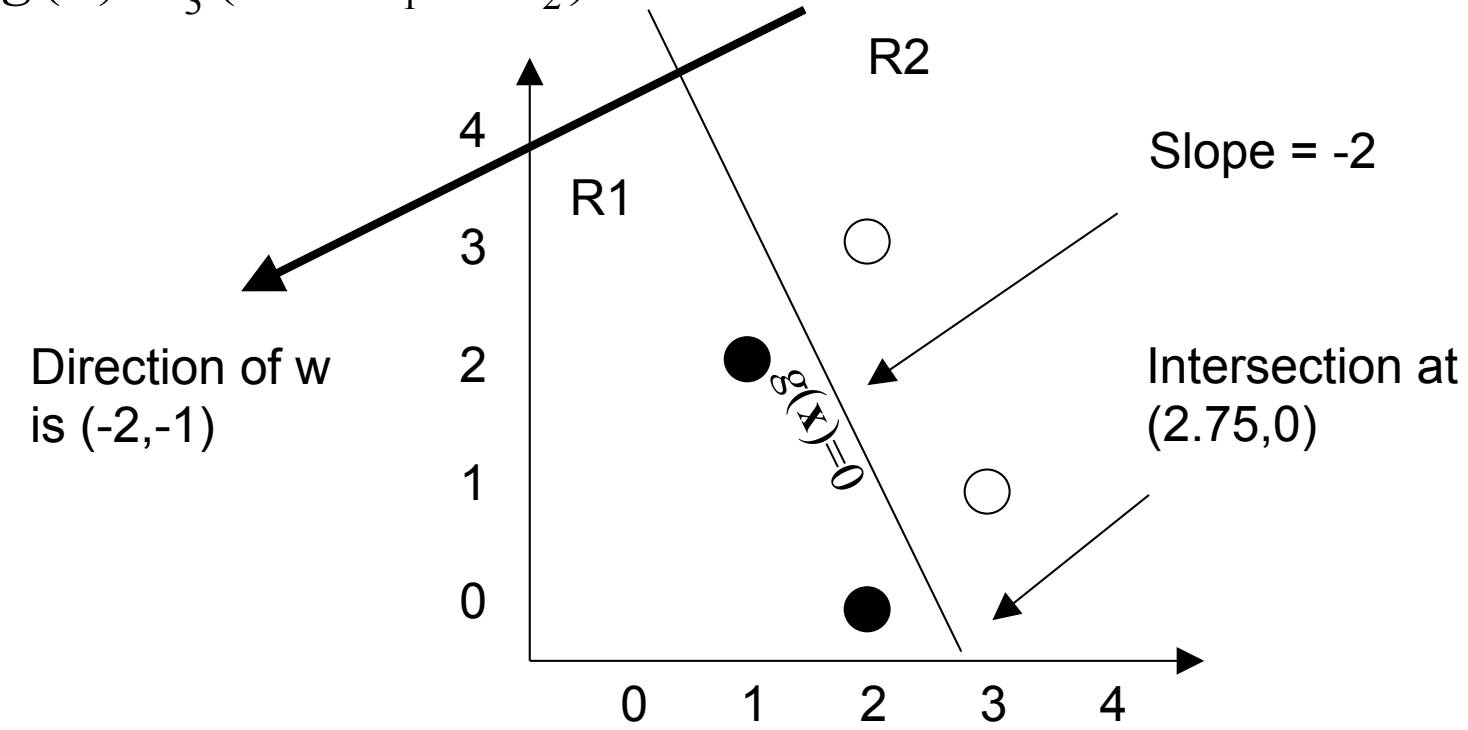
Example (cont)

Step 5. Express in required form

$$\mathbf{a} = (11/3, -4/3, -2/3)^T = (w_0, w_1, w_2)^T$$

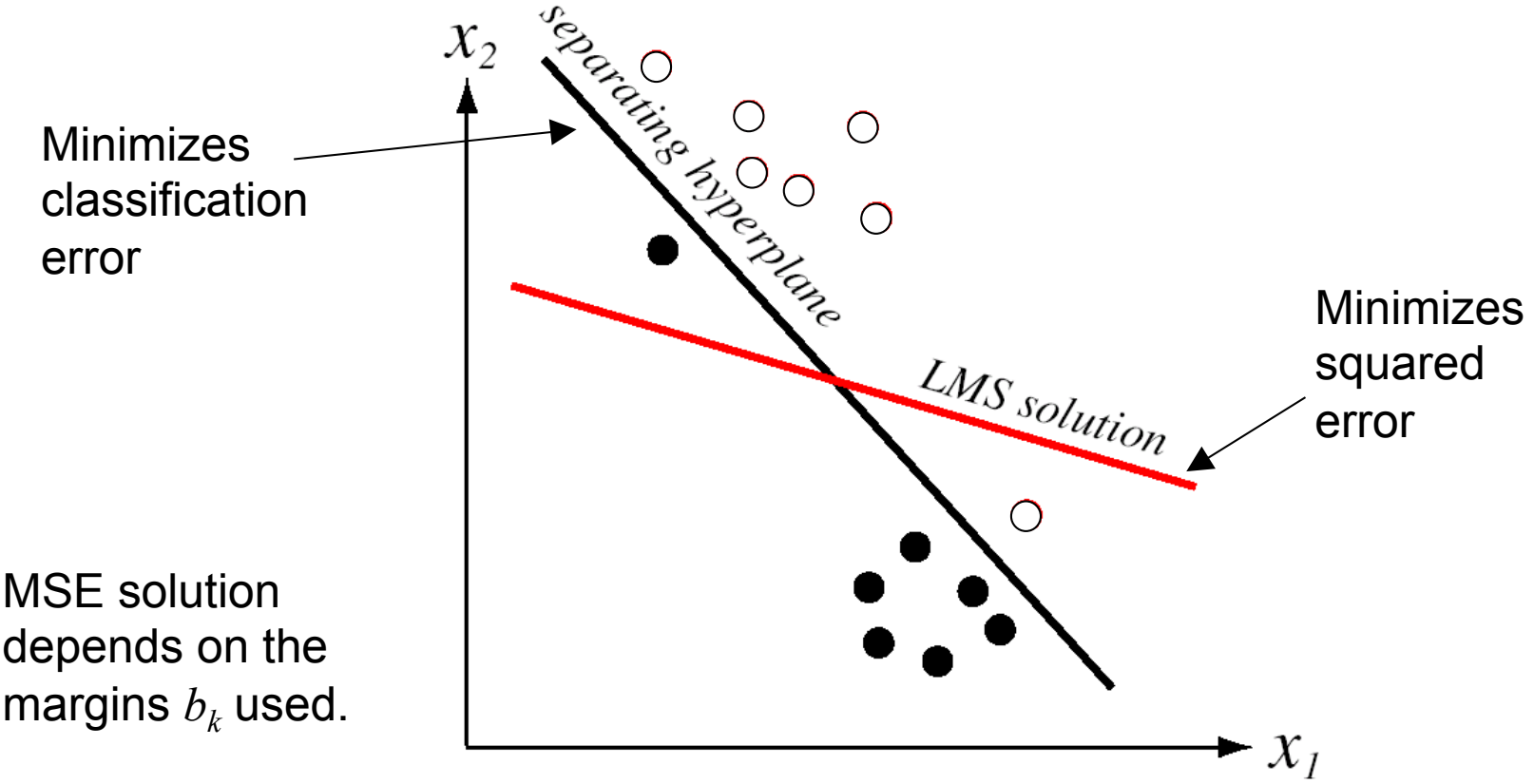
So we have

$$g(\mathbf{x}) = \tfrac{1}{3}(11 - 4x_1 - 2x_2)$$



MSE and Separating Hyperplanes

Minimizing the squared error need not converge to a separating hyperplane solution, *even if one exists*.



Learning in the Multicategory Case

Both MSE and Perceptron learning can be extended to multi-category systems.

For MSE, if we find the MSE solution to:

$$\mathbf{a}_i^t \mathbf{y} = 1 \quad \text{for all } \mathbf{y} \in Y_i \qquad \text{(i.e. } \mathbf{y} \text{ in category } i\text{)}$$

$$\mathbf{a}_i^t \mathbf{y} = 0 \quad \text{for all } \mathbf{y} \notin Y_i \qquad \text{(}\mathbf{y} \text{ not in category } i\text{)}$$

then it turns out that $\mathbf{a}_i^T \mathbf{y}$ is asymptotically (for a sufficient number of samples) an MSE approximation to:

$$P(\omega_i \mid \mathbf{x})$$

The probability of x being labelled ω_i. Therefore the decision rule:

$$\omega(\mathbf{y}) = \omega_i \text{ if } \mathbf{a}_i^T \mathbf{y} > \mathbf{a}_j^T \mathbf{y} \text{ for all } j \neq i$$

assigns the most probably category (more later).

This MSE solution is also a *linear machine*

The Pseudoinverse solution

We can now construct the multicategory pseudoinverse solution.
Let **Y** be the set of samples partitioned into c sub-matrices:

$$\mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_c \end{bmatrix} \qquad \text{samples labelled } i \text{ are rows of } \mathbf{Y}_i$$

Let $\mathbf{A}=[\mathbf{a}_1,\ldots,\mathbf{a}_c]$ be the matrix of weight vectors and define

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \\ \vdots \\ \mathbf{B}_c \end{bmatrix}, \ \mathbf{B}_i \text{ is } 0, \text{ except } i\text{th column is } 1.$$

The squared error: $\| (\mathbf{YA} - \mathbf{B}) \|_F^2 = \sum_i \| (\mathbf{Y}_i \mathbf{a}_i - \mathbf{b}_i) \|^2$

is minimized by: $\mathbf{A} = \mathbf{Y}^+ \mathbf{B}$

Multilayer Neural Networks

Linear discriminants are good for many problems but not general enough for demanding applications. We saw that we can get more complex decision surfaces with nonlinear pre-processing,

$$y_i = \varphi_i(\mathbf{x})$$

Where $\varphi_i(.)$, is, for example, a polynomial expansion to some order k.

But these may have too many free parameters, so we may not have enough data points to fix them. Instead we can try to *learn* which nonlinearities to use.

The best-known method is based on gradient descent: the so-called *backpropagation* algorithm.

Three-Layer Network

Network has:

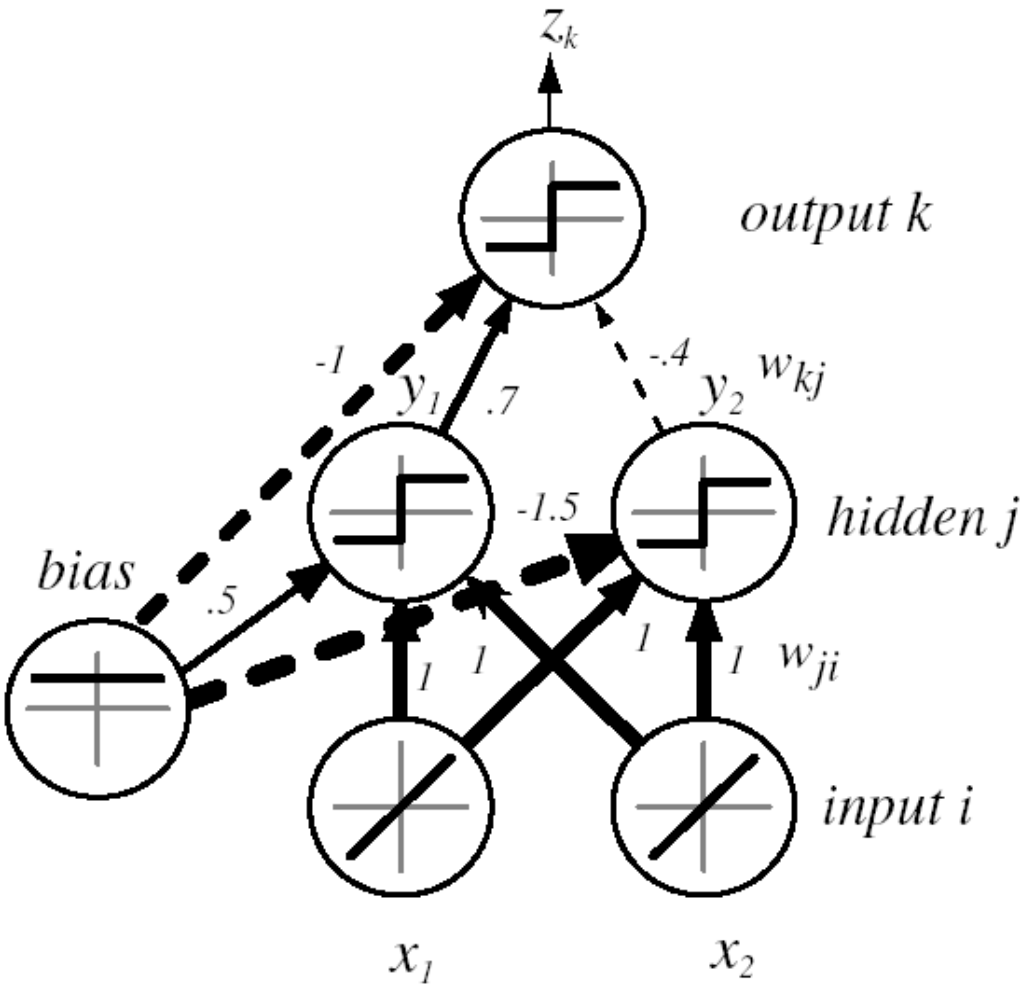
- Input layer
- Hidden layer
- Output layer

with adjustable weights between layers

Also:

- Bias unit

with weights to all hidden and output units.



Biological terms sometimes used:
"neuron" = unit; "synapses" = connection; "synaptic weight" = weight.

Operation

Step 1: each d-dimensional input vector ($x_1,\ldots,x_d$) is presented to input layer of the network and augmented with a bias term $x_0 = 1$ to give $x = (x_0, x_1,\ldots,x_d)$

Step 2: at each hidden layer we calculate the weighted sum of inputs to give the net activation:

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} = \mathbf{w}_j^T \mathbf{x}$$

where w_{ji} is the weight from the input unit *i* to the hidden unit *j*

Step 3: The hidden unit emits the output: $y_j = f(net_j)$

where *f*(.) is some nonlinear *activation function*

Operation (cont)

Step 4: At each output unit we calculate the weighted sum of the hidden layer units it is connected to giving:

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \mathbf{w}_k^T \mathbf{y}$$

where w_{kj} is the weight from the hidden unit *j* to the output unit *k*

Step 5: each output unit emits $z_k = f(net_k)$

where *f*(.) is again the nonlinear *activation function*

We can therefore think of the network as calculating c discriminant functions:

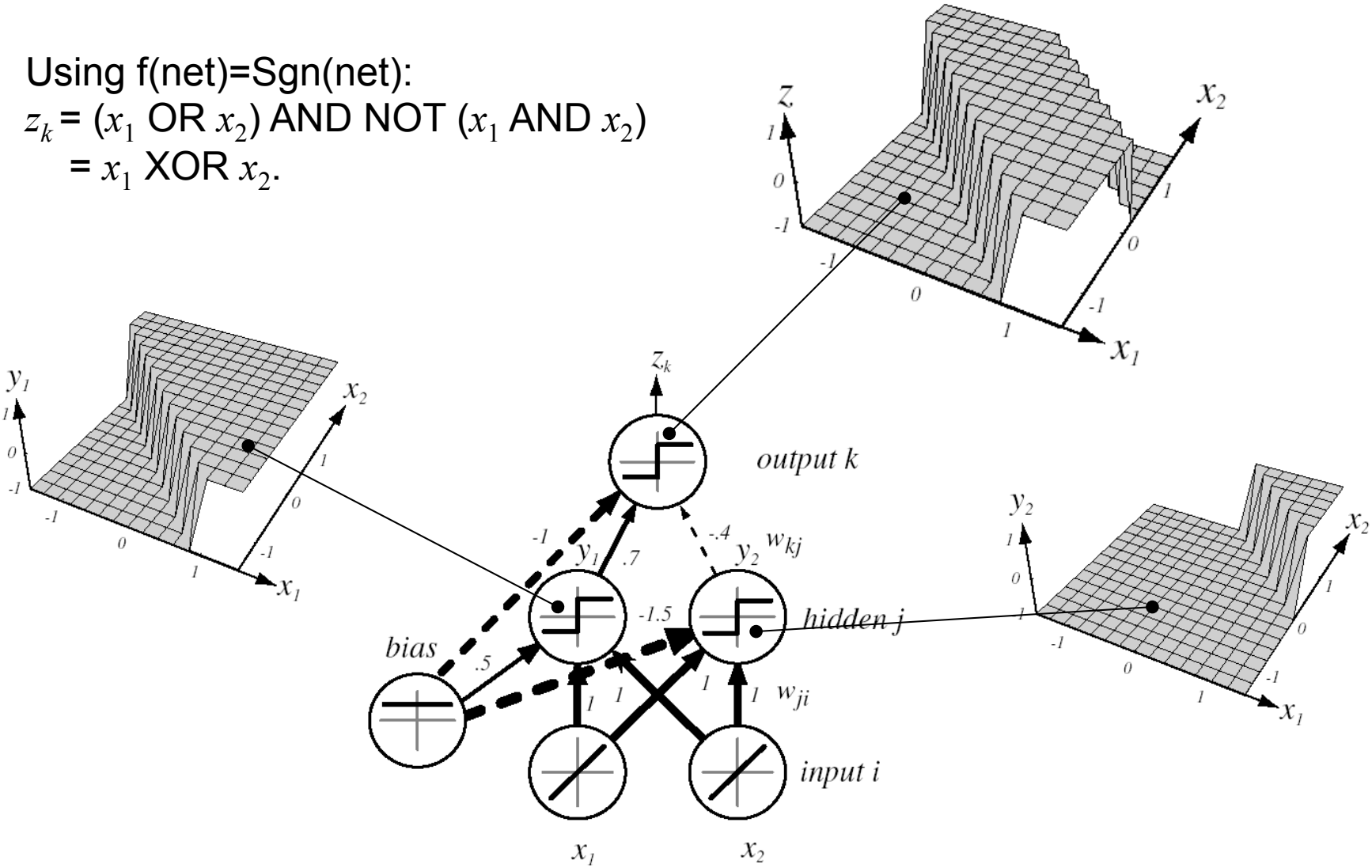
$$z_k = g_k(\mathbf{x})$$

Example: XOR Problem

Using f(net)=Sgn(net):

z_k = (x_1 OR x_2) AND NOT (x_1 AND x_2)

= x_1 XOR x_2.



General Feedfoward Operation

General form of output discriminant functions:

$$g_k(\mathbf{x}) \equiv z_k = f\left(\sum_{j=1}^{n_H} \tilde{w}_{kj} f\left(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}\right) + \tilde{w}_{k0}\right)$$

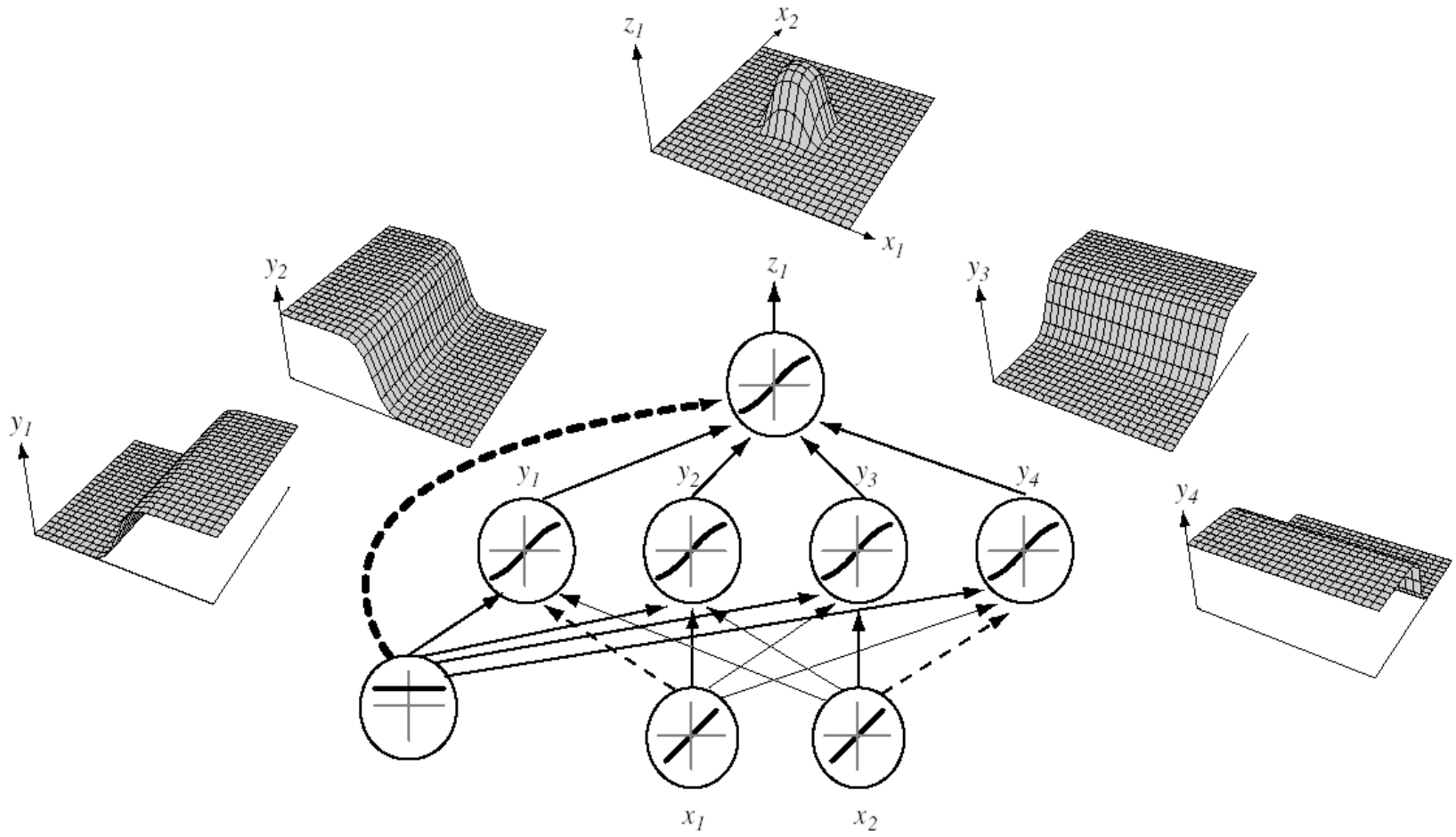
[Note that the $\tilde{w}_{kj}$s are different from the w_{ji}s.]

Question: can every decision be implemented with a 3-layer network?

Answer [Kolmogorov + others]: (in theory) Yes.

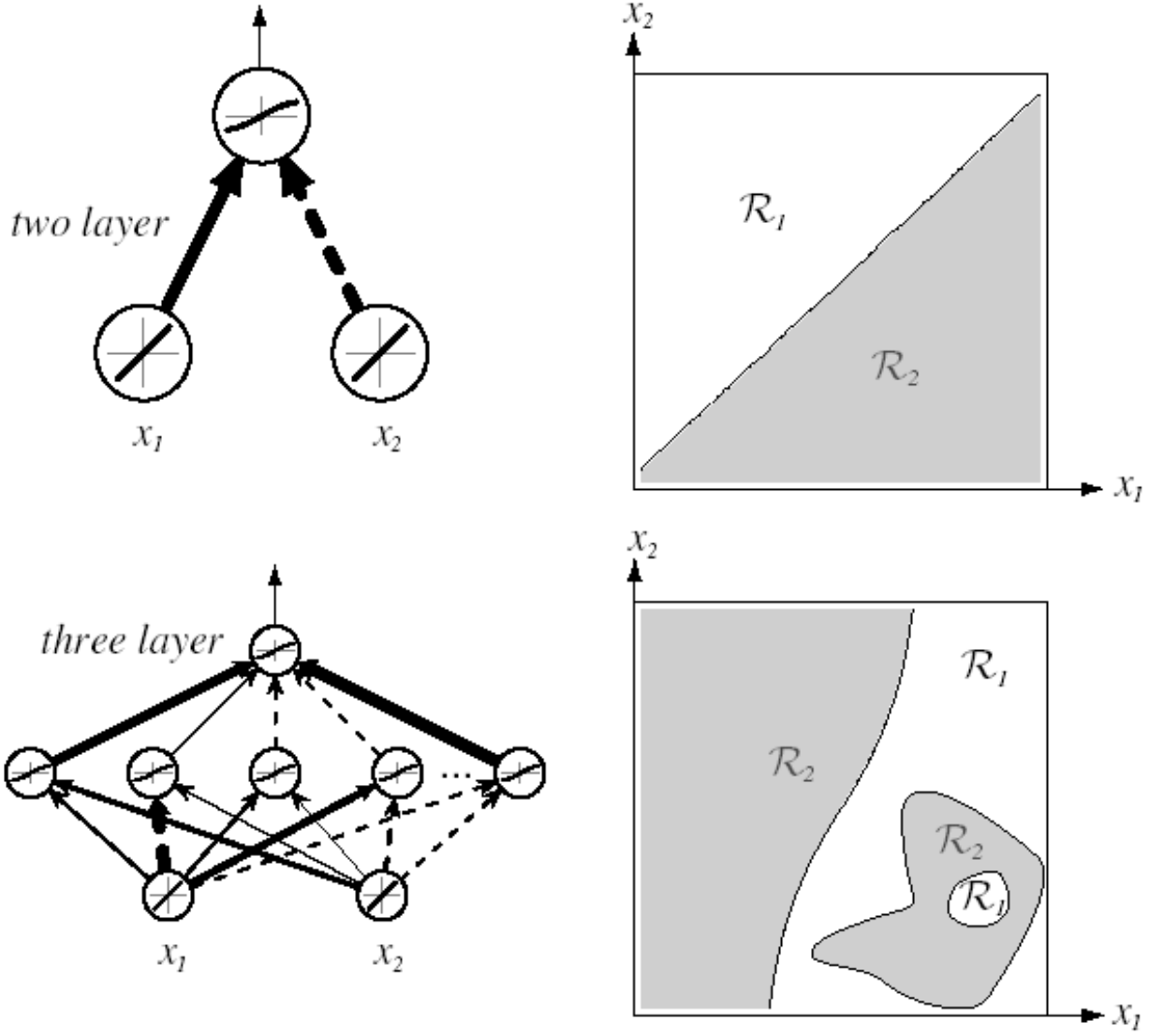
Typically this needs very nasty functions at each layer. We can also argue (using Fourier analysis) that any function can be approximated with enough hidden units.

Expressive power



Network making one “bump”. It possible to approximate any function with such bumps if we have enough hidden units (in theory, anyway).

Possible Classification Boundaries



Backpropagation

We have seen that we can *approximate* any function but how do we *learn* functions?

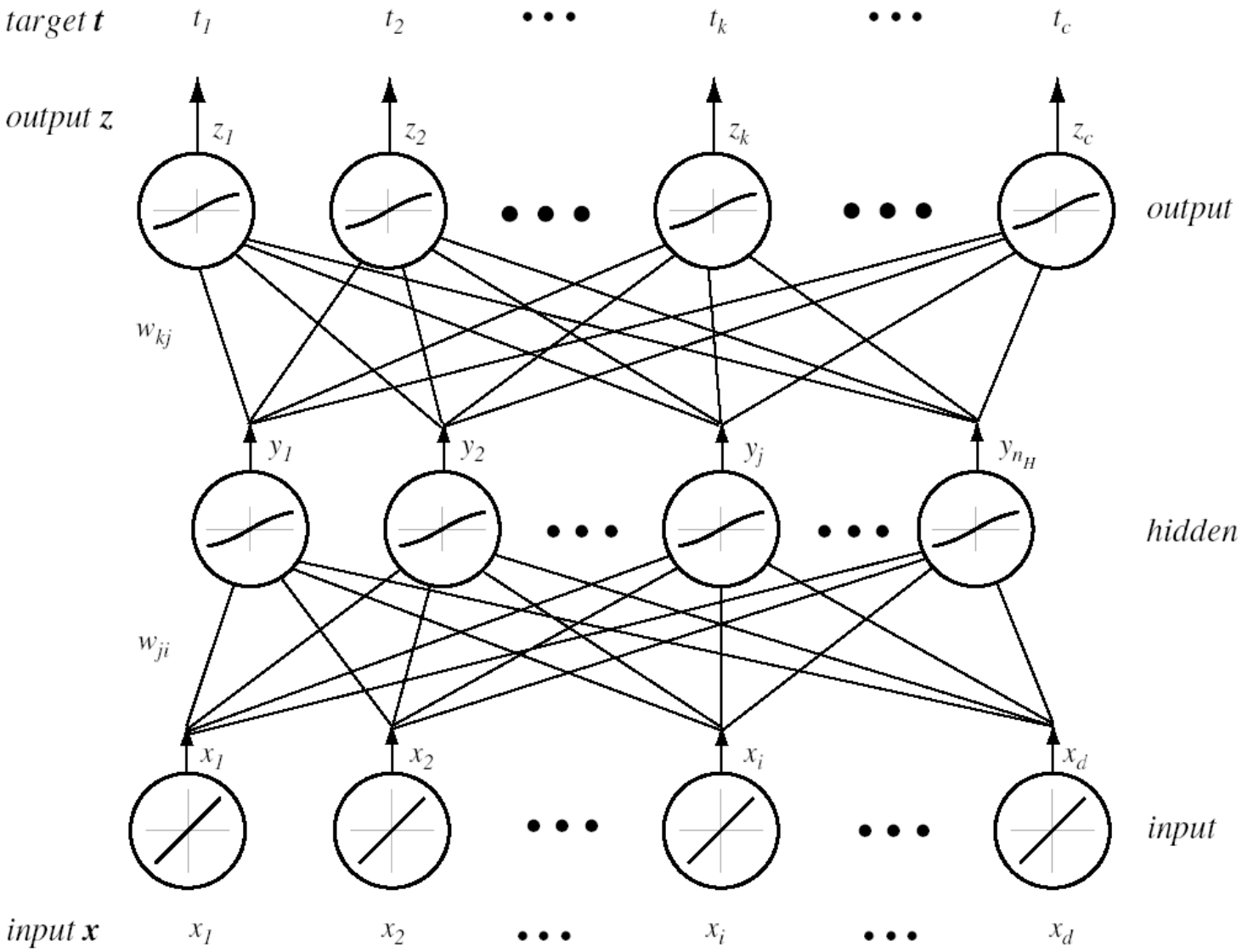
Perceptrons (single layer network): each input affects the output via its weight: so we know which weight to change to reduce errors.

Multilayer networks (a.k.a *multilayer perceptron*, *MLP*): hidden units have no “teacher” – so how reduce the error?

This is known as the *credit assignment problem*.

Backpropagation solves the credit assignment problem using a smooth activation function *f(.)* and gradient descent.

Backpropagation Network



Backpropagation Outline

Step 1: start with untrained network (random weights)

Step 2: present training data to network and calculate outputs: **z**(n) = g(**x**(n))

This generates a training error:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{n}\sum_{k=1}^{c}(t_k(n) - z_k(n))^2 = \frac{1}{2}\sum_{n}\| \mathbf{t}(n) - \mathbf{z}(n) \|^2$$

Where **w** represents the set of all weights in the network and **t**(n) are the target outputs

Step 3: change the weights in the direction of the negative gradient:

$$\Delta\mathbf{w} = -\eta\frac{\partial J}{\partial \mathbf{w}} \qquad \text{or} \qquad \Delta w_{pq} = -\eta\frac{\partial J}{\partial w_{pq}}$$

where η is the learning rate (step size).

Step 4: iterate until convergence

Backpropagation of Errors

Let us calculate the gradient for a 3-layer network. We work backwards, starting at the hidden-to-output weights.

The *chain rule* gives:

$$\frac{\partial J}{\partial \tilde{w}_{kj}} = \sum_n \frac{\partial J}{\partial n\tilde{e}t_k(n)} \frac{\partial n\tilde{e}t_k(n)}{\partial \tilde{w}_{kj}} = -\sum_n \tilde{\delta}_k(n) \frac{\partial n\tilde{e}t_k(n)}{\partial \tilde{w}_{kj}}$$

where $\tilde{\delta}_k(n) = -\partial J / \partial n\tilde{e}t_k(n)$ is termed the *sensitivity* of *J* to the net activation of *k*. Applying the *chain rule* again:

$$\tilde{\delta}_k(n) = -\frac{\partial J}{\partial n\tilde{e}t_k(n)} = -\frac{\partial J}{\partial z_k(n)} \frac{\partial z_k(n)}{\partial n\tilde{e}t_k(n)} = \underbrace{(t_k(n) - z_k(n))}_{\text{error}} f'(n\tilde{e}t_k(n))$$

where *f'(.) is the derivative of f(.).* Noting that:

$$n\tilde{e}t_k(n) = \sum_j \tilde{w}_{kj} y_j(n) \Rightarrow \frac{\partial n\tilde{e}t_k(n)}{\partial \tilde{w}_{kj}} = y_j(n)$$

Hence:

$$\Delta \tilde{w}_{kj} = -\eta \partial J / \partial \tilde{w}_{kj} = \eta \sum_n \tilde{\delta}_k(n) y_j(n) = \eta \sum_n (t_k(n) - z_k(n)) f'(n\tilde{e}t_k(n)) y_j(n)$$

Backpropagation to Hidden Units

Again using the chain rule for input-to-hidden units we have

$$\frac{\partial J}{\partial w_{ji}} = \sum_n \frac{\partial J}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial net_j(n)} \frac{\partial net_j(n)}{\partial w_{ji}(n)}$$

Note that $\partial J / \partial y_j(\mathrm{n})$ involves all outputs *k*

$$\frac{\partial J}{\partial y_j(n)} = \frac{\partial}{\partial y_j(n)} \left[\tfrac{1}{2} \sum_{k=1}^{c} (t_k(n) - z_k(n))^2 \right] = -\sum_{k=1}^{c} (t_k(n) - z_k(n)) \frac{\partial z_k(n)}{\partial y_j(n)}$$

$$= -\sum_{k=1}^{c} (t_k(n) - z_k(n)) \frac{\partial z_k(n)}{\partial n\tilde{e}t_k(n)} \frac{\partial n\tilde{e}t_k(n)}{\partial y_j(n)} = -\sum_{k=1}^{c} (t_k(n) - z_k(n)) f'(n\tilde{e}t_k(n)) \tilde{w}_{kj}$$

$$= -\sum_{k=1}^{c} \tilde{\delta}_k(n) \tilde{w}_{kj}(n)$$

since $\tilde{\delta}_k(n) = (t_k(n) - z_k(n)) f'(n\tilde{e}t_k(n))$

Hidden Units (cont)

We also have $\partial y_j(n)/\partial net_j(n) = f'(net_j(n))$ so we can define

$$\delta_j(n) \equiv -\frac{\partial J}{\partial net_j(n)} = -\frac{\partial J}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial net_j(n)} = f'(net_j(n))\sum_{k=1}^{c}\tilde{w}_{kj}\tilde{\delta}_k(n)$$

as the *hidden unit sensitivities*. Note that the output sensitivities are propagated back to the hidden unit sensitivites – hence *"back-propagation of errors"*

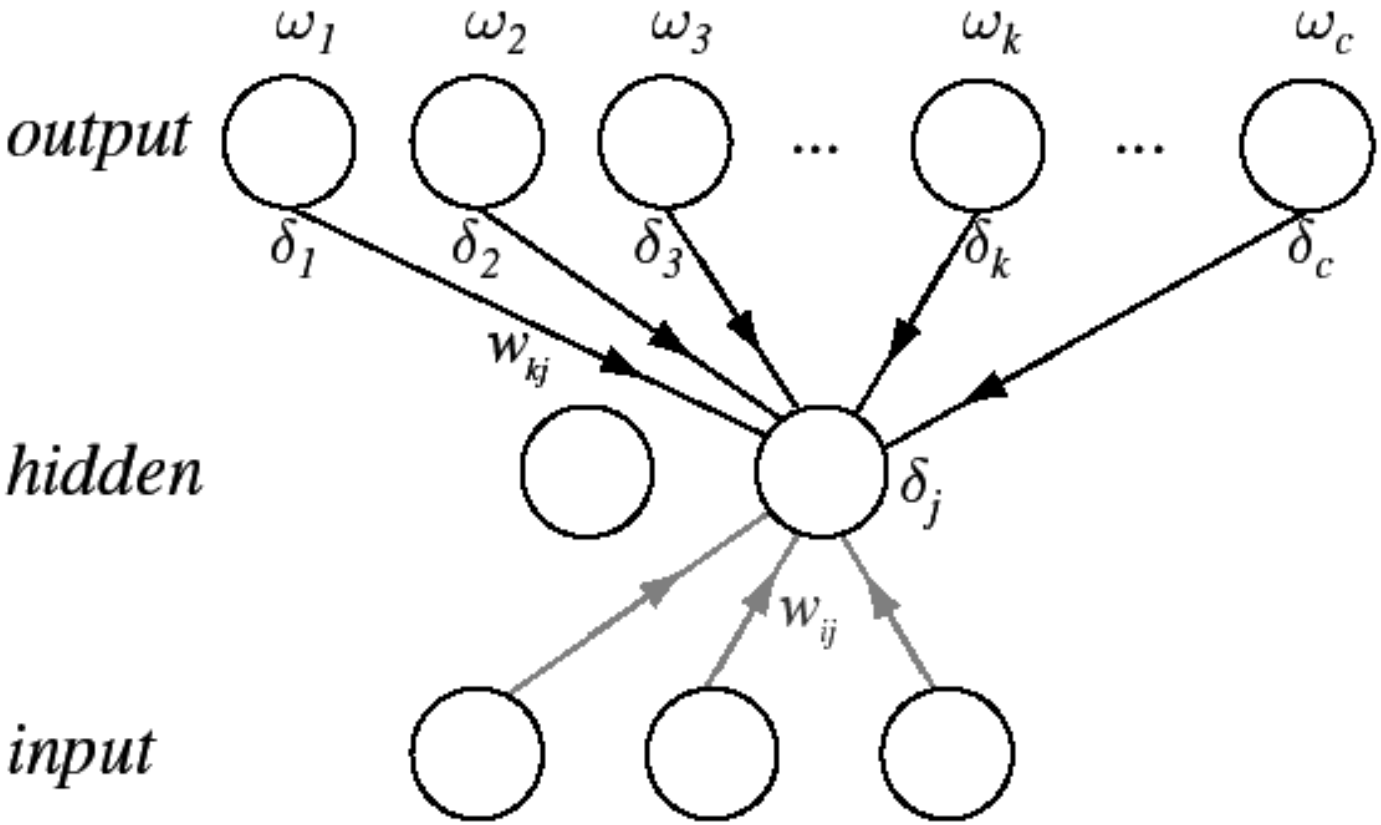
Finally we have:

$$\partial net_j(n)/\partial w_{ji} = x_i(n)$$

So the update rule for the input-to-hidden weights is:

$$\Delta w_{ji} = -\eta\frac{\partial J}{\partial w_{ji}} = \eta\sum_n x_i(n)\delta_j(n) = \eta\sum_n\left[\sum_{k=1}^{c}\tilde{w}_{kj}\tilde{\delta}_k(n)\right]f'(net_j(n))x_i(n)$$

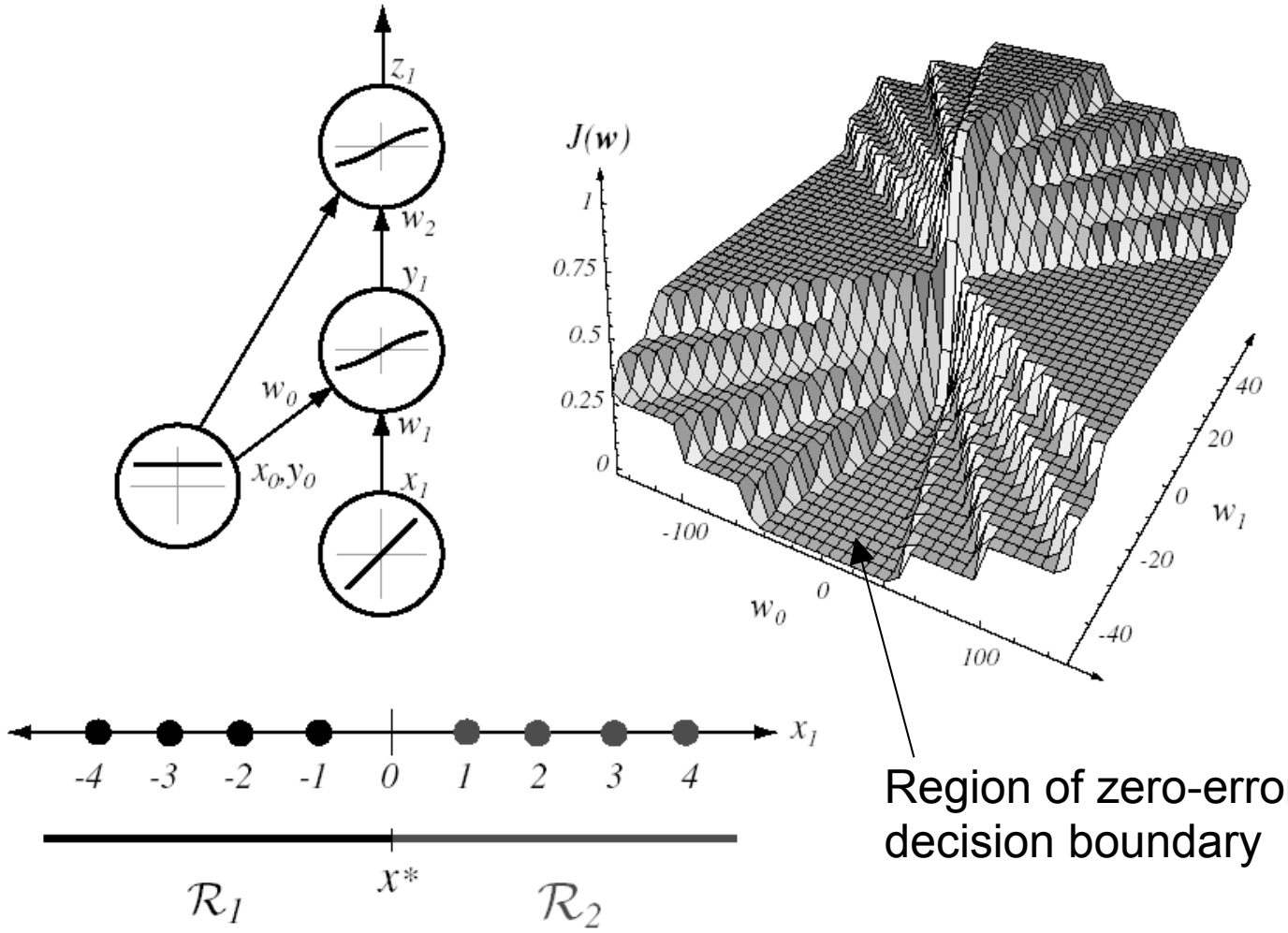
Backwards propagation



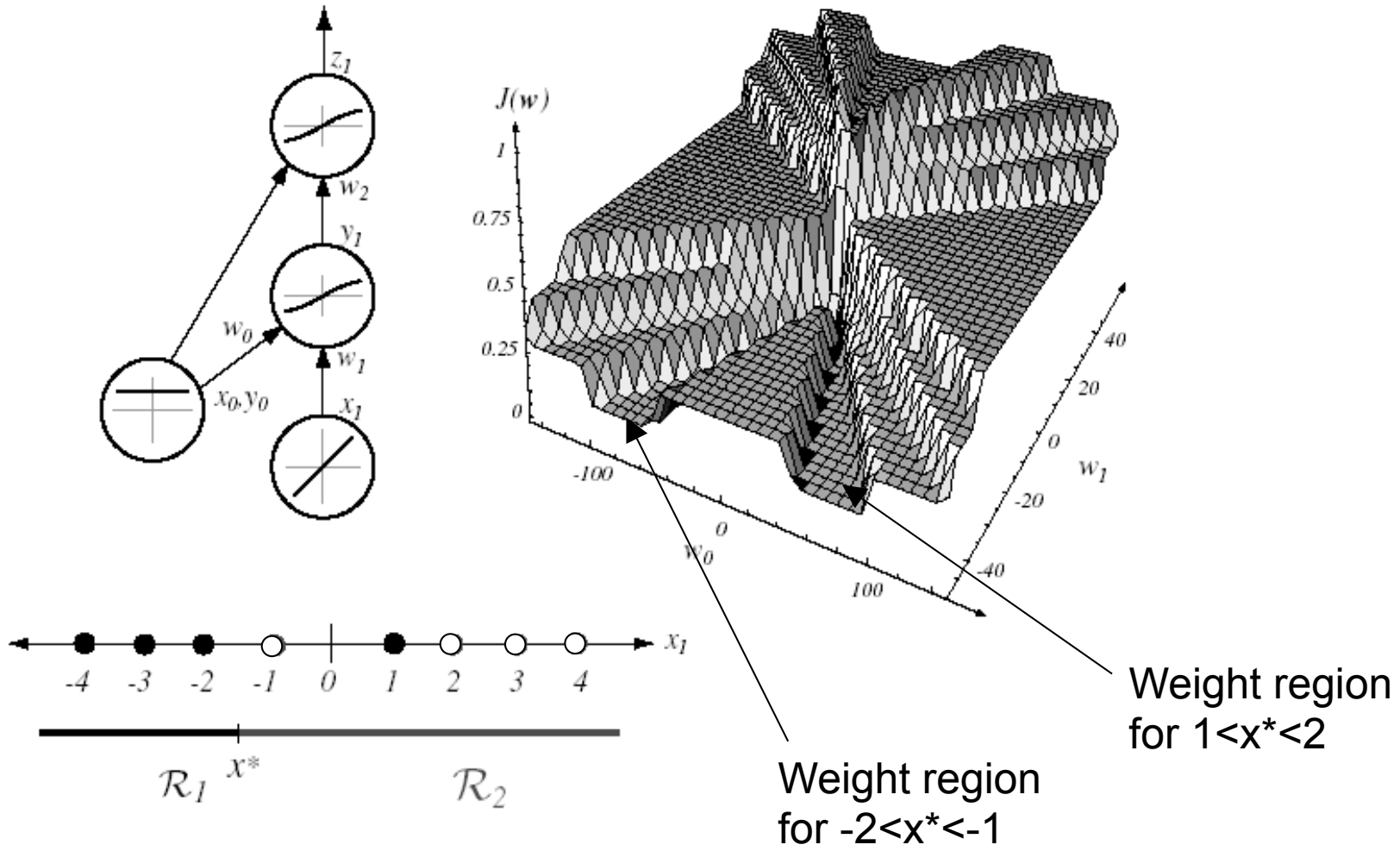
The backpropagation algorithm can be easily generalized to any network with feed-forward connections, e.g. more layers, different nonlinearities in different layers, etc.

Error Surfaces

Example for 1-1-1 network. Weight space shown is 2D: w0 and w1 (other weights fixed at final values)



Error Surfaces (2)



Both “best” regions misclassify one pattern.

Criticisms of MLPs

MLPs provide one way to achieve the required expressive power needed to build general classifiers. However they do have their weaknesses:

- Possibility of multiple (local) minima
- g(**x**) is nonlinear in terms of the weights: this makes training slow.
- *ad hoc* solution (how many units, hidden layers, etc?)

Other popular discriminant learning structures:

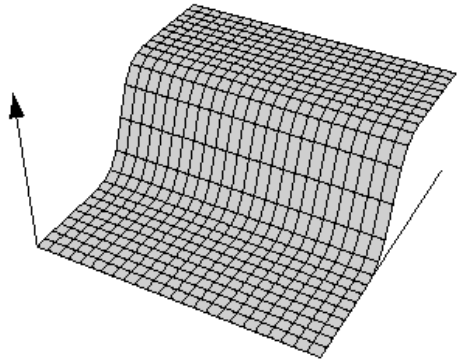
- Radial Basis Function (RBF) Networks &
- Support Vector Machines (SVMs)
- Convolutional Networks (CNNs)

Additional Networks

Radial Basis Functions (RBFs)

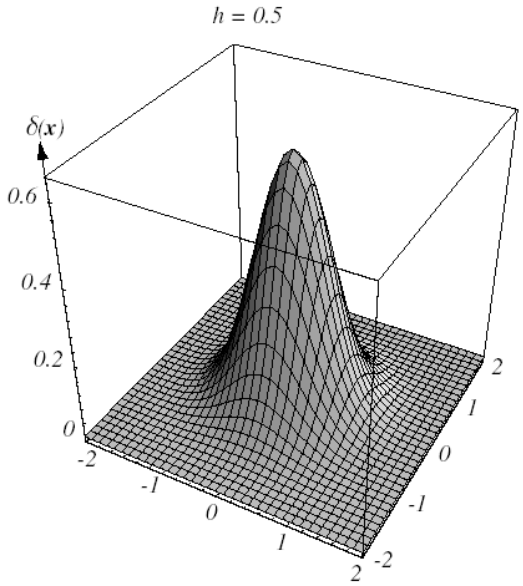
Instead of the usual sigmoid activation function, e.g.

$$y = f_i(\mathbf{x}) = tanh(\mathbf{w}_i^T \mathbf{x})$$



A Radial Basis Network (RBF) uses functions based upon radial distance, such as the Gaussian function:

$$y_i = \varphi_i(\mathbf{x}) = e^{-\frac{1}{2\sigma^2}|\mathbf{x}-\mathbf{c}_i|^2}$$



for some width σ.

Each function has a different centre $\mathbf{c}_i$

Training RBF Networks

Suppose we have a prefixed set of centres $\mathbf{c}_i$ the output then takes the form:

$$z(\mathbf{x}) = \sum_{j=1}^{n_H} w_j \varphi_j(\mathbf{x})$$

and is linear in the weights **w**. Let

$$\Phi = (\varphi_1(\mathbf{x}), \ldots, \varphi_{n_H}(\mathbf{x}))$$

then the MSE solution has a direct form:

$$\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{t}$$

i.e. RBFs are just a special case of generalized linear discriminant function. Centres often chosen as a subset of the data

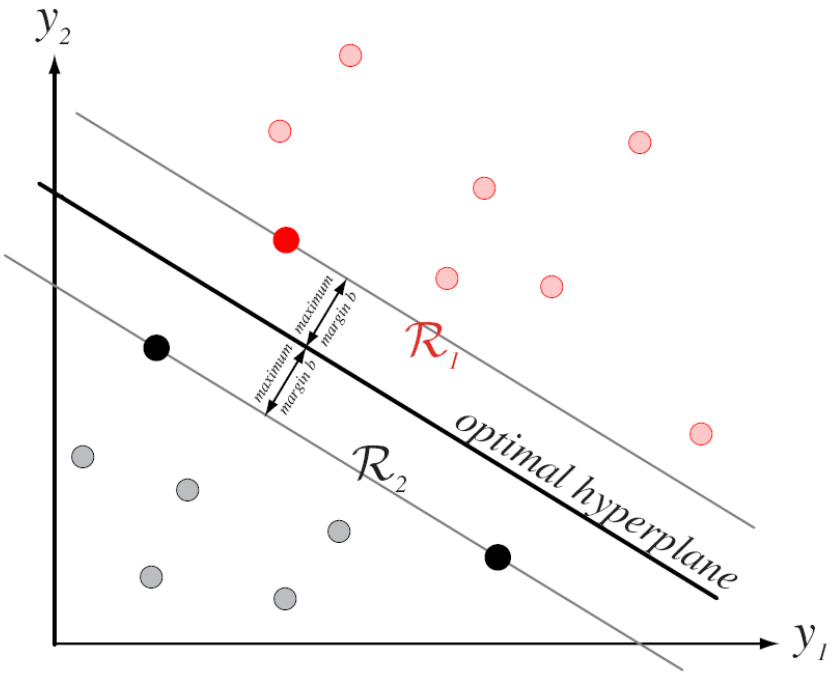
$$c_i = x_k$$

(more complicated selection criteria also exist).

Support Vector Machines

SVMs aim to solve the linearly separable problem. First map feature space into high (possibly $\infty-$) dimensional space.

Then find separating hyperplane with **maximal margin**. Recall, intuitively we are more confident in classifying point far away from the decision boundary.



Learning takes the form of a constrained optimization scheme.

SVMs: Maximizing the Margin

Suppose that we have a margin γ, such that $\omega^{(i)} a^T y_i \geq \gamma$ for all points, $i = 1, 2, \ldots, n$. We therefore want to solve the following:

$$\max \gamma, \text{such that: } \omega^{(i)} a^T y_i \geq \gamma$$
$$\text{and: } \|a\| = 1$$

This is a messy optimization problem. However, equivalently we can solve:

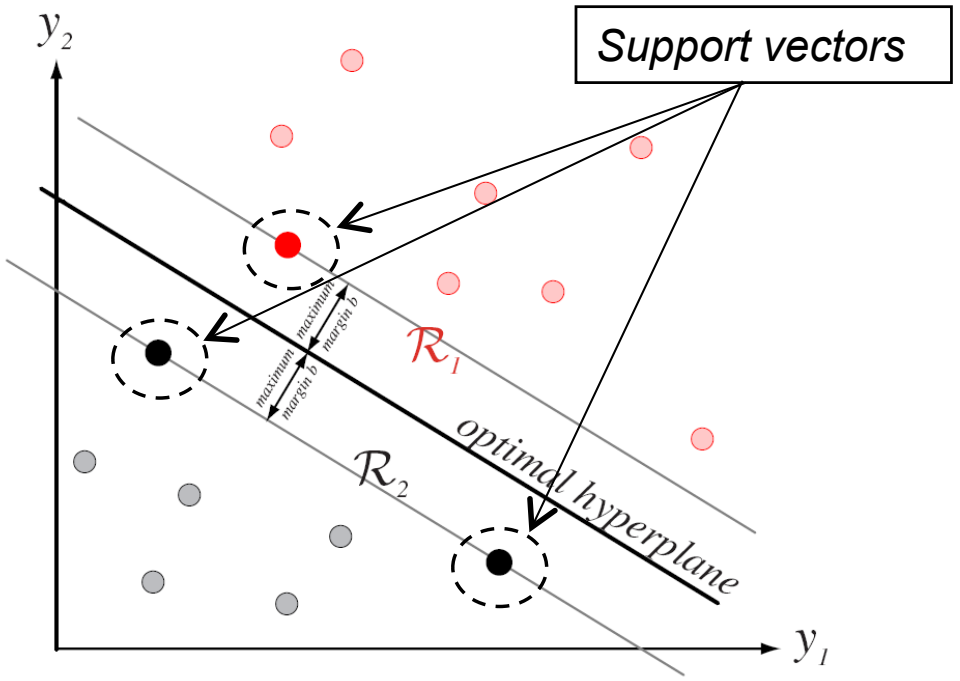
$$\min \|a\|^2, \text{such that: } \omega^{(i)} a^T y_i \geq 1$$

That is, we search for the minimum size weight vector that is able to separate the data with a margin of $\gamma = 1$.

This form of the problem is a constrained quadratic optimization problem. It is convex and (relatively) easy to solve.

The Support Vectors

Interestingly the Max margin solution only depends on a subset of the training data – those that lie exactly on the margin (why?). These are called the **support vectors** (SVs).



SVMs can be shown to generalize well in terms of cross validation error

SVM Generalization Error

- **Cross Validation (CV)** – Break the data into a *training set* and a *testing set*. Use the training data to learn the classifier then evaluate on the test data
- **Leave-one-out CV**: choose one data point at random as the testing set.

$$\{x_1, x_2, \ldots, \cancel{x_p}, x_{p+1}, \ldots x_n\}$$

- Note the LOOCV error will be unaffected unless x_p is a support vector. Therefore we have:

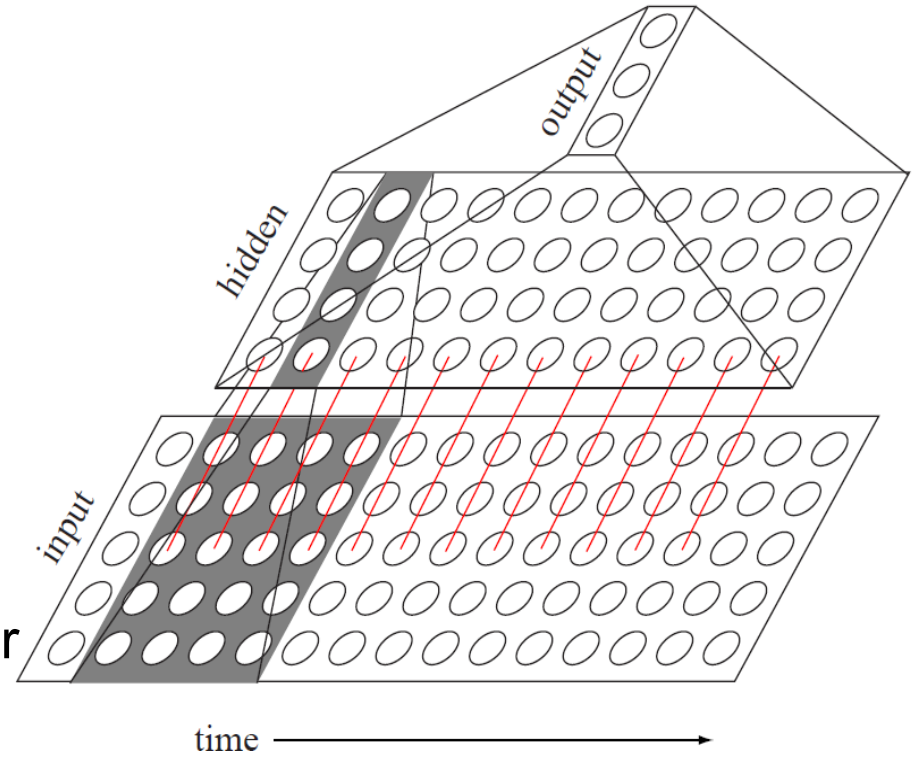
$$error_{LOOCV} \leq \frac{\#\ \text{of SVs}}{n}$$

- Therefore the number of SVs tells us how confident we are in the SVM

Convolutional Networks (CNNs)

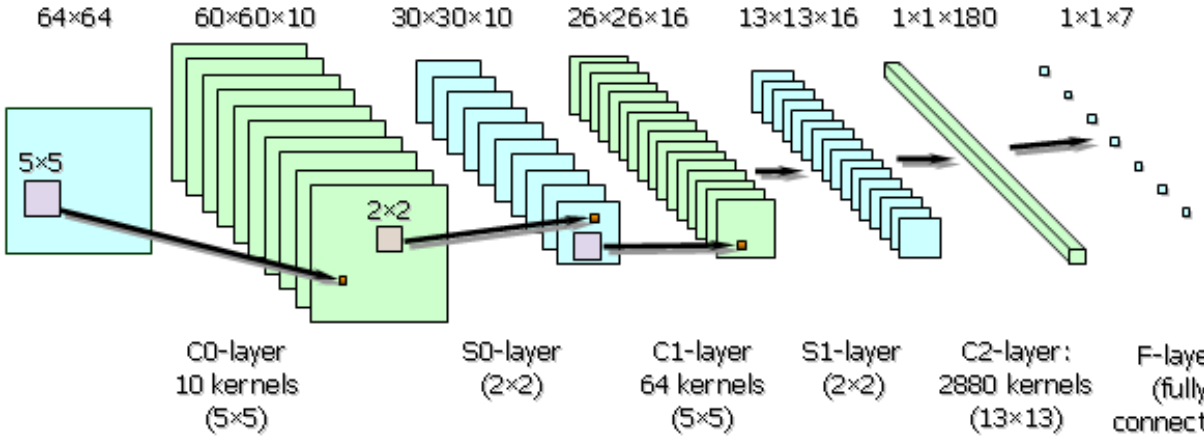
Consider the problem of building a classifier that is insensitive to translation (or scale, rotation, etc.). A Convolutional Network encodes the invariance within the MLP structure [LeCun 1998].

- Connections are restricted: hidden units are connected identically to neighbours to encode shift/delays
- Training using back prop. but with ties weights across shifted units (weight sharing)
- Output units pool results across shifted units
- The resulting MLP has much fewer weights to train than a traditional MLP



CNNs & Deep Learning

- CNNs are often considered the ancestors of Deep Learning.
- The idea is to use MLP with many (≥ 3) hidden layers. This involves lots of ‘tricks’ to make training work: convolution, subsampling, pooling of outputs,…pretraining



Application to face detection and pose estimation

but exhibit state-of-the-art performance…

e.g. see - http://www.cs.nyu.edu/~yann/research