Linear Discriminant Functions

Chapter 5 (Duda et al.)

CS479/679 Pattern Recognition
Dr. George Bebis

Generative vs Discriminant Approach

- Generative approaches estimate the **discriminant function** by first estimating the probability distribution of the data belonging to each class.

- Discriminative approaches estimate the **discriminant function** explicitly, without assuming a probability distribution.

Linear Discriminants
(case of two categories)

- A linear discriminant has the following form:

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

Decide ω_1 if $g(\mathbf{x}) > 0$ and ω_2 if $g(\mathbf{x}) < 0$

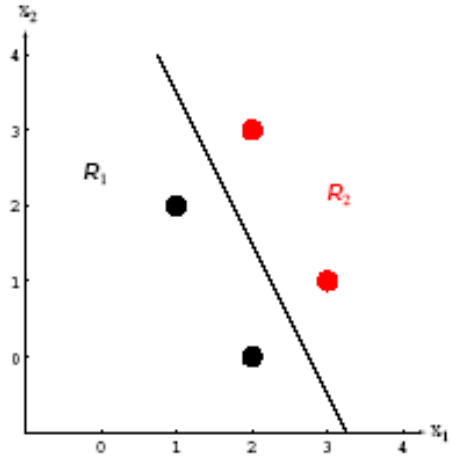
If $g(\mathbf{x})=0$, then $\mathbf{x}$ lies on the decision boundary and can be assigned to either class.

Decision Boundary

- The decision boundary g(**x**)=0 is a **hyperplane.**
- The orientation of the hyperplane is determined by **w** and its location by w_0.

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

  - **w** is the normal to the hyperplane.
  - If w_0=0, it passes through the origin



Decision Boundary Estimation

- Use "learning" algorithms to estimate **w** and w_0 from training data $\mathbf{x}_k$.
- Let us suppose that:

true class label

$$z_k = \begin{cases} +1 \; if \;\; \mathbf{x}_k \in \omega_1 \\ -1 \; if \;\; \mathbf{x}_k \in \omega_2 \end{cases}$$

predicted class label:

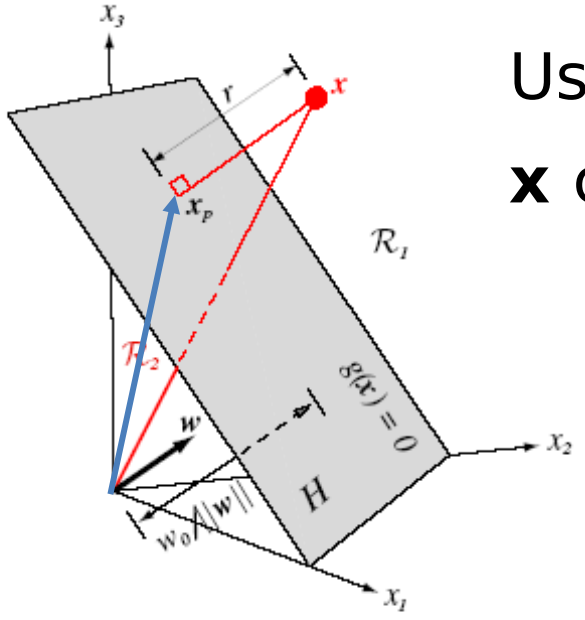
$$\hat{z}_k = \begin{cases} +1 \; if \;\; g(\mathbf{x}_k) > 0 \\ -1 \; if \;\; g(\mathbf{x}_k) < 0 \end{cases}$$

- The solution can be found by minimizing an error function, e.g., the "**training error**" or "**empirical risk**":

$$J(\mathbf{w}, w_0) = \frac{1}{n} \sum_{k=1}^{n} [z_k - \hat{z}_k]^2$$

Geometric Interpretation

- Let's look at *g(***x***)* from a geometrical point of view.



Using vector algebra,

x can be expressed as follows:

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

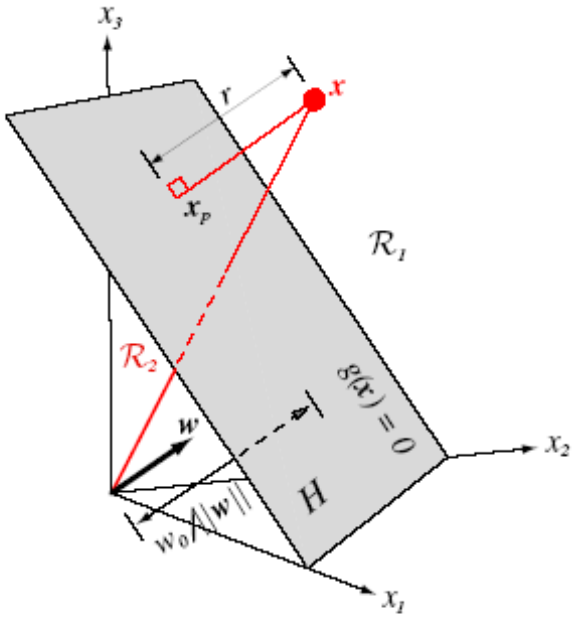
Substitute **x** in $g(\mathbf{x}) = \mathbf{w}^t\mathbf{x} + w_0$

Geometric Interpretation (cont'd)

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

$$= \mathbf{w}^t\mathbf{x}_p + r\frac{\mathbf{w}^t\mathbf{w}}{\|\mathbf{w}\|} + w_0 = r\|\mathbf{w}\|$$

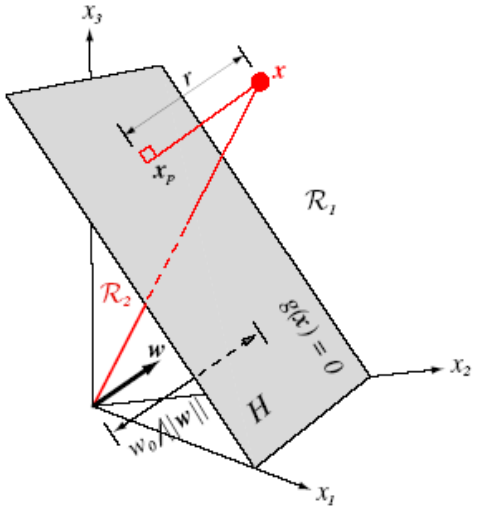
$$\mathbf{w}^t\mathbf{x}_p + w_0 = 0 \qquad \mathbf{w}^t\mathbf{w} = \|\mathbf{w}\|^2$$



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

Geometric Interpretation (cont'd)

- The distance of **x** from the hyperplane is given by:



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

distance

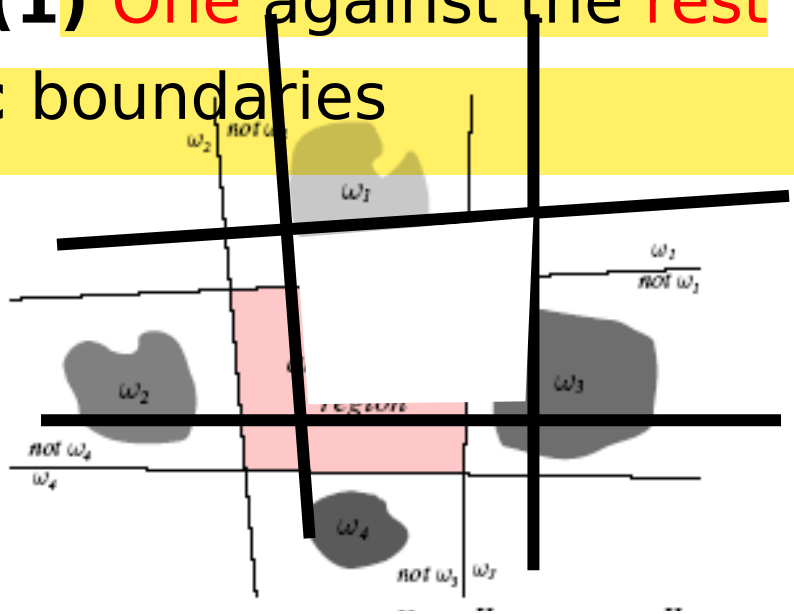
Setting x=0: $r = \frac{w_0}{\|\mathbf{w}\|}$ (i.e., distance of the plane from origin)

Linear Discriminant Functions: case of c categories

- There are several ways to devise multi-category classifiers using linear discriminant functions:

(1) One against the rest

c boundaries



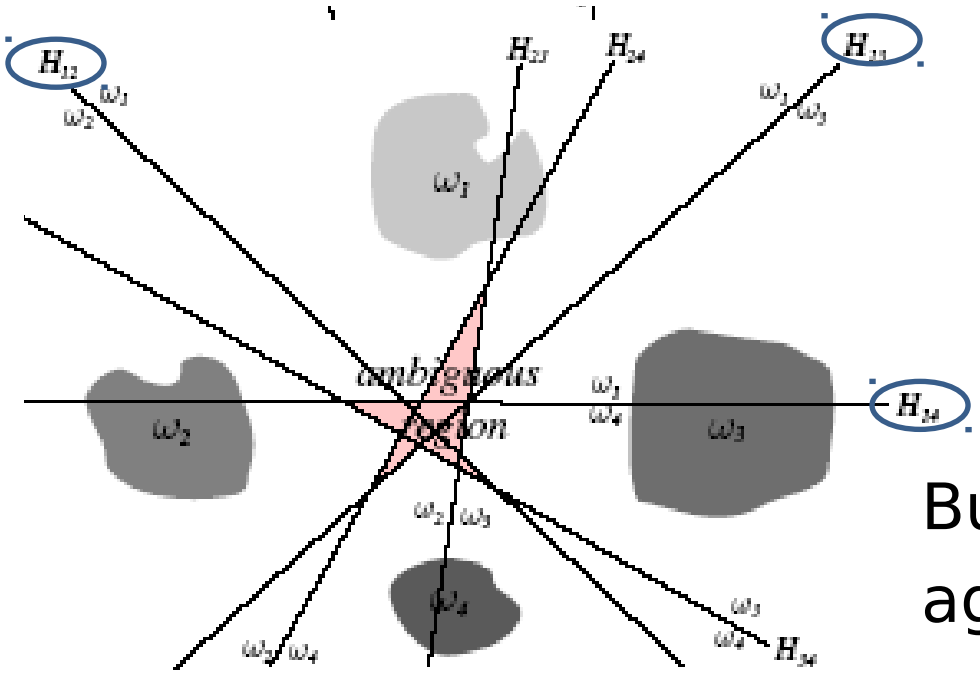
How many decision boundaries are there?

But there is a problem

ambiguous region

Linear Discriminant Functions: case of c categories (cont'd)

(2) One against another **c(c-1)/2** boundaries

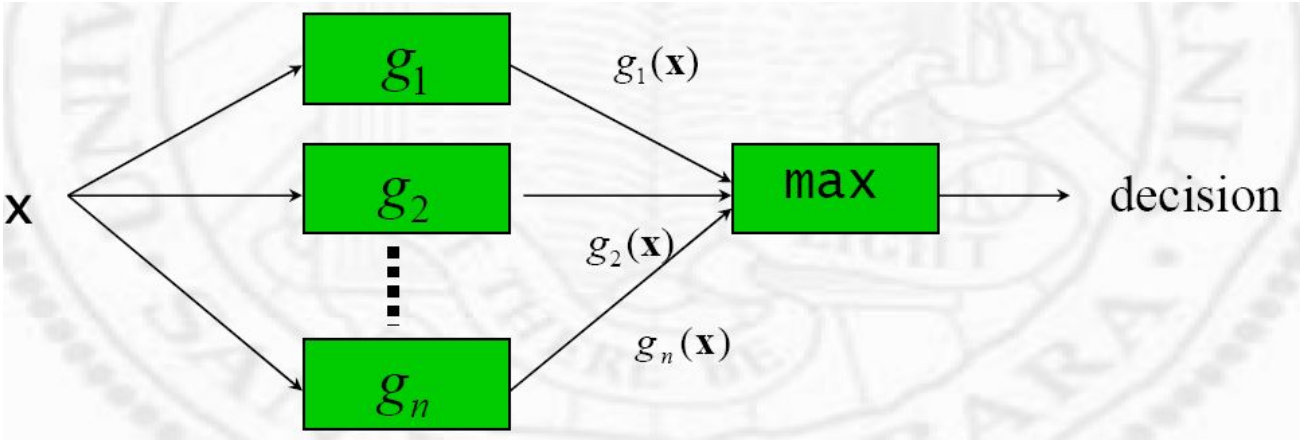


How many decision boundaries are there?

But there is a problem again: ambiguous regior

Linear Discriminant Functions: case of c categories (cont'd)

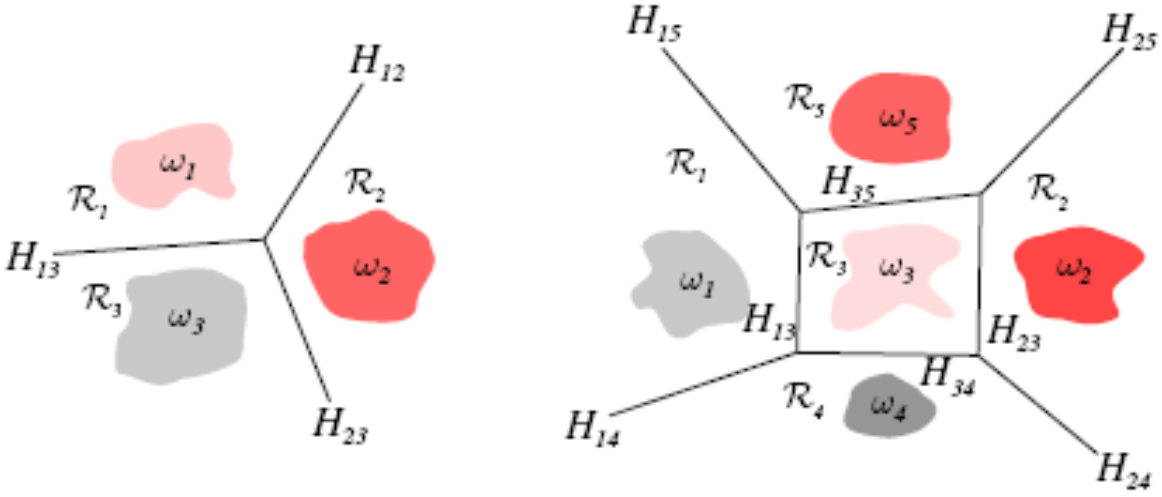
- To avoid the problem with ambiguous regions:
  - Define c linear functions $g_i(\mathbf{x})$, $i=1,2,\ldots,c$
  - Assign $\mathbf{x}$ to ω_i if $g_i(\mathbf{x}) > g_j(\mathbf{x})$ for all $j \neq i$.



- The resulting classifier is called a linear machine.

Linear Discriminant Functions: case of c categories (cont'd)

- A linear machine divides the feature space in **c** convex decisions regions.



If **x** is in region R_i, the $g_i(\mathbf{x})$ is the largest.

Note: although there are **c(c-1)/2** region pairs, there typically less decision boundaries (i.e., 8 instead of 10 in the five class example above).

Geometric Interpretation

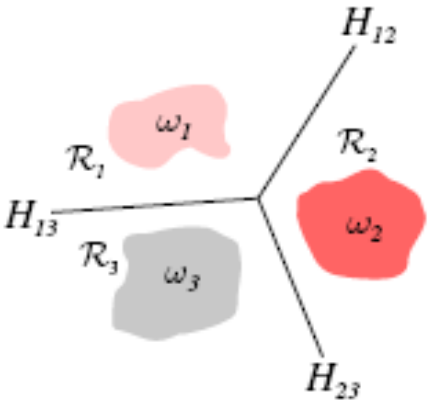
- The decision boundary between adjacent regions R_i and R_j is a portion of the hyperplane H_{ij} :

$$g_i(\mathbf{x}) = g_j(\mathbf{x}) \quad or \quad g_i(\mathbf{x}) - g_j(\mathbf{x}) = 0$$

$$g_i(\mathbf{x}) = \mathbf{w}_i^t \mathbf{x} + w_{i0}$$

$$g_j(\mathbf{x}) = \mathbf{w}_j^t \mathbf{x} + w_{j0}$$

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



  - $(\mathbf{w}_i - \mathbf{w}_j)$ is normal to H_{ij}

  - The distance from $\mathbf{x}$ to H_{ij} is:

$$r = \frac{g_i(\mathbf{x}) - g_j(\mathbf{x})}{\| \mathbf{w}_i - \mathbf{w}_j \|}$$

Higher Order Discriminant Functions

- Higher order discriminants yield more **complex** decision boundaries than linear discriminant functions.
- Quadratic discriminant - add terms corresponding to products of pairs of components of **x**:

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

- Polynomial discriminant – add even higher order products such as:

$$x_i x_j x_k w_{ijk}$$

Linear Discriminants Revisited – A More General Definition

More convenient when the decision boundary passes through the origin – augment feature space!

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

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{bmatrix} \Rightarrow \mathbf{y} = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_d \end{bmatrix}$$

d features — d+1 features

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_d \end{bmatrix} \Rightarrow \boldsymbol{\alpha} = \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_d \end{bmatrix}$$

d parameters — d+1 parameters

Linear Discriminants Revisited – A More General Definition (cont'd)

Discriminant: $g(\mathbf{y}) = \boldsymbol{\alpha}^t \mathbf{y}$

Classification rule:

Decide ω_1 if $\boldsymbol{\alpha}^t\mathbf{y} > 0$ and ω_2 if $\boldsymbol{\alpha}^t\mathbf{y} < 0$

- Separates points in (d+1)-space by a hyperplane.
- Decision boundary passes through the origin.

Generalized Discriminants

- The main idea is **mapping** the data to a space of higher dimensionality.

$$d \rightarrow \hat{d} \text{ where } \hat{d} >> d$$

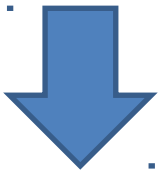
- This can be done by transforming the data through properly chosen functions $y_i(\mathbf{x})$, i=1,2, …, $\hat{d}$ (called φ functions):

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{bmatrix} \overset{\varphi}{\Rightarrow} \begin{bmatrix} y_1(\mathbf{x}) \\ y_2(\mathbf{x}) \\ \dots \\ y_{\hat{d}}(\mathbf{x}) \end{bmatrix}$$

Generalized Discriminants (cont'd)

- A **generalized discriminant** is defined as a linear discriminant in the d-dimensional space:

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



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{bmatrix} \Rightarrow^{\varphi} \begin{bmatrix} y_1(\mathbf{x}) \\ y_2(\mathbf{x}) \\ \dots \\ y_{\hat{d}}(\mathbf{x}) \end{bmatrix}$$

$$g(\mathbf{x}) = \sum_{i=1}^{\hat{d}} a_i y_i(\mathbf{x}) \quad or \quad g(\mathbf{x}) = \boldsymbol{\alpha}^t \mathbf{y}$$

Generalized Discriminants (cont'd)

- Why are generalized discriminants attractive?

properly φ not become

$\hat{d}$ -

Example

$$g(x) > 0 \;\; if \;\; x < -1 \; or \; x > 0.5$$

- The corresponding decision regions R_1, R_2 in the 1D-space are **not** simply connected (i.e., not linearly separable).

- Consider the following mapping and generalized discriminant:

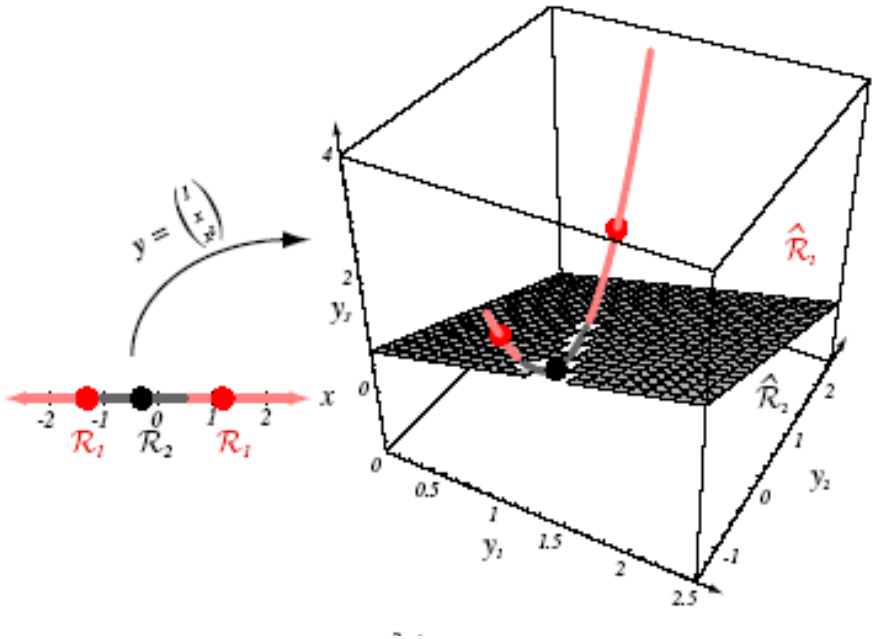
$$\mathbf{y} = \begin{bmatrix} y_1(x) \\ y_2(x) \\ y_3(x) \end{bmatrix} = \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix} \qquad g(x) = \boldsymbol{\alpha}^t \mathbf{y} \qquad \boldsymbol{\alpha} = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}$$

$$d=1 \rightarrow \hat{d} = 3$$

$$g(x) = -1 + x + 2x^2$$

Example (cont'd)

g(**x**) maps a line in d-space to a parabola in $\hat{d}$ - space.



The problem has now become linearly separable

$$\boldsymbol{\alpha}^t \mathbf{y} = 0$$

The plane, $\hat{H}$, divides the $\hat{d}$-space in two decision regions $\hat{\mathcal{R}}_1, \hat{\mathcal{R}}_2$

Learning Linearly Separable Categories

- Given a linear discriminant function

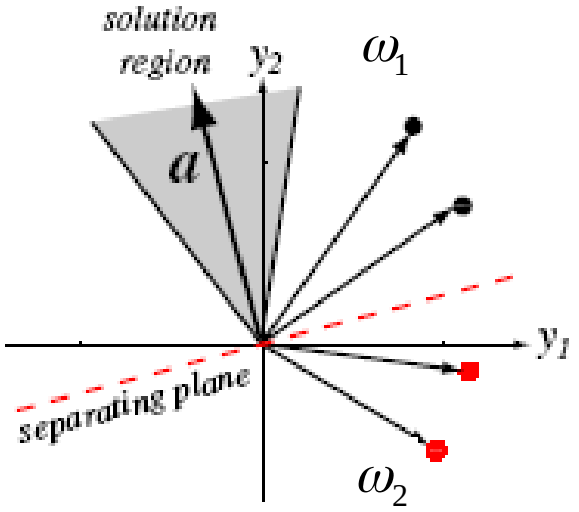
$$g(\mathbf{x}) = \boldsymbol{\alpha}^t \mathbf{y}$$

the goal is to **"learn"** the parameters (weights) **α** from a set of ***n*** labeled samples $\mathbf{y_i}$, where each $\mathbf{y_i}$ has a class label ω_1 or ω_2.

Learning: effect of training examples

- Every training sample $\mathbf{y}_i$ places a constraint on the weight vector $\boldsymbol{\alpha}$

- Visualize solution in "feature space":
  - $\boldsymbol{\alpha}^t\mathbf{y}=0$ defines a hyperplane in the feature space with $\boldsymbol{\alpha}$ being the normal vector.
  - Given *n* examples, the solution $\boldsymbol{\alpha}$ must lie within a certain region (shaded region in the example).

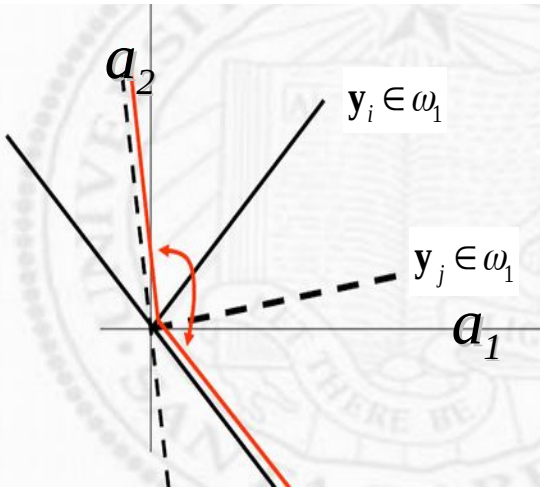


Learning: effect of training examples (cont'd)

- Visualize solution in "parameter space":
  - $\boldsymbol{\alpha}^t\mathbf{y}=0$ defines a hyperplane in the parameter space with **y** being the normal vector.
  - Given *n* examples, the solution **α** must lie on the intersection of *n* half-spaces

    (shown by the red lines in the example).

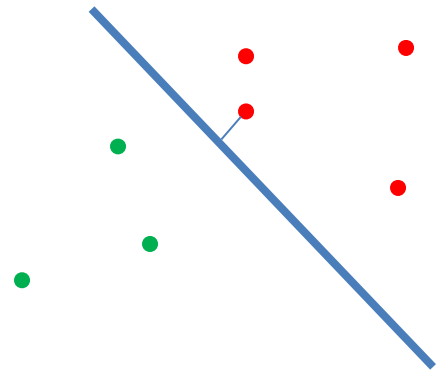
parameter space (α_1, α_2)



Uniqueness of Solution

olution vector α is usually not unique; we can impos
dditional constraints to enforce uniqueness, e.g.,:

"Find unit-length weight vector α that maximizes the minimum distance from the training examples to the separating plane"

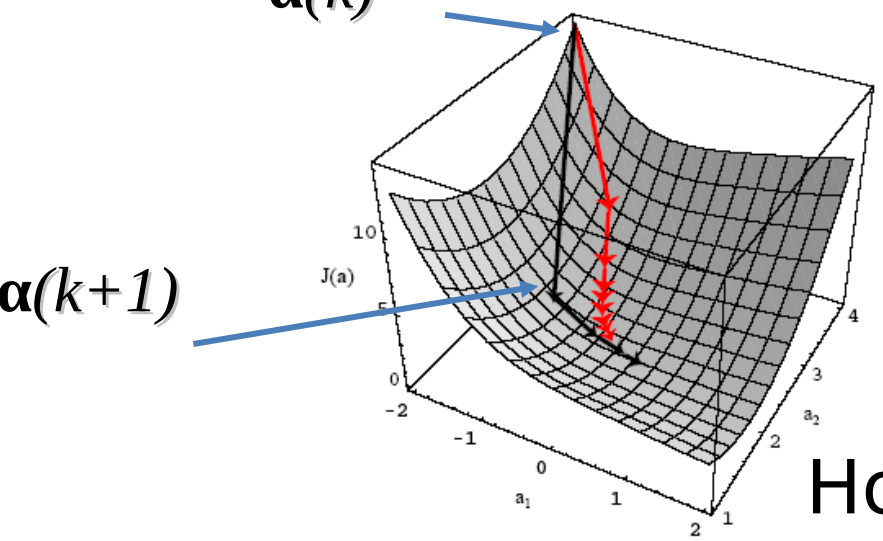


"Learning" Using Iterative Optimization

- Minimize some error function $J(\boldsymbol{\alpha})$ with respect to $\boldsymbol{\alpha}$.
  e.g., $J(\boldsymbol{\alpha}) = \frac{1}{n}\sum_{k=1}^{n}[z_k - \hat{z}_k]^2$

- Minimize $J(\boldsymbol{\alpha})$ iteratively:

$$\boldsymbol{\alpha}(k+1) = \boldsymbol{\alpha}(k) + \eta(k)\mathbf{p}_k$$



$\mathbf{p}_k$ search direction

$\eta(k)$ learning rate (search step)

How should we choose $\mathbf{p}_k$?

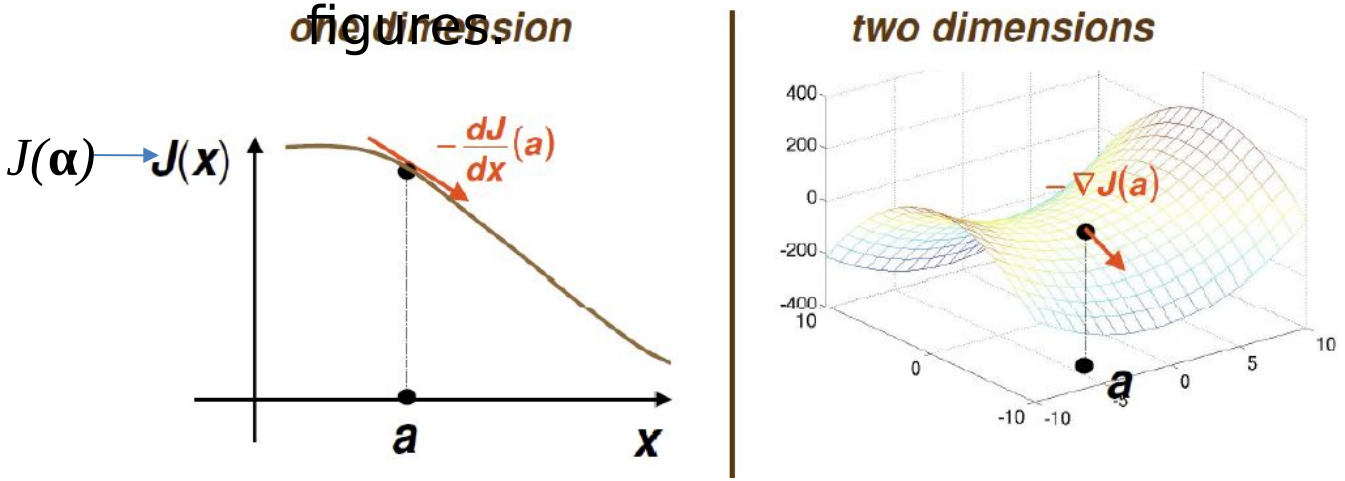
Choose p_k using Gradie

$$\begin{bmatrix} \frac{\partial}{\partial x_1} J(x) \\ \vdots \\ \frac{\partial}{\partial x_d} J(x) \end{bmatrix} = \nabla J(x)$$

Warning: notation is reversed in the figures.

- ∇J

points in the direction of steepest decrease!



$$\boldsymbol{\alpha}(k+1) = \boldsymbol{\alpha}(k) + \eta(k)\mathbf{p}_k$$

$$\mathbf{p}_k = -\nabla J(\boldsymbol{\alpha}(k))$$

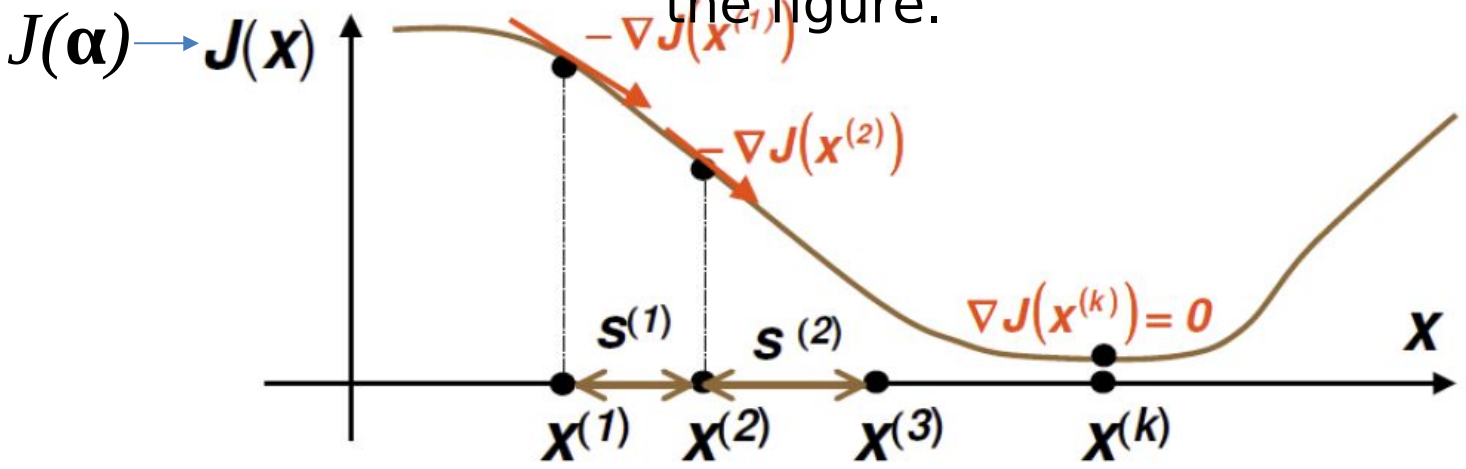
$$\boldsymbol{\alpha}(k+1) = \boldsymbol{\alpha}(k) - \eta(k)\nabla J(\boldsymbol{\alpha}(k))$$

Gradient Descent

Algorithm 1 (Basic gradient descent)

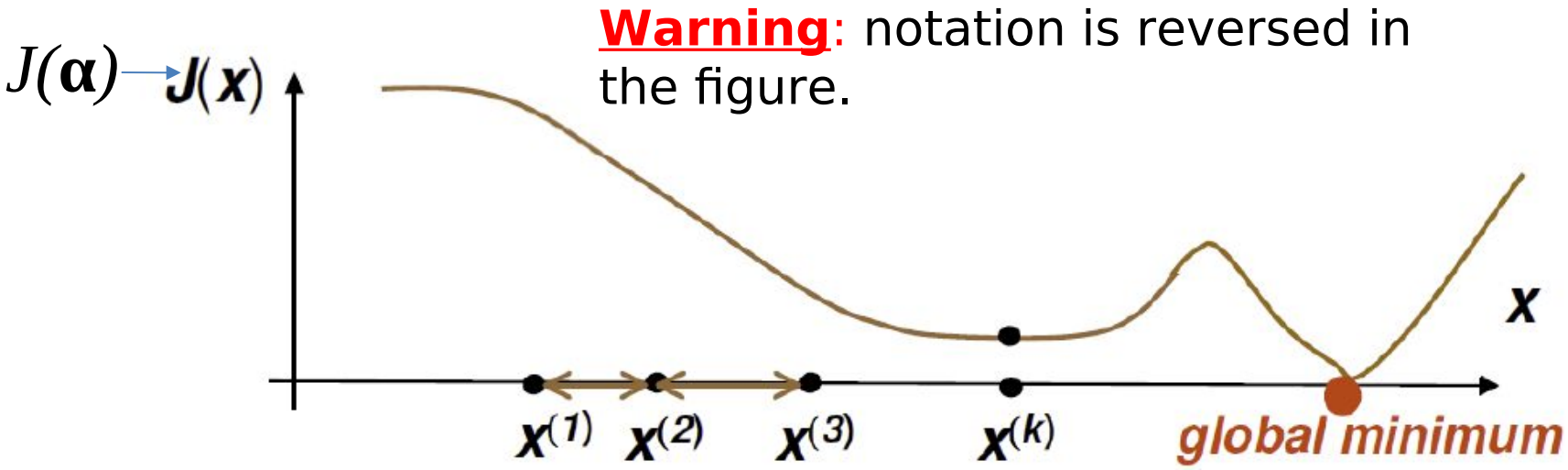
1 **begin initialize** $\mathbf{a}$, criterion $\theta, \eta(\cdot), k = 0$
2 **do** $k \leftarrow k + 1$
3 $\mathbf{a} \leftarrow \mathbf{a} - \eta(k)\nabla J(\mathbf{a})$
4 **until** $\eta(k)\nabla J(\mathbf{a}) < \theta$
5 **return** a
6 **end**

Warning: notation is reversed in the figure.



Gradient Descent (cont'd)

- Gradient descent is very popular due to its simplicity but can get stuck in local minima.

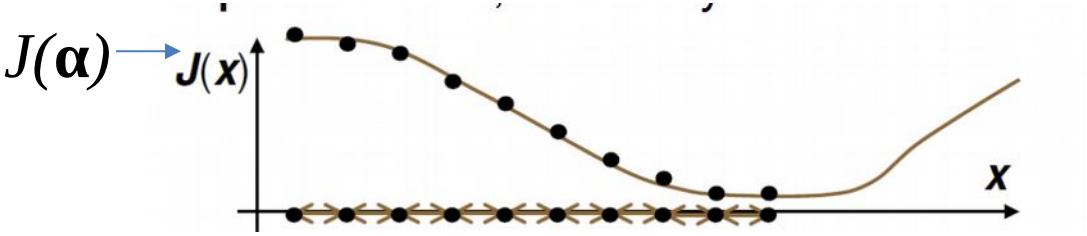


Gradient Descent

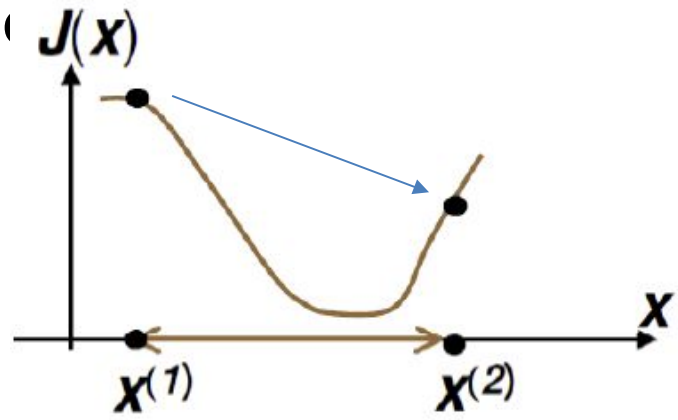
What is the effect of the learning rate $\eta(k)$?

- If it is too small, it takes too many iterations.
- If it is too big, it might **overshoot** the solution (and never find it), possibly leading to oscillations (**no convergenc**

Warning: notation is reversed in the figure.



Take bigger steps to converge faster.

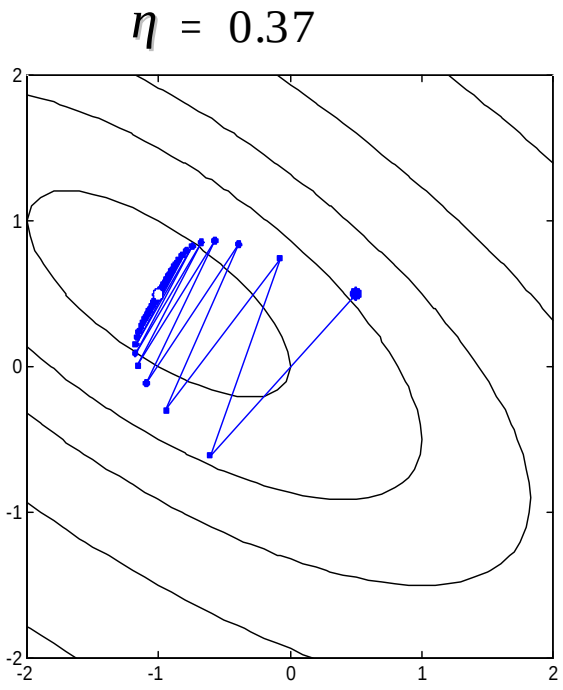


Take smaller steps to avoid overshooting.

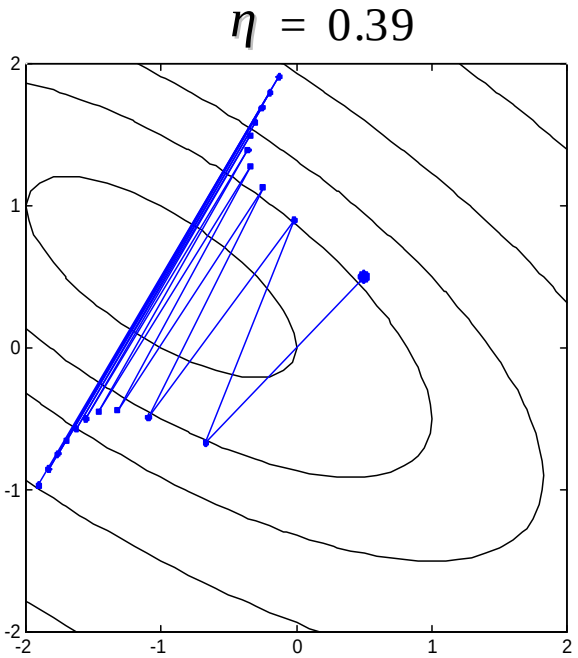
Gradient Descent (cont'd)

- Even a small change in the learning rate might lead to overshooting the solution.

$J(\boldsymbol{\alpha})$



Converges to the solution!



Overshoots the solution!

Gradient Descent (cont'd)

- Could we choose $\eta(k)$ adaptively?
  - Yes; let's review Taylor Series expansion first.

Expands f(x) around x_0 using derivatives:

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 \\ &\quad + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \frac{f''''(x_0)}{4!}(x - x_0)^4 + \cdots \\ &= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \end{aligned}$$

Gradient Descent (cont'd)

- Expand J(**α**) around **α**$_0$=**α**(k) using Taylor Series (up to second derivative):

$$J(\boldsymbol{\alpha}) \approx J(\boldsymbol{\alpha}(k)) + \nabla J^{t}(\boldsymbol{\alpha} - \boldsymbol{\alpha}(k)) + \frac{1}{2}(\boldsymbol{\alpha} - \boldsymbol{\alpha}(k))^{t}\,\mathbf{H}\,(\boldsymbol{\alpha} - \boldsymbol{\alpha}(k))$$

$$\nabla J \equiv \nabla J(\boldsymbol{\alpha}(k))$$

Hessian matrix

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

- Evaluate J(**α**) at **α**=**α**(k+1)

$$\boldsymbol{\alpha}(k+1) = \boldsymbol{\alpha}(k) - \eta(k)\nabla J(\boldsymbol{\alpha}(k))$$

$$J(\boldsymbol{\alpha}(k+1)) \approx J(\boldsymbol{\alpha}(k)) - \eta(k)\|\nabla J\|^2 + \frac{1}{2}\eta^2(k)\nabla J^{t}\mathbf{H}\nabla J$$

$$\eta(k) \approx \frac{\|\nabla J\|^2}{\nabla J^{t}\mathbf{H}\nabla J}$$

Expensive to compute in practice!

Choosing $\mathbf{p}_k$ using Hessian

$$\boldsymbol{\alpha}(k+1) = \boldsymbol{\alpha}(k) + \eta(k)\mathbf{p}_k$$

$$\mathbf{p}_k = -H^{-1}\nabla J(\boldsymbol{\alpha}(k))$$

$$\boldsymbol{\alpha}(k+1) = \boldsymbol{\alpha}(k) - \eta(k)\mathbf{H}^{-1}\nabla J$$

requires inverting H;
expensive in practice!

Gradient descent can be seen as a special case of Newton's method assuming **H=I**

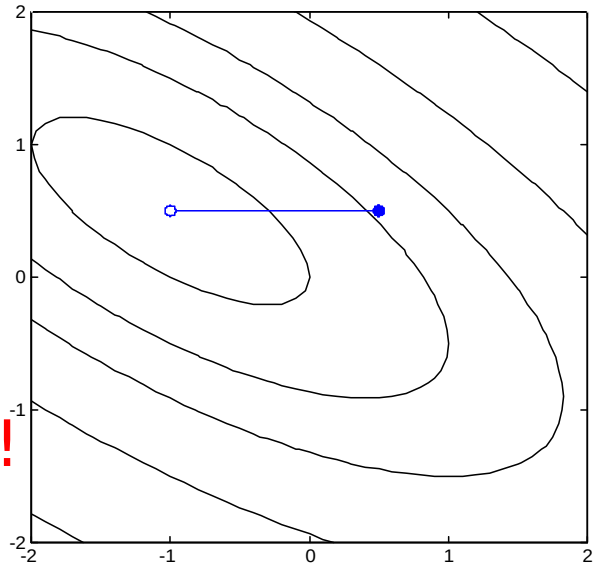
Newton's Method

Algorithm 2 (Newton descent)

1 **begin** **initialize** a, criterion θ
2 **do**
3 $\mathbf{a} \leftarrow \mathbf{a} - \mathbf{H}^{-1}\nabla J(\mathbf{a})$
4 **until** $\mathbf{H}^{-1}\nabla J(\mathbf{a}) < \theta$
5 **return** a
6 **end**

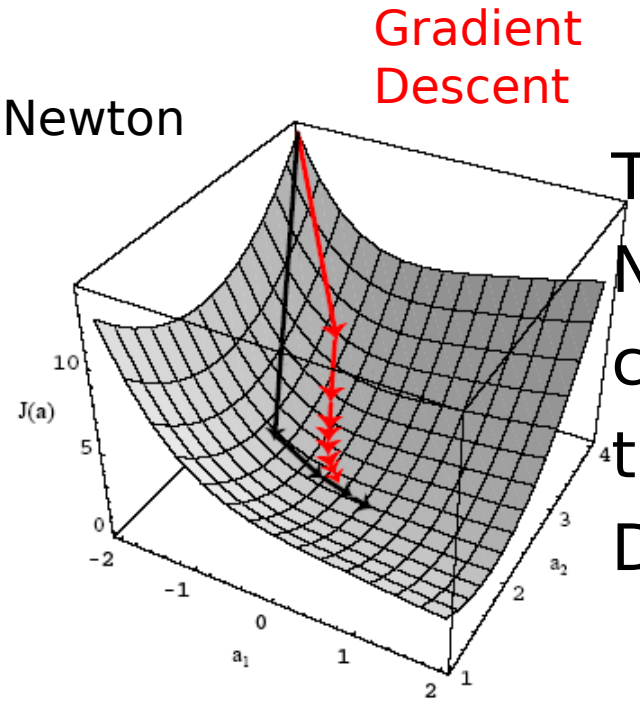
$\eta(k)=1$

If $J(\boldsymbol{\alpha})$ is **quadratic**, Newton's method converges in **one iteration**!



$J(\boldsymbol{\alpha})$

Gradient descent vs Newton's method



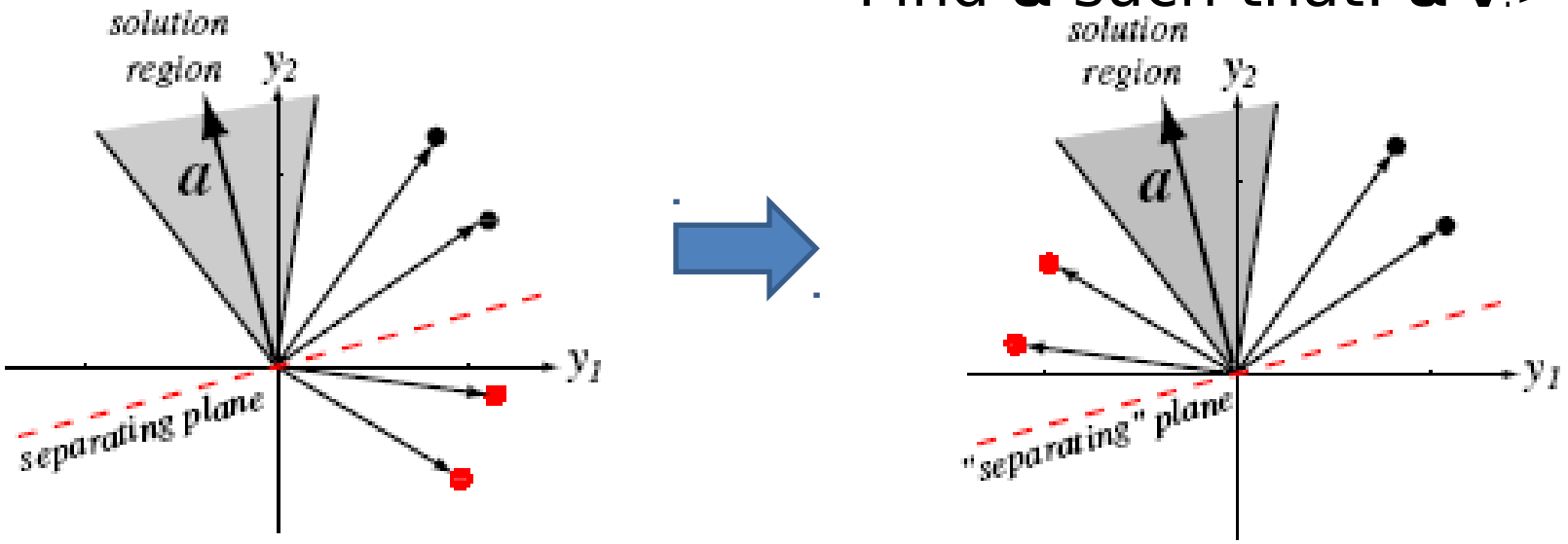
Typically, Newton's method converges faster than Gradient Descent.

Figure 5.10: The sequence of weight vectors given by a simple gradient descent method (red) and by Newton's (second order) algorithm (black). Newton's method typically leads to greater improvement per step, even when using optimal learning rates for both methods. However the added computational burden of inverting the Hessian matrix used in Newton's method is not always justified, and simple descent may suffice.

“Dual” Classification Problem

If $\boldsymbol{\alpha}^t\mathbf{y}_i>0$ assign $\mathbf{y}_i$ to ω_1
else if $\boldsymbol{\alpha}^t\mathbf{y}_i<0$ assign $\mathbf{y}_i$ to ω_2

- If $\mathbf{y}_i$ in ω_2, replace $\mathbf{y}_i$ by $-\mathbf{y}_i$
- Find $\boldsymbol{\alpha}$ such that: $\boldsymbol{\alpha}^t\mathbf{v}_i>0$



Seeks a hyperplane that separates patterns from different categories

Seeks a hyperplane that puts normalized patterns on the same (positive) side

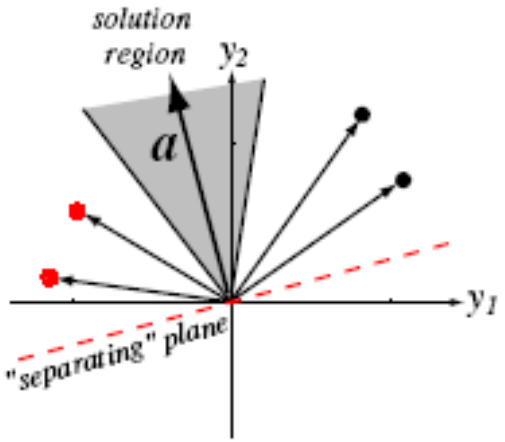
Perceptron rule

- The perceptron rule minimizes the following error function:

$$J_p(\boldsymbol{\alpha}) = \sum_{\mathbf{y} \in Y(\boldsymbol{\alpha})} (-\boldsymbol{\alpha}^t \mathbf{y})$$

where $Y(\boldsymbol{\alpha})$ is the set of samples misclassified by $\boldsymbol{\alpha}$.

- If $Y(\boldsymbol{\alpha})$ is empty, $J_p(\boldsymbol{\alpha})=0$; otherwise, $J_p(\boldsymbol{\alpha})>0$



Find $\boldsymbol{\alpha}$ such that: $\boldsymbol{\alpha}^t \mathbf{y}_i > 0$ for all i

Perceptron rule (cont'd)

- Apply gradient descent using $J_p(\boldsymbol{\alpha})$:

$$\boldsymbol{\alpha}(k+1) = \boldsymbol{\alpha}(k) - \eta(k)\nabla J(\boldsymbol{\alpha}(k))$$

- Compute the gradient of $J_p(\boldsymbol{\alpha})$

$$J_p(\boldsymbol{\alpha}) = \sum_{\mathbf{y}\in Y(\boldsymbol{\alpha})} (-\boldsymbol{\alpha}^t\mathbf{y}) \qquad \nabla J_p = \sum_{\mathbf{y}\in Y(\boldsymbol{\alpha})} (-\mathbf{y})$$

$$\boldsymbol{\alpha}(k+1) = \boldsymbol{\alpha}(k) + \eta(k) \sum_{\mathbf{y}\in Y(\boldsymbol{\alpha})} \mathbf{y}$$

Perceptron rule (cont'd)

Algorithm 3 (Batch Perceptron)

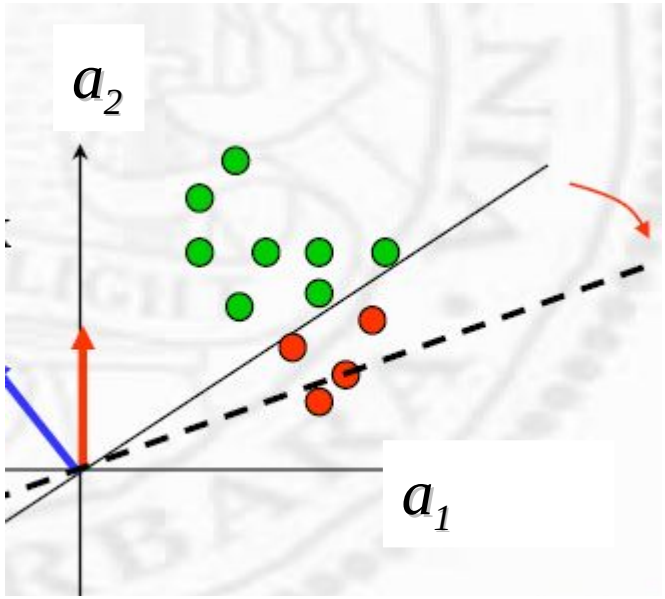
1 **begin** **initialize** $\mathbf{a}, \eta(\cdot), \text{criterion } \theta, k = 0$
2 **do** $k \leftarrow k + 1$
3 $\mathbf{a} \leftarrow \mathbf{a} + \eta(k) \sum_{\mathbf{y} \in \mathcal{Y}_k} \mathbf{y}$
4 **until** $\eta(k) \sum_{\mathbf{y} \in \mathcal{Y}_k} \mathbf{y} < \theta$
5 **return** a
6 **end**

missclassified examples

Perceptron rule (cont'd)

- Keep updating the orientation of the hyperplane until all training samples are on its positive side.

Example:



Perceptron rule (cont'd)

Algorithm 4 (Fixed-increment single-sample Perceptron)

1 begin initialize $\mathbf{a}, k = 0$
2 do $k \leftarrow (k+1) \bmod n$
3 if $\mathbf{y}_k$ is misclassified by $\mathbf{a}$ then $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + \eta(k)\mathbf{y}_k$
4 until all patterns properly classified
5 return a
6 end

Update is done using one misclassified example at a time

Perceptron Convergence Theorem: If training samples are linearly separable, then the perceptron algorithm will terminate at a solution vector in a finite number of steps.

Perceptron rule (cont'd)

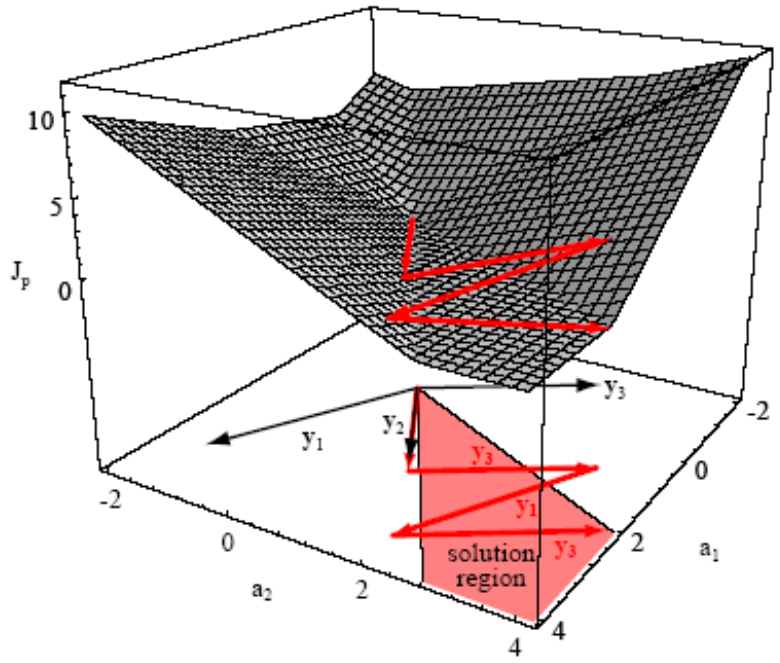


Figure 5.12: The Perceptron criterion, J_p is plotted as a function of the weights a_1 and a_2 for a three-pattern problem. The weight vector begins at $\mathbf{0}$, and the algorithm sequentially adds to it vectors equal to the "normalized" misclassified patterns themselves. In the example shown, this sequence is $\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_1, \mathbf{y}_3$, at which time the vector lies in the solution region and iteration terminates. Note that the second update (by $\mathbf{y}_3$) takes the candidate vector *farther* from the solution region than after the first update (cf. Theorem 5.1. (In an alternate, batch method, *all* the misclassified points are added at each iteration step leading to a smoother trajectory in weight space.)

order of examples:

y_2 y_3 y_1 y_3

"Batch" algorithm leads to a smoother trajectory in solution space.

Quiz

- When: April 21st
- What: Linear Discriminants

CS434a/541a: Pattern Recognition
Prof. Olga Veksler

Lecture 9

Announcements

- Final project proposal due Nov. 1
  - 1-2 paragraph description
  - Late Penalty: is 1 point off for each day late
- Assignment 3 due November 10
- Data for final project due Nov. 15
  - Must be ported in Matlab, send me .mat file with data and a short description file of what the data is
  - Late penalty is 1 point off for each day late
- Final project progress report
  - Meet with me the week of November 22-26
  - 5 points of if I will see you that have done NOTHNG yet
- Assignment 4 due December 1
- Final project due December 8

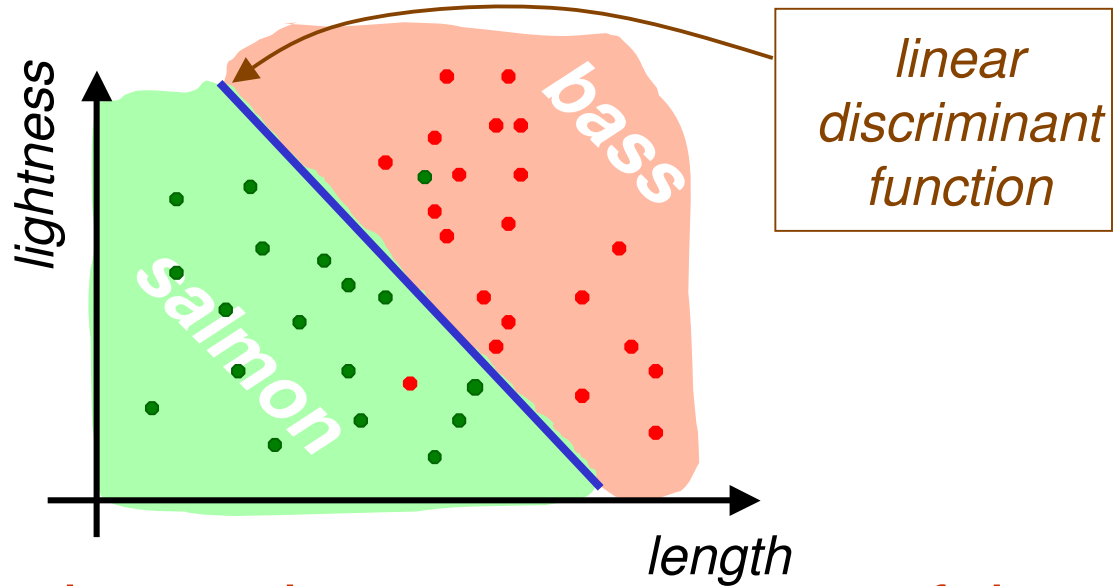
Today

- Linear Discriminant Functions
  - Introduction
  - 2 classes
  - Multiple classes
  - Optimization with gradient descent
  - Perceptron Criterion Function
    - Batch perceptron rule
    - Single sample perceptron rule

Linear discriminant functions on Road Map

- No probability distribution (no shape or parameters are known)
- Labeled data 

- The shape of discriminant functions is known

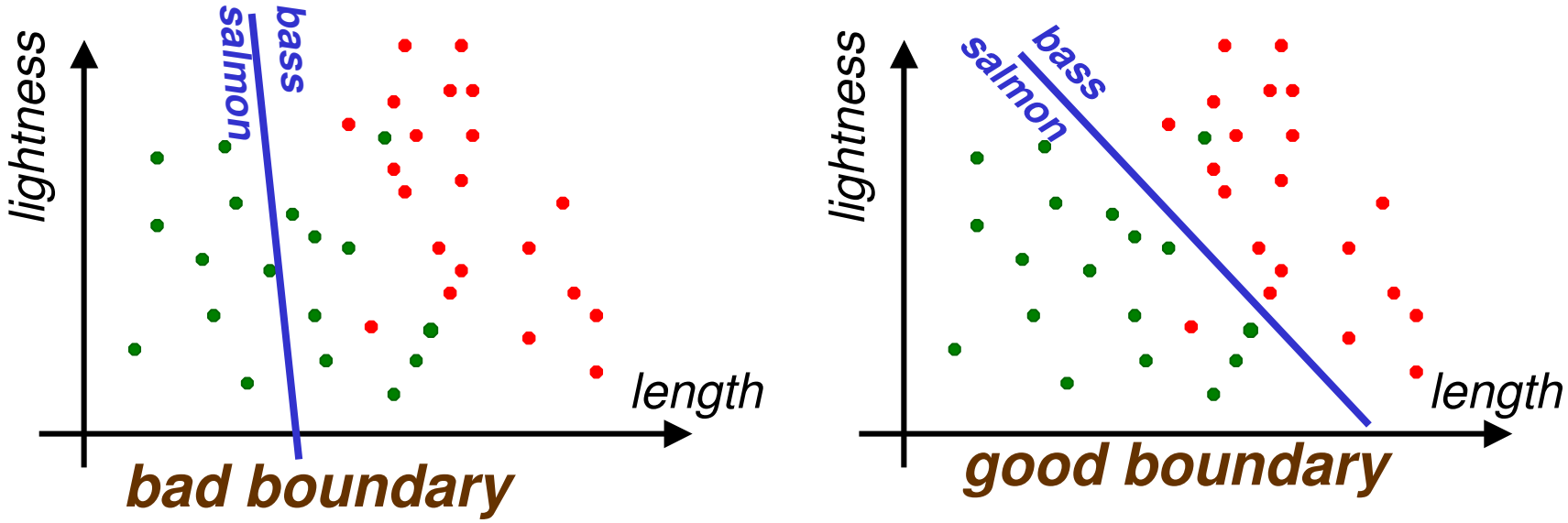


- Need to estimate parameters of the discriminant function (parameters of the line in case of linear discriminant)

a lot is known

little is known

Linear Discriminant Functions: Basic Idea



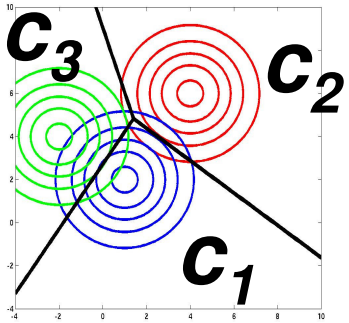
- Have samples from 2 classes $\boldsymbol{x_1}$, $\boldsymbol{x_2}$,…, $\boldsymbol{x_n}$
- Assume 2 classes can be separated by a linear boundary $\boldsymbol{l}(\theta)$ with some unknown parameters θ
- Fit the "best" boundary to data by optimizing over parameters θ
- What is best?
  - Minimize classification error on training data?
    - Does not guarantee small testing error

Parametric Methods vs. Discriminant Functions

Assume the shape of density for classes is known $p_1(x|\theta_1)$, $p_2(x|\theta_2)$,…

Estimate θ_1, θ_2,… from data

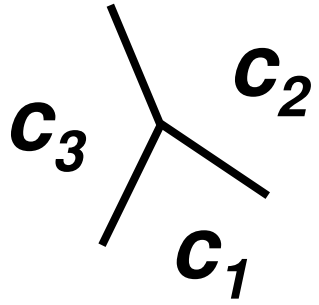
Use a Bayesian classifier to find decision regions



Assume discriminant functions are or known shape $l(\theta_1)$, $l(\theta_2)$, with parameters θ_1, θ_2,…

Estimate θ_1, θ_2,… from data

Use discriminant functions for classification



- In theory, Bayesian classifier minimizes the risk
  - In practice, do not have confidence in assumed model shapes
  - In practice, do not really need the actual density functions in the end
- Estimating accurate density functions is much harder than estimating accurate discriminant functions
- Some argue that estimating densities should be skipped
  - Why solve a harder problem than needed ?

LDF: Introduction

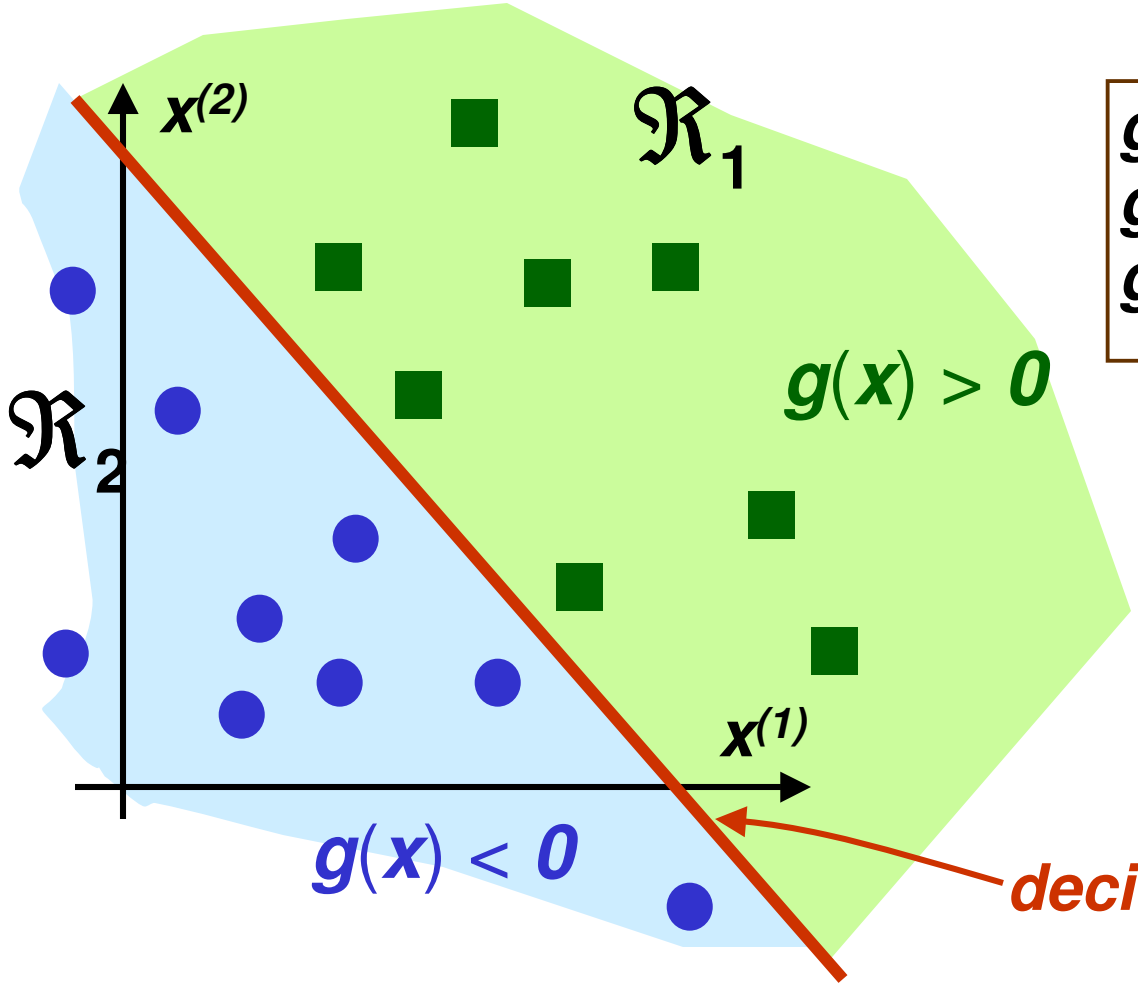
- Discriminant functions can be more general than linear
- For now, we will study linear discriminant functions
  - Simple model (should try simpler models first)
  - Analytically tractable
- Linear Discriminant functions are optimal for Gaussian distributions with equal covariance
- May not be optimal for other data distributions, but they are very simple to use
- Knowledge of class densities is not required when using linear discriminant functions
  - we can say that this is a non-parametric approach

LDF: 2 Classes

- A discriminant function is linear if it can be written as

$$g(x) = w^t x + w_0$$

  - w is called the weight vector and w_0 called bias or threshold

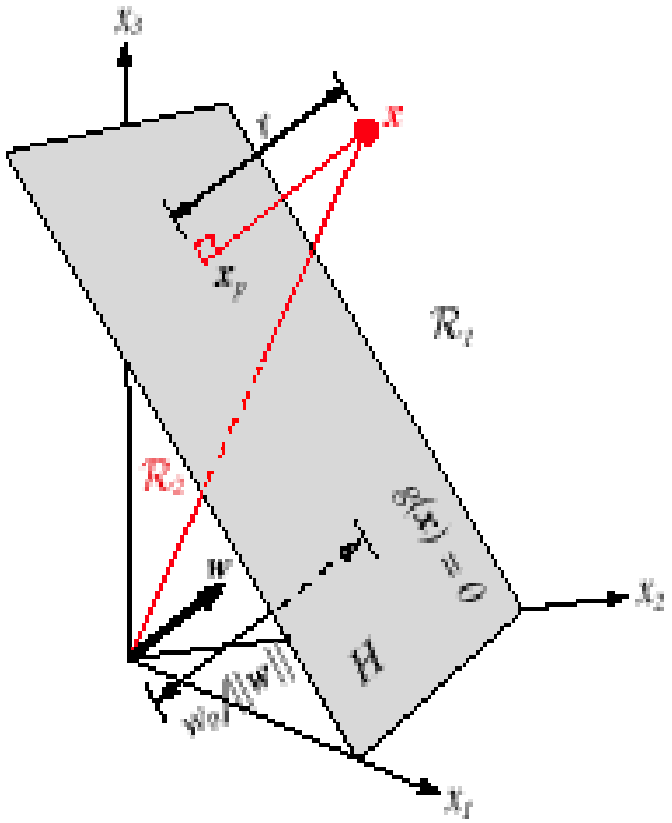


$$g(x) > 0 \Rightarrow x \in \textit{class 1}$$
$$g(x) < 0 \Rightarrow x \in \textit{class 2}$$
$$g(x) = 0 \Rightarrow \textit{either class}$$

LDF: 2 Classes

- Decision boundary $\boldsymbol{g(x) = w^t x + w_0 = 0}$ is a ***hyperplane***
  - set of vectors x which for some scalars $\alpha_0, \ldots, \alpha_d$ satisfy $\alpha_0 + \alpha_1 \boldsymbol{x}^{(1)} + \ldots + \alpha_d \boldsymbol{x}^{(d)} = 0$

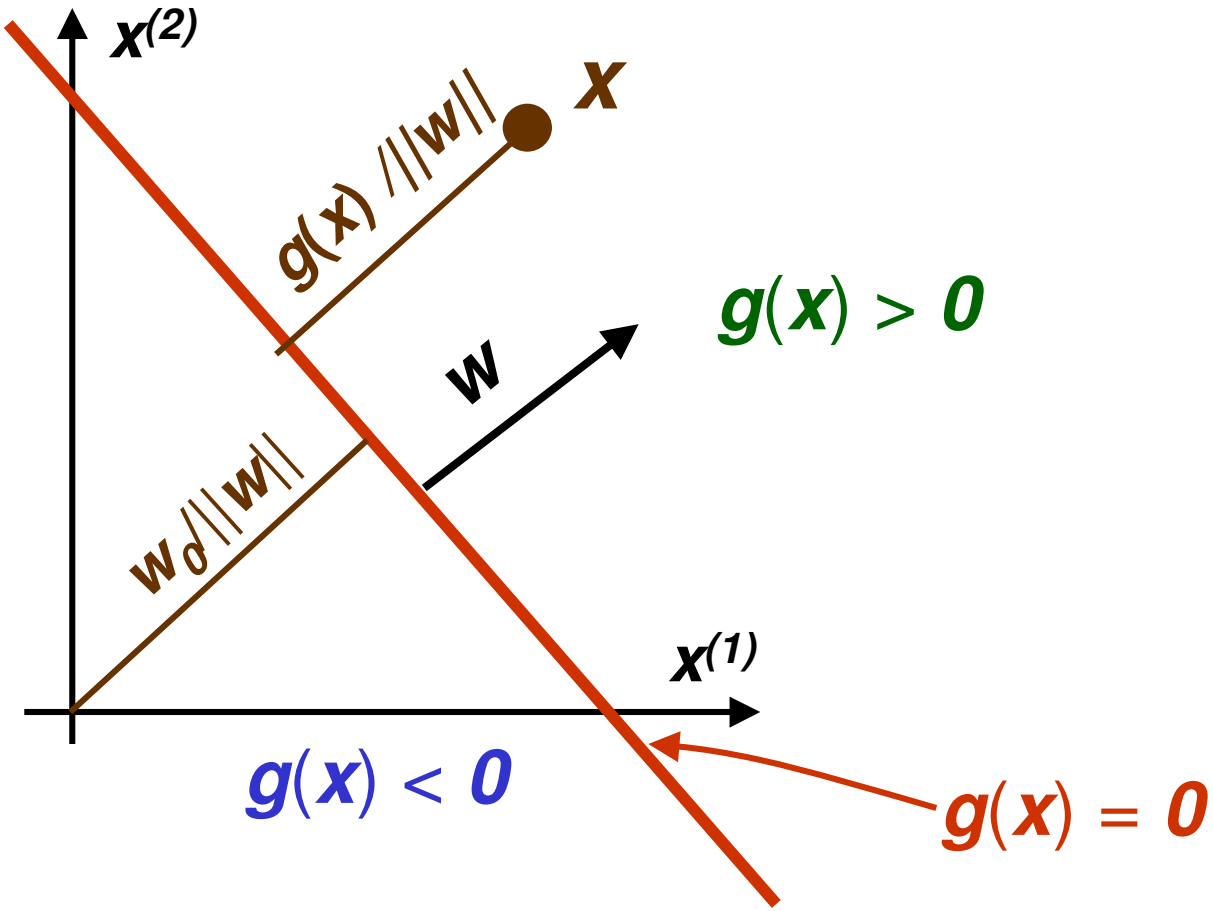
- A hyperplane is
  - a point in 1D
  - a line in 2D
  - a plane in 3D



LDF: 2 Classes

$$g(\boldsymbol{x}) = \boldsymbol{w}^t\boldsymbol{x} + w_0$$

- $\boldsymbol{w}$ determines orientation of the decision hyperplane
- w_0 determines location of the decision surface



LDF: 2 Classes

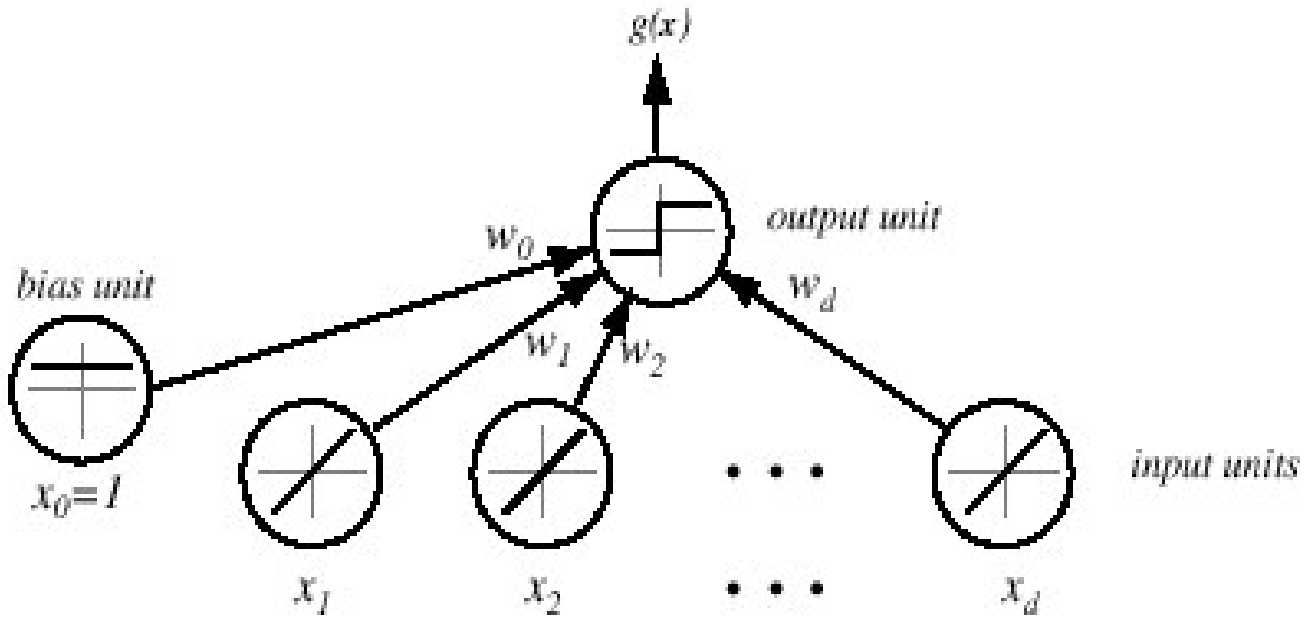


FIGURE 5.1. A simple linear classifier having d input units, each corresponding to the values of the components of an input vector. Each input feature value x_i is multiplied by its corresponding weight w_i; the effective input at the output unit is the sum all these products, $\sum w_i x_i$. We show in each unit its effective input-output function. Thus each of the d input units is linear, emitting exactly the value of its corresponding feature value. The single bias unit unit always emits the constant value 1.0. The single output unit emits a $+1$ if $\mathbf{w}^t\mathbf{x} + w_0 > 0$ or a -1 otherwise. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

LDF: Many Classes

- Suppose we have m classes
- Define m linear discriminant functions

$$g_i(x) = w_i^t x + w_{i0} \qquad i = 1,\ldots,m$$

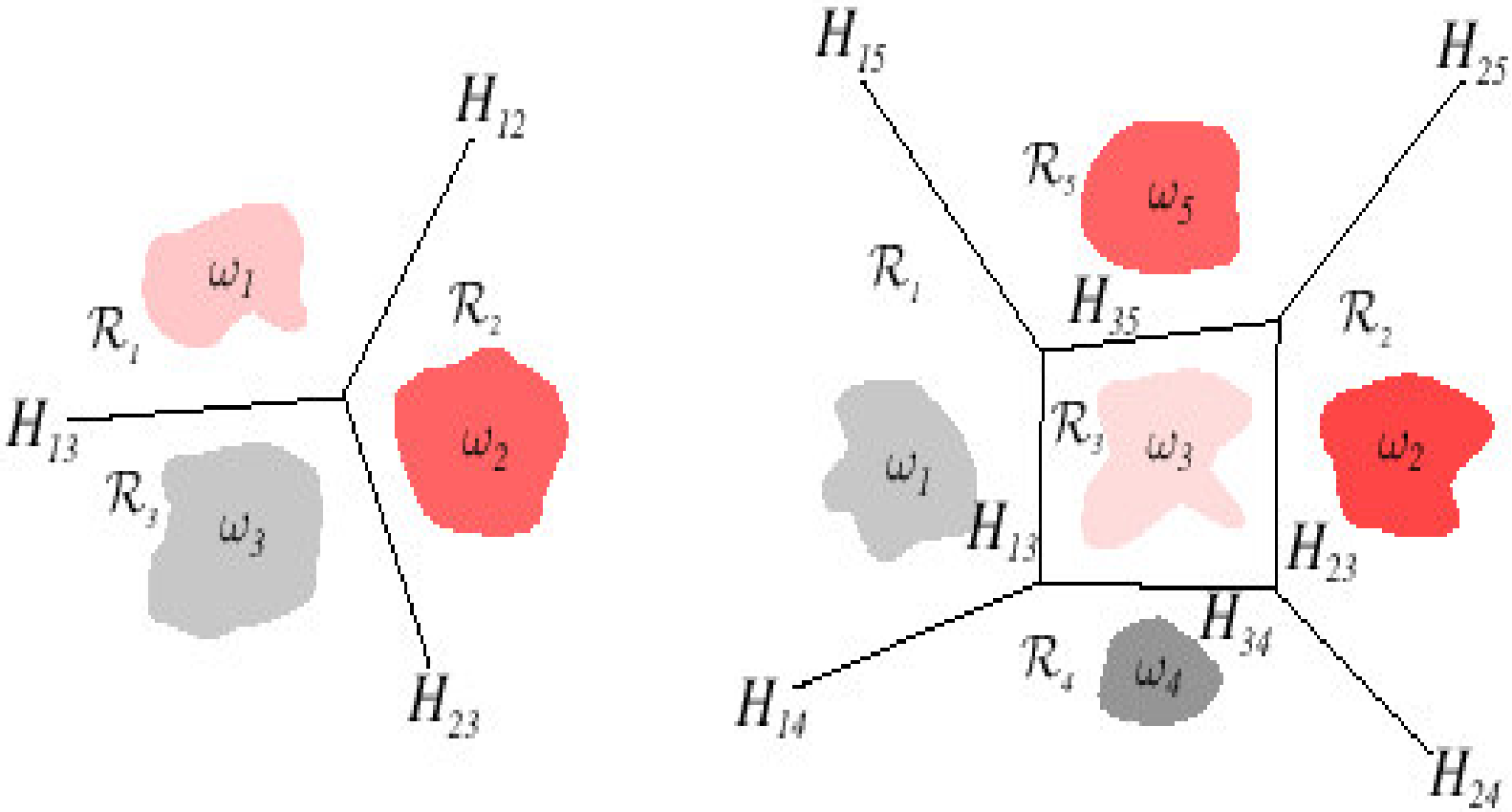
- Given x, assign class c_i if

$$g_i(x) \geq g_j(x) \qquad \forall j \neq i$$

- Such classifier is called a ***linear machine***

- A linear machine divides the feature space into c decision regions, with $g_i(x)$ being the largest discriminant if x is in the region R_i

LDF: Many Classes



LDF: Many Classes

- For a two contiguous regions R_i and R_j; the boundary that separates them is a portion of hyperplane $\boldsymbol{H_{ij}}$ defined by:

$$\begin{aligned} \boldsymbol{g_i(x) = g_j(x)} \quad &\Leftrightarrow \boldsymbol{w_i^t x + w_{i0} = w_j^t x + w_{j0}} \\ &\Leftrightarrow \boldsymbol{\left(w_i - w_j\right)^t x + \left(w_{i0} - w_{j0}\right) = 0} \end{aligned}$$

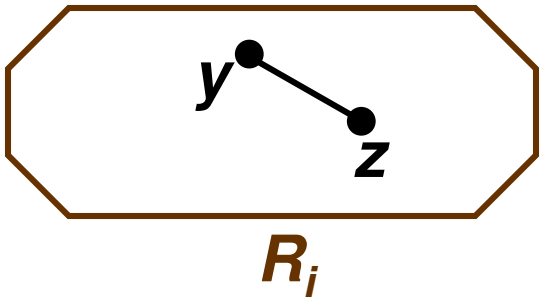
- Thus $\boldsymbol{w_i - w_j}$ is normal to $\boldsymbol{H_{ij}}$
- And distance from $\boldsymbol{x}$ to $\boldsymbol{H_{ij}}$ is given by

$$\boldsymbol{d(x, H_{ij}) = \frac{g_i(x) - g_j(x)}{\left\| w_i - w_j \right\|}}$$

LDF: Many Classes

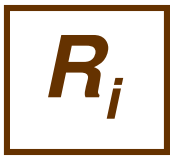
- Decision regions for a linear machine are ***convex***

$$\mathbf{y}, \mathbf{z} \in \mathbf{R}_i \Rightarrow \alpha \mathbf{y} + (1-\alpha)\mathbf{z} \in \mathbf{R}_i$$



$$\begin{aligned} &\forall j \neq i \quad g_i(\mathbf{y}) \geq g_j(\mathbf{y}) \text{ and } g_i(\mathbf{z}) \geq g_j(\mathbf{z}) \Leftrightarrow \\ \Leftrightarrow &\forall j \neq i \quad g_i(\alpha \mathbf{y} + (1-\alpha)\mathbf{z}) \geq g_j(\alpha \mathbf{y} + (1-\alpha)\mathbf{z}) \end{aligned}$$

- In particular, decision regions must be spatially contiguous

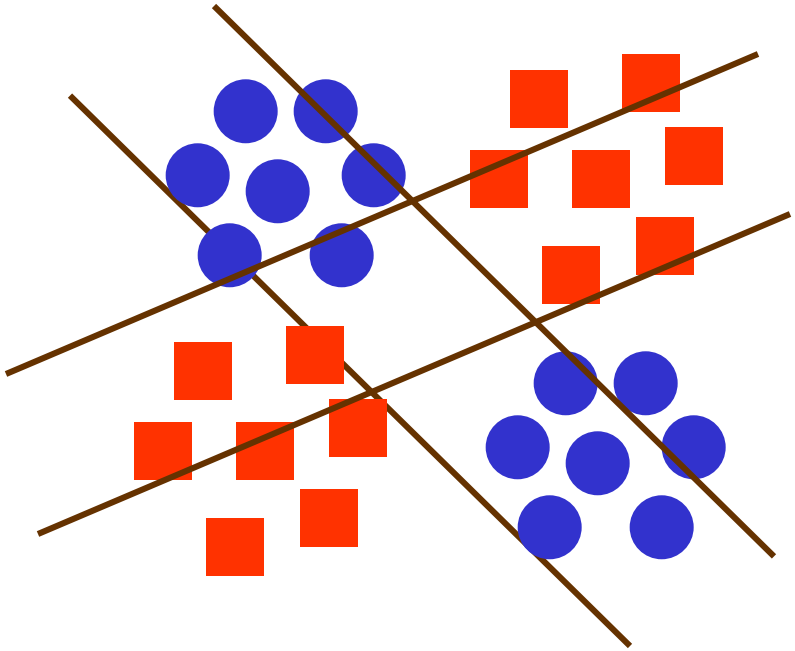


R_j is a valid decision region

R_j is not a valid decision region

LDF: Many Classes

- Thus applicability of linear machine to mostly limited to unimodal conditional densities $\boldsymbol{p}(\boldsymbol{x}|\theta)$
  - even though we did not assume any parametric models
- Example:



  - need non-contiguous decision regions
  - thus linear machine will fail

LDF: Augmented feature vector

- Linear discriminant function: $g(x) = w^t x + w_0$
- Can rewrite it: $g(x) = \underbrace{\begin{bmatrix} w_0 & w^t \end{bmatrix}}_{\text{new weight vector a}} \underbrace{\begin{bmatrix} 1 \\ x \end{bmatrix}}_{\text{new feature vector y}} = a^t y = g(y)$
- y is called the ***augmented feature vector***
- Added a dummy dimension to get a completely equivalent new ***homogeneous*** problem

old problem

$$g(x) = w^t x + w_0$$

$$\begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}$$

new problem

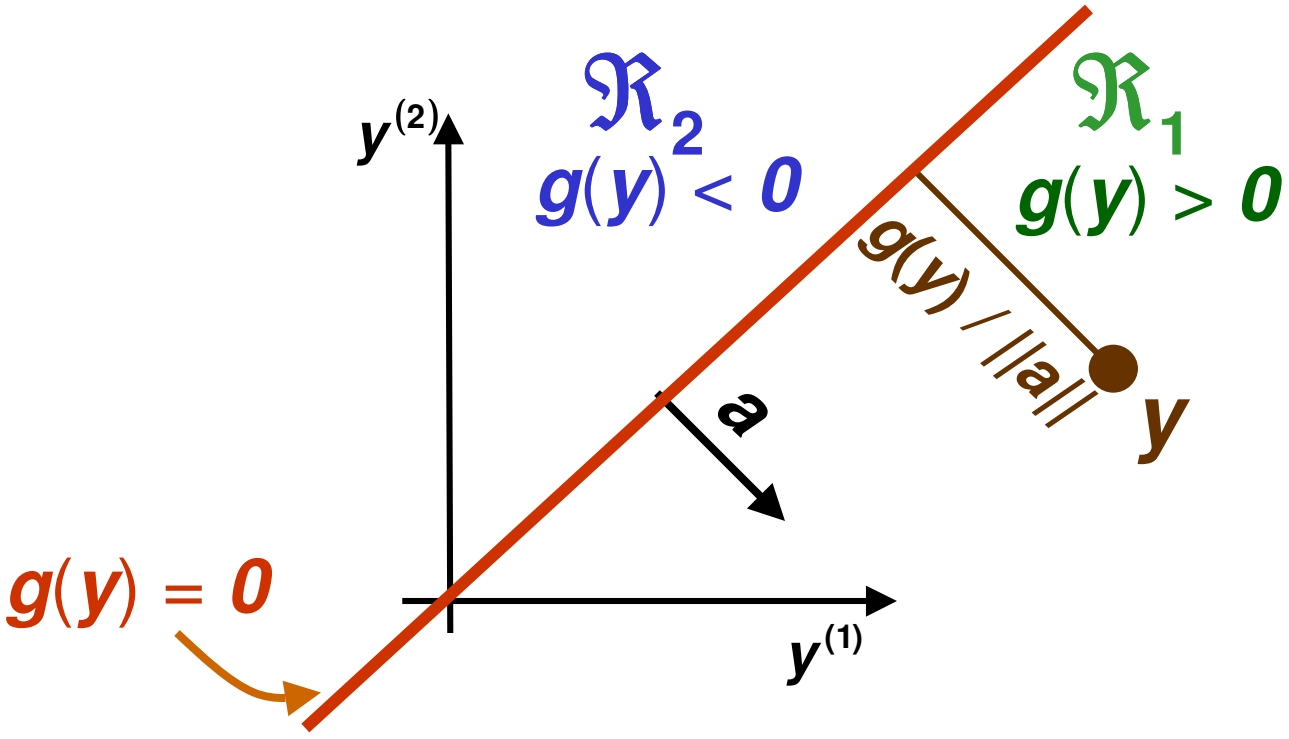
$$g(y) = a^t y$$

$$\begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}$$

LDF: Augmented feature vector

- Feature augmenting is done for simpler notation
- From now on we always assume that we have augmented feature vectors
  - Given samples $\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n$ convert them to augmented samples $\boldsymbol{y}_1,\ldots,\boldsymbol{y}_n$ by adding a new dimension of value 1

$$\boldsymbol{y}_i = \begin{bmatrix} 1 \\ \boldsymbol{x}_i \end{bmatrix}$$



LDF: Training Error

- For the rest of the lecture, assume we have 2 classes
- Samples $\boldsymbol{y_1},\ldots,\boldsymbol{y_n}$ some in class 1, some in class 2
- Use these samples to determine weights $\boldsymbol{a}$ in the discriminant function $\boldsymbol{g(y) = a^t y}$
- What should be our criterion for determining $\boldsymbol{a}$?
  - For now, suppose we want to minimize the training error (that is the number of misclassifed samples $\boldsymbol{y_1},\ldots,\boldsymbol{y_n}$)
- Recall that

$$\boldsymbol{g(y_i) > 0 \Rightarrow y_i \ \ classified \ \ c_1}$$
$$\boldsymbol{g(y_i) < 0 \Rightarrow y_i \ \ classified \ \ c_2}$$

- Thus training error is $\boldsymbol{0}$ if

$$\begin{cases} \boldsymbol{g(y_i) > 0} & \boldsymbol{\forall y_i \in c_1} \\ \boldsymbol{g(y_i) < 0} & \boldsymbol{\forall y_i \in c_2} \end{cases}$$

LDF: Problem "Normalization"

- Thus training error is **0** if $\begin{cases} \boldsymbol{a}^t \boldsymbol{y}_i > 0 & \forall \boldsymbol{y}_i \in \boldsymbol{c}_1 \\ \boldsymbol{a}^t \boldsymbol{y}_i < 0 & \forall \boldsymbol{y}_i \in \boldsymbol{c}_2 \end{cases}$
- Equivalently, training error is **0** if

$$\begin{cases} \boldsymbol{a}^t \boldsymbol{y}_i > 0 & \forall \boldsymbol{y}_i \in \boldsymbol{c}_1 \\ \boldsymbol{a}^t (-\boldsymbol{y}_i) > 0 & \forall \boldsymbol{y}_i \in \boldsymbol{c}_2 \end{cases}$$

- This suggest problem "normalization":
  1. Replace all examples from class $\boldsymbol{c}_2$ by their negative

$$\boldsymbol{y}_i \rightarrow -\boldsymbol{y}_i \qquad \forall \boldsymbol{y}_i \in \boldsymbol{c}_2$$

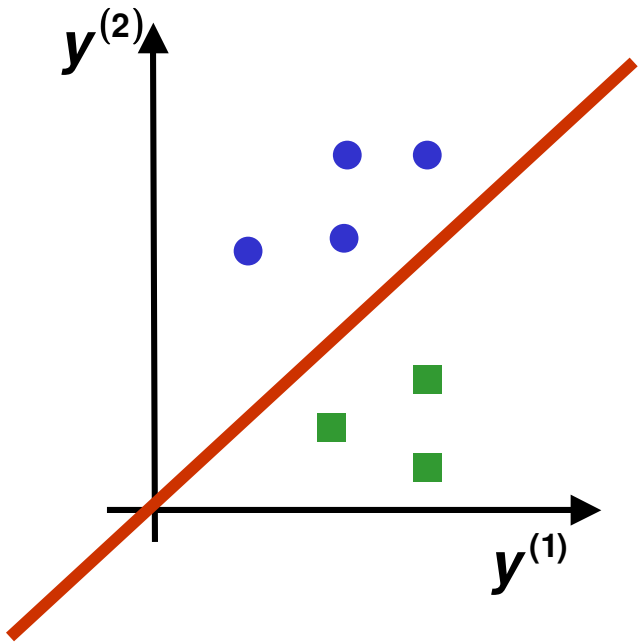
  2. Seek weight vector $\boldsymbol{a}$ s.t.

$$\boldsymbol{a}^t \boldsymbol{y}_i > 0 \qquad \forall \boldsymbol{y}_i$$

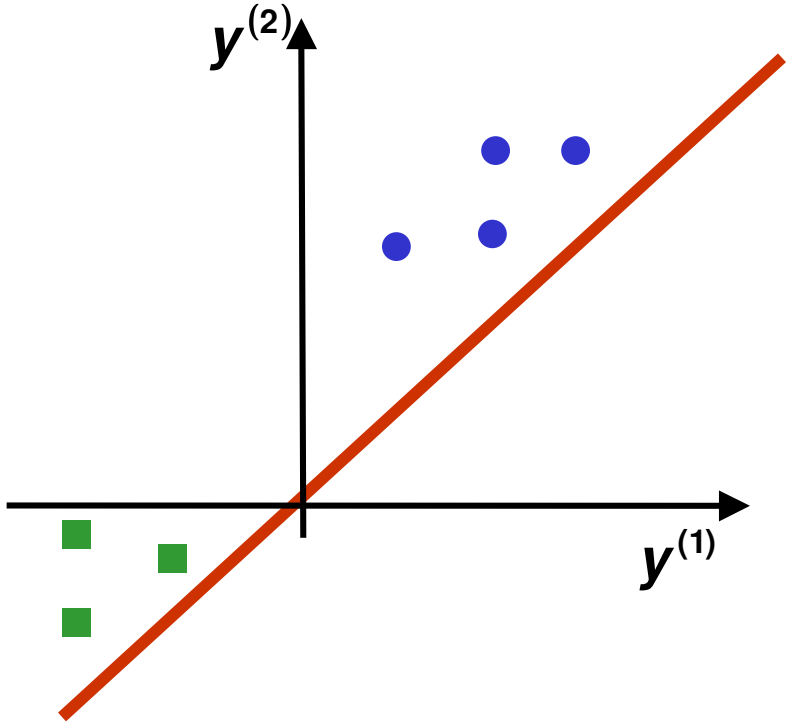
     - If such $\boldsymbol{a}$ exists, it is called a *separating* or *solution* vector
     - Original samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ can indeed be separated by a line then

LDF: Problem "Normalization"

before normalization



after "normalization"



Seek a hyperplane that separates patterns from different categories

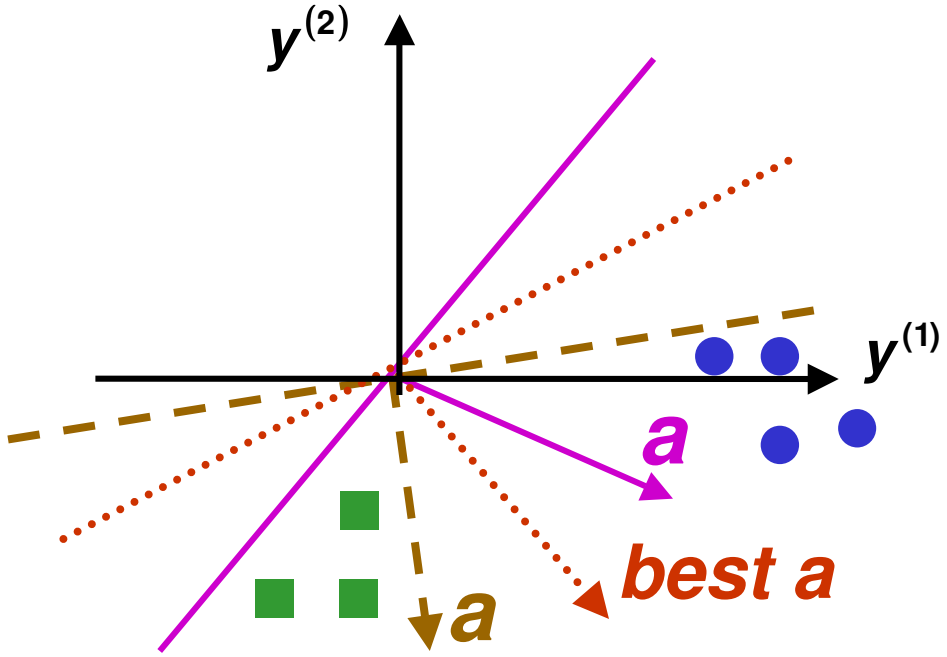
⇦

Seek hyperplane that puts *normalized* patterns on the same (positive) side

LDF: Solution Region

- Find weight vector $\boldsymbol{a}$ s.t. for all samples $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$

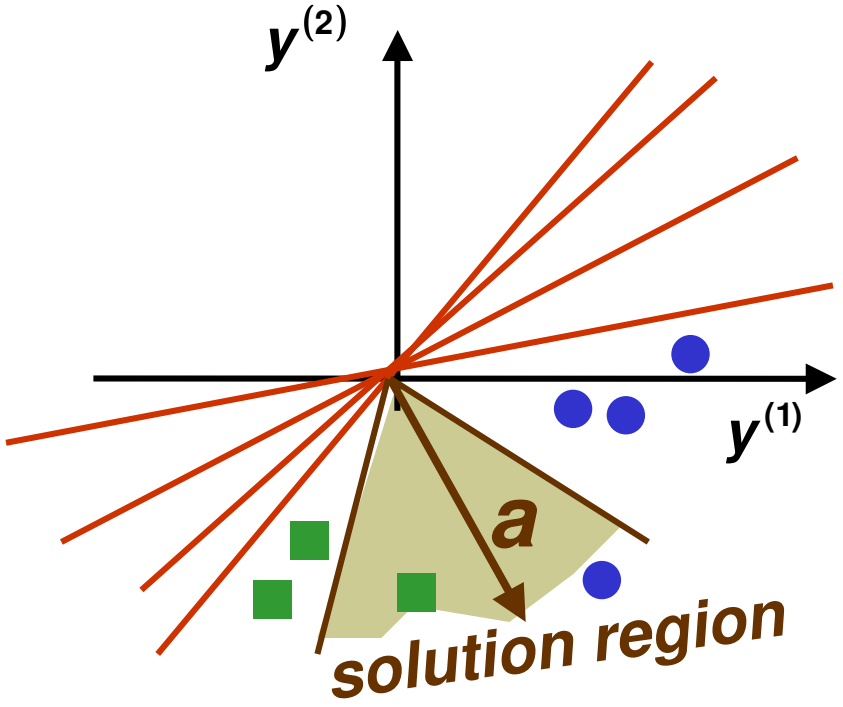
$$\boldsymbol{a}^t \boldsymbol{y}_i = \sum_{k=0}^{d} \boldsymbol{a}_k \boldsymbol{y}_i^{(k)} > 0$$



- In general, there are many such solutions $\boldsymbol{a}$

LDF: Solution Region

- ***Solution region*** for ***a***: set of all possible solutions
  - defined in terms of normal ***a*** to the separating hyperplane



Optimization

- Need to minimize a function of many variables

$$J(\boldsymbol{x}) = J(x_1, \ldots, x_d)$$

- We know how to minimize $J(\boldsymbol{x})$
  - Take partial derivatives and set them to zero

$$\begin{bmatrix} \frac{\partial}{\partial x_1} J(\boldsymbol{x}) \\ \vdots \\ \frac{\partial}{\partial x_d} J(\boldsymbol{x}) \end{bmatrix} = \nabla J(\boldsymbol{x}) = \mathbf{0}$$

gradient

- However solving analytically is not always easy
  - Would you like to solve this system of nonlinear equations?

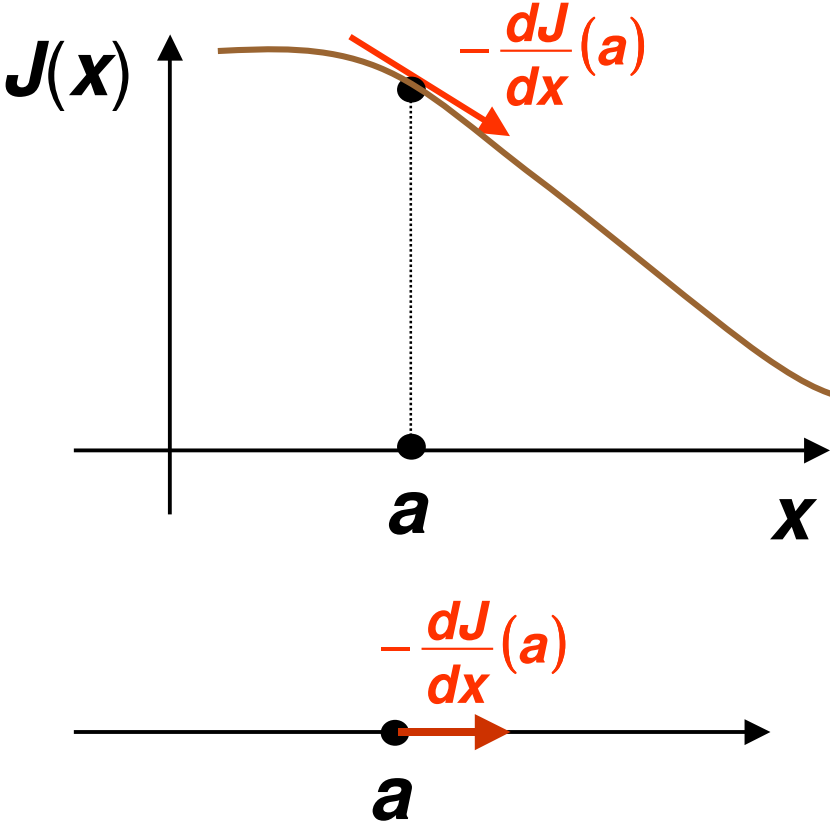
$$\begin{cases} \sin\left(x_1^2 + x_2^3\right) + e^{x_4^2} = 0 \\ \cos\left(x_1^2 + x_2^3\right) + \log\left(x_5^3\right)^{x_4^2} = 0 \end{cases}$$

  - Sometimes it is not even possible to write down an analytical expression for the derivative, we will see an example later today

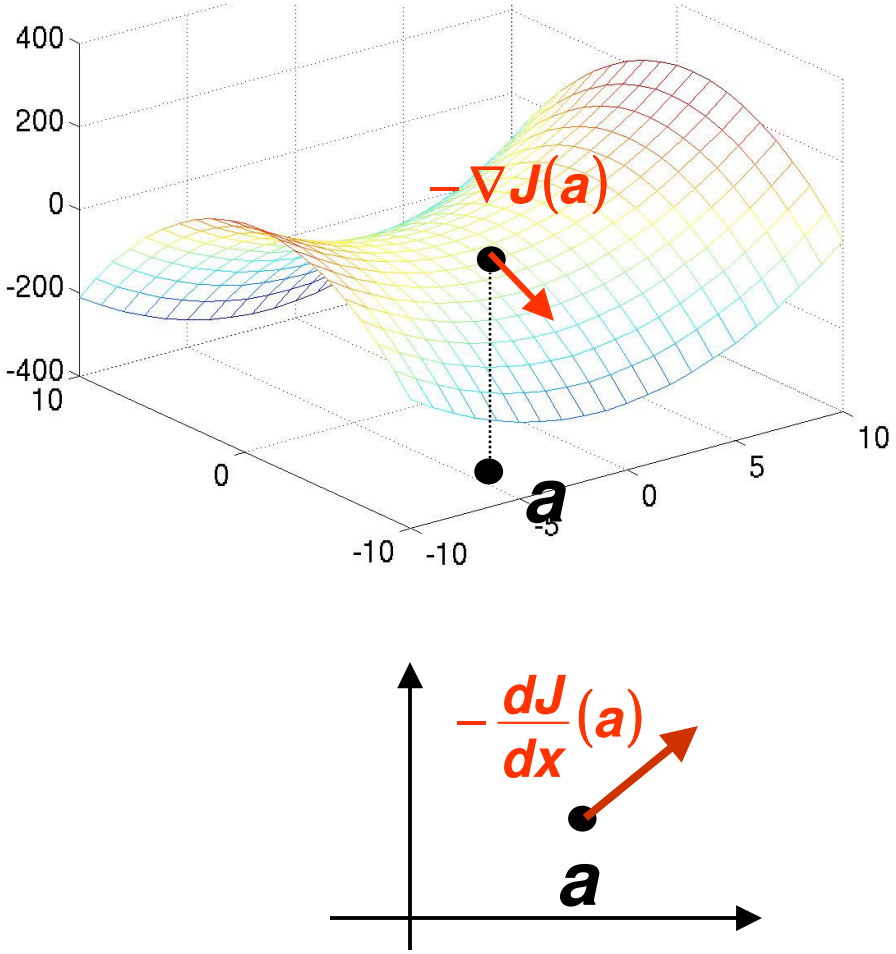
Optimization: Gradient Descent

- Gradient $\nabla J(\boldsymbol{x})$ points in direction of steepest increase of $J(\boldsymbol{x})$, and $-\nabla J(\boldsymbol{x})$ in direction of steepest decrease

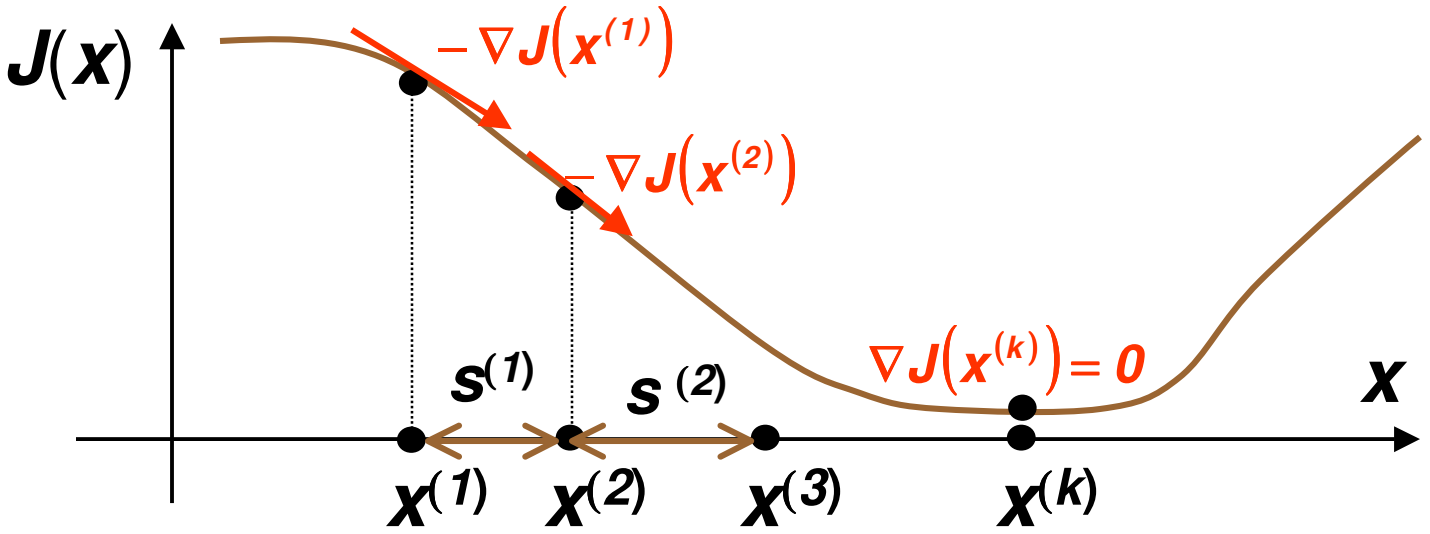
one dimension



two dimensions



Optimization: Gradient Descent



Gradient Descent for minimizing any function $J(x)$

set $k = 1$ ***and*** $x^{(1)}$ to some initial guess for the weight vector

while $\eta^{(k)} \left| \nabla J\left(x^{(k)}\right) \right| > \varepsilon$

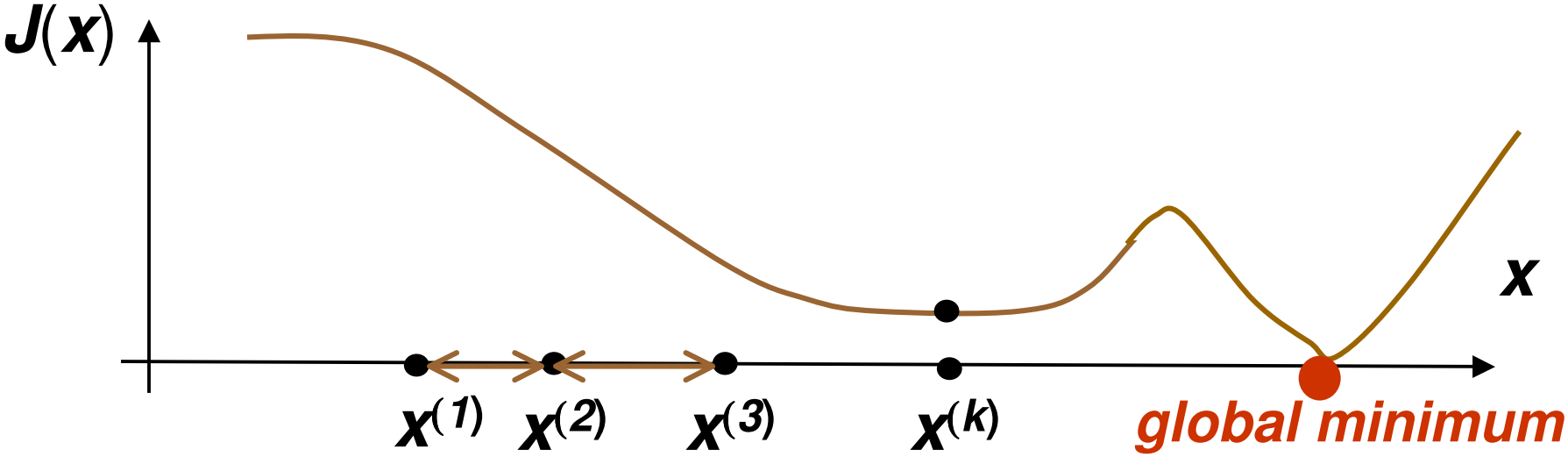
choose learning rate $\eta^{(k)}$

$$x^{(k+1)} = x^{(k)} - \eta^{(k)} \nabla J(x)$$ (***update rule***)

$$k = k + 1$$

Optimization: Gradient Descent

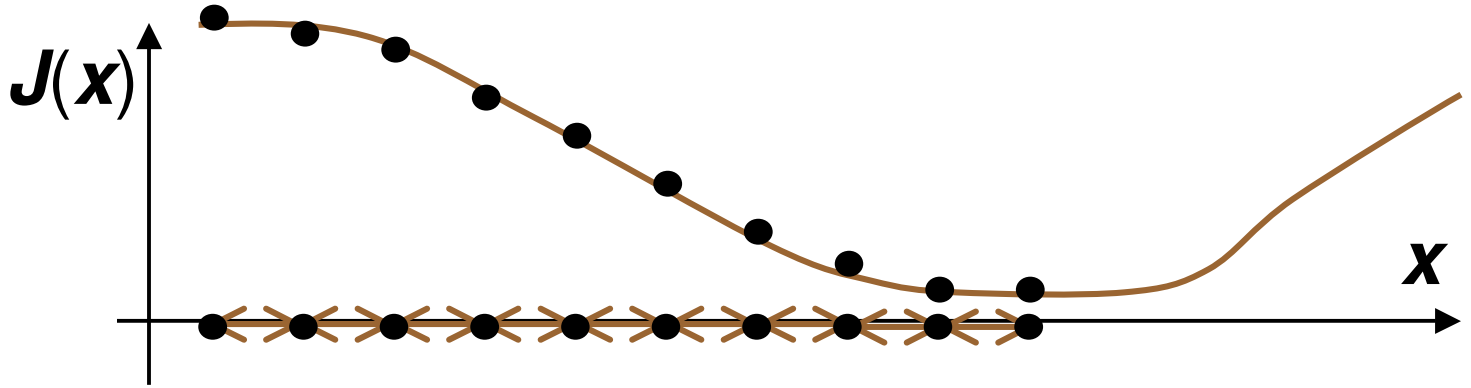
- Gradient descent is guaranteed to find only a local minimum



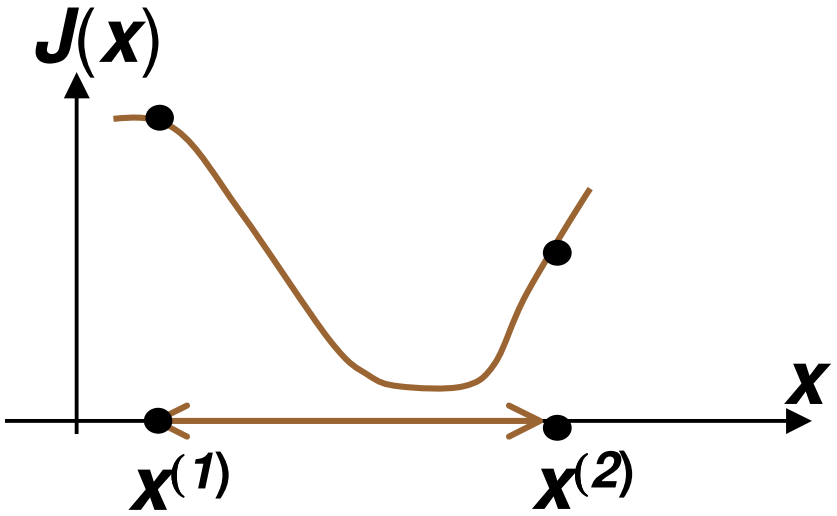
- Nevertheless gradient descent is very popular because it is simple and applicable to any function

Optimization: Gradient Descent

- Main issue: how to set parameter η (***learning rate***)
- If η is too small, need too many iterations



- If η is too large may overshoot the minimum and possibly never find it (if we keep overshooting)



Today

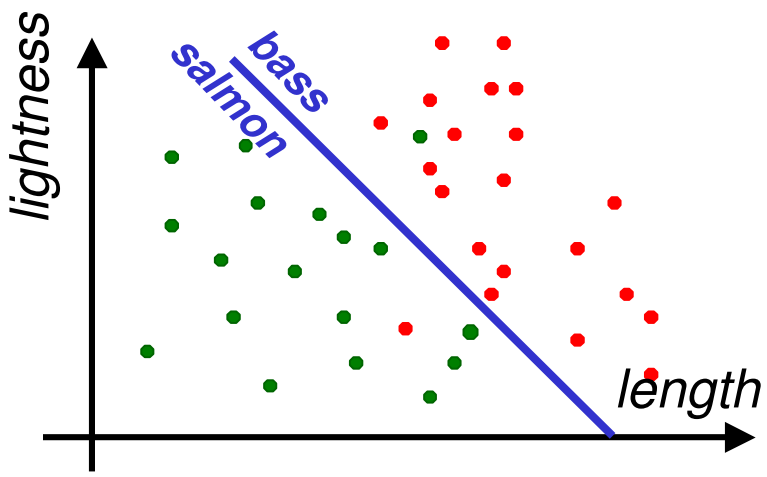
- Continue Linear Discriminant Functions
  - Perceptron Criterion Function
    - Batch perceptron rule
    - Single sample perceptron rule

LDF: Augmented feature vector

- Linear discriminant function:

$$g(\boldsymbol{x}) = \boldsymbol{w}^t \boldsymbol{x} + w_0$$

  - need to estimate parameters $\boldsymbol{w}$ and $\boldsymbol{w_0}$ from data



- Augment samples $\boldsymbol{x}$ to get equivalent homogeneous problem in terms of samples $\boldsymbol{y}$:

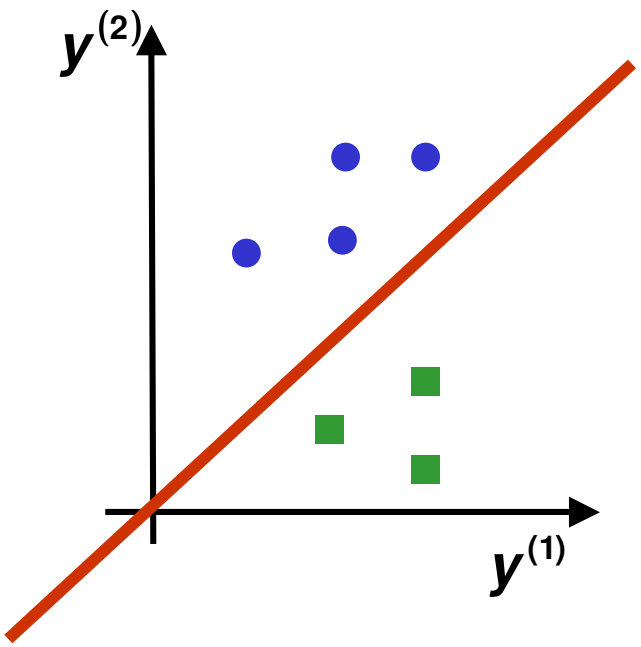
$$g(\boldsymbol{x}) = \begin{bmatrix} w_0 & \boldsymbol{w}^t \end{bmatrix} \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix} = \boldsymbol{a}^t \boldsymbol{y} = g(\boldsymbol{y})$$

- "normalize" by replacing all examples from class $\boldsymbol{c_2}$ by their negative

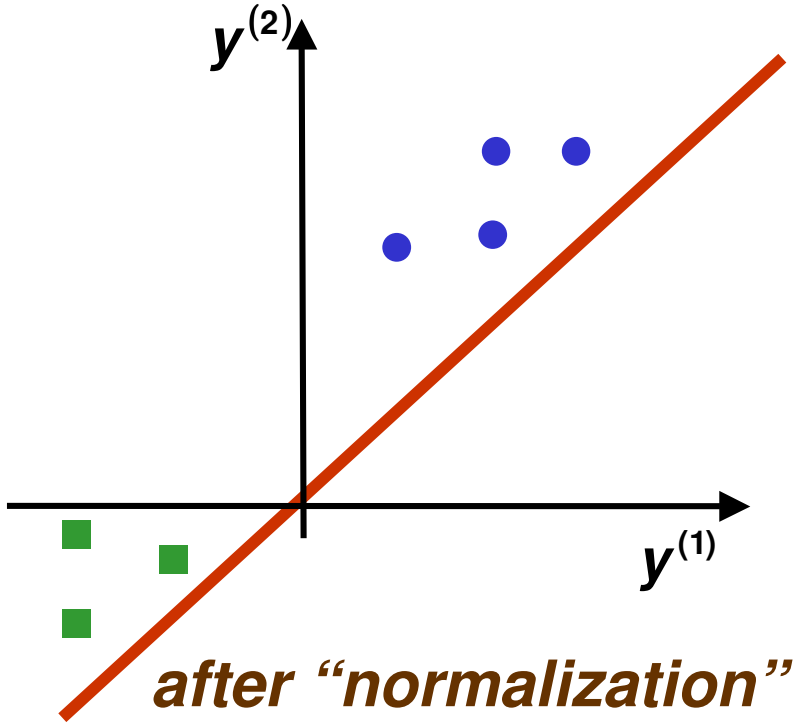
$$\boldsymbol{y}_i \rightarrow -\boldsymbol{y}_i \qquad \forall \boldsymbol{y}_i \in \boldsymbol{c}_2$$

LDF

- Augmented and "normalized" samples $\boldsymbol{y_1},\ldots,\boldsymbol{y_n}$
- Seek weight vector $\boldsymbol{a}$ s.t. $\boldsymbol{a}^t\boldsymbol{y}_i > \mathbf{0} \qquad \forall \boldsymbol{y}_i$



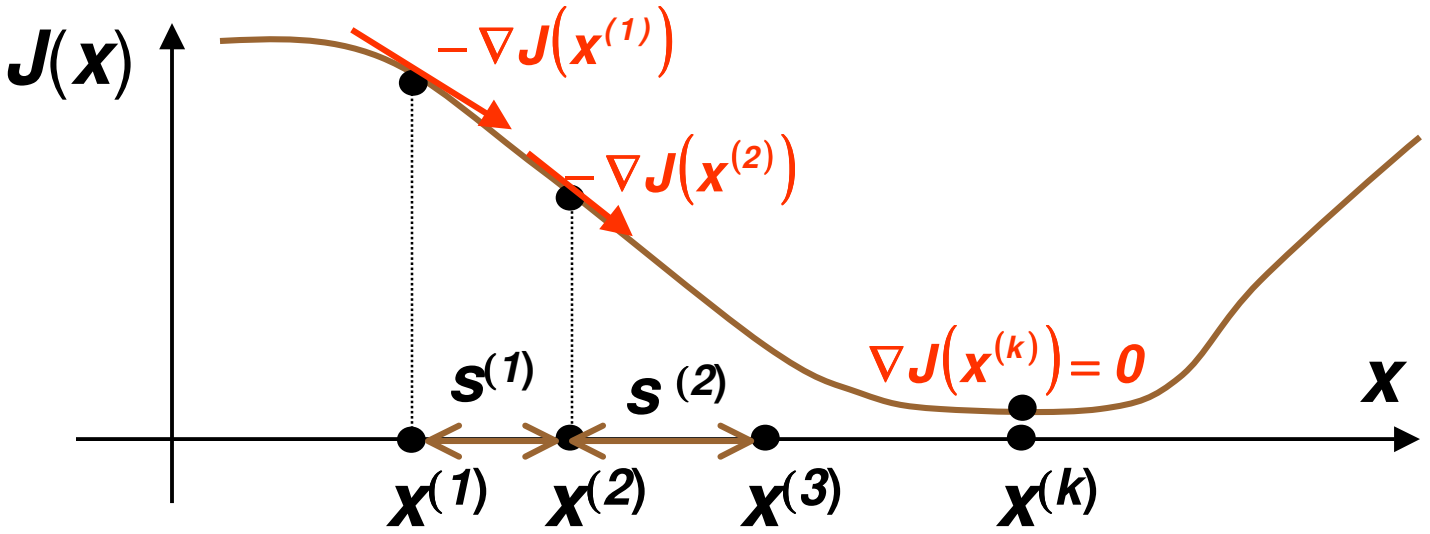
before normalization



after "normalization"

- If such $\boldsymbol{a}$ exists, it is called a *separating* or *solution* vector
- original samples $\boldsymbol{x_1},\ldots,\boldsymbol{x_n}$ can indeed be separated by a line then

Optimization: Gradient Descent



$$s^{(k+1)} = x^{(k+1)} - x^{(k)} = \eta^{(k)}\left(-\nabla J\left(x^{(k)}\right)\right)$$

Gradient Descent for minimizing any function $J(x)$

set $k = 1$ ***and*** $x^{(1)}$ to some initial guess for the weight vector

while $\eta^{(k)}\left|\nabla J\left(x^{(k)}\right)\right| > \varepsilon$

choose learning rate $\eta^{(k)}$

$x^{(k+1)} = x^{(k)} - \eta^{(k)} \nabla J(x)$ (***update rule***)

$k = k + 1$

LDF: Criterion Function

- Find weight vector $\boldsymbol{a}$ s.t. for all samples $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$

$$\boldsymbol{a}^t \boldsymbol{y}_i = \sum_{k=0}^{d} \boldsymbol{a}_k \boldsymbol{y}_i^{(k)} > \boldsymbol{0}$$

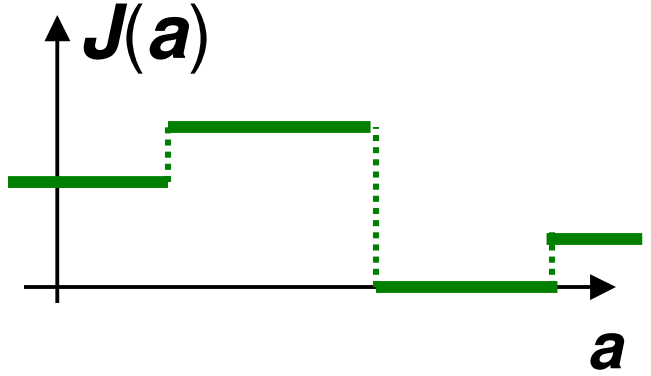
- Need criterion function $\boldsymbol{J}(\boldsymbol{a})$ which is minimized when $\boldsymbol{a}$ is a solution vector
- Let $\boldsymbol{Y_M}$ be the set of examples misclassified by $\boldsymbol{a}$

$$\boldsymbol{Y_M}(\boldsymbol{a}) = \{\boldsymbol{sample}\ \ \boldsymbol{y}_i\ \ \boldsymbol{s.t.}\ \ \boldsymbol{a}^t \boldsymbol{y}_i < \boldsymbol{0}\}$$

- First natural choice: number of misclassified examples

$$\boldsymbol{J}(\boldsymbol{a}) = |\boldsymbol{Y_M}(\boldsymbol{a})|$$

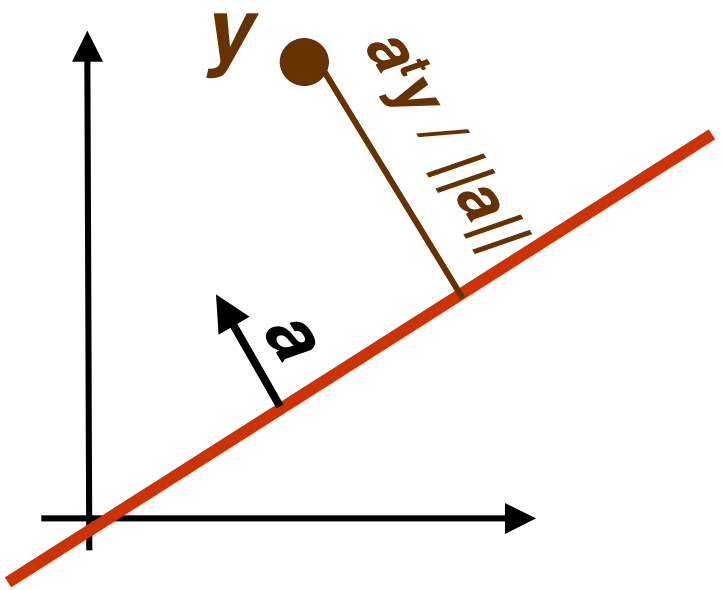
  - piecewise constant, gradient descent is useless



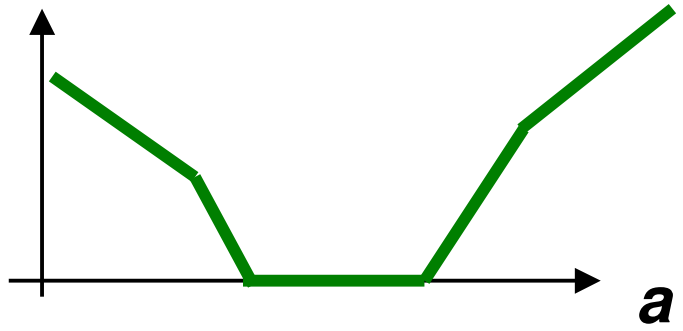
LDF: Perceptron Criterion Function

- Better choice: ***Perceptron*** criterion function

$$J_p(\boldsymbol{a}) = \sum_{\boldsymbol{y} \in Y_M} \left(-\boldsymbol{a}^t \boldsymbol{y}\right)$$



- If $\boldsymbol{y}$ is misclassified, $\boldsymbol{a}^t \boldsymbol{y} \leq 0$
- Thus $J_p(\boldsymbol{a}) \geq 0$
- $J_p(\boldsymbol{a})$ is $\|\boldsymbol{a}\|$ times sum of distances of misclassified examples to decision boundary
- $J_p(\boldsymbol{a})$ is piecewise linear and thus suitable for gradient descent



LDF: Perceptron Batch Rule

$$J_p(\boldsymbol{a}) = \sum_{\boldsymbol{y} \in \boldsymbol{Y}_M} (-\boldsymbol{a}^t \boldsymbol{y})$$

- Gradient of $J_p(\boldsymbol{a})$ is $\nabla J_p(\boldsymbol{a}) = \sum_{\boldsymbol{y} \in \boldsymbol{Y}_M} (-\boldsymbol{y})$
  - $\boldsymbol{Y}_M$ are samples misclassified by $\boldsymbol{a}^{(k)}$
  - It is not possible to solve $\nabla J_p(\boldsymbol{a}) = \boldsymbol{0}$ analytically because of $\boldsymbol{Y}_M$
- Update rule for gradient descent: $\boldsymbol{x}^{(k+1)} = \mathbf{x}^{(k)} - \eta^{(k)} \nabla J(\boldsymbol{x})$
- Thus *gradient decent batch update rule* for $J_p(\boldsymbol{a})$ is:

$$\boldsymbol{a}^{(k+1)} = \boldsymbol{a}^{(k)} + \eta^{(k)} \sum_{\boldsymbol{y} \in \boldsymbol{Y}_M} \boldsymbol{y}$$

- It is called batch rule because it is based on all misclassified examples

LDF: Perceptron Single Sample Rule

- Thus *gradient decent single sample rule* for $J_p(\boldsymbol{a})$ is:

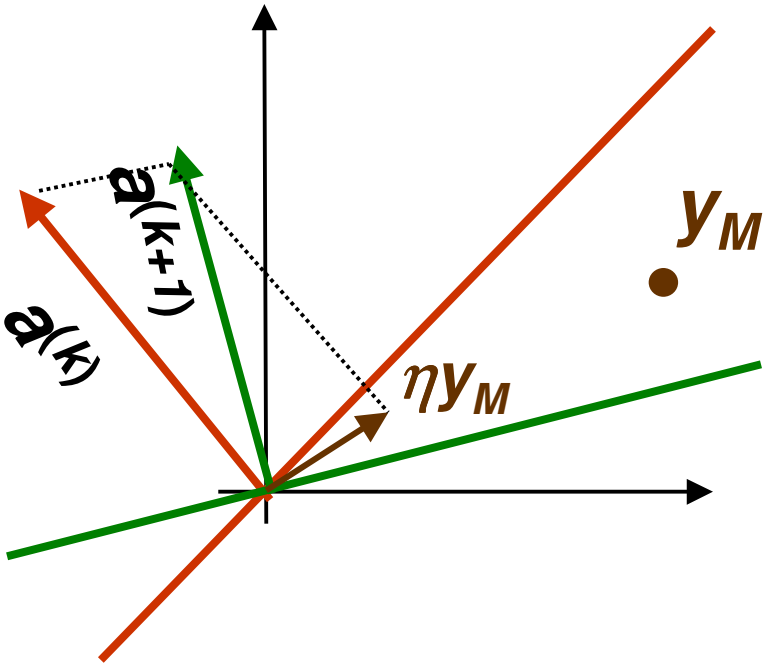
$$\boldsymbol{a}^{(k+1)} = \boldsymbol{a}^{(k)} + \eta^{(k)} \boldsymbol{y}_M$$

  - note that y_M is one sample misclassified by $\boldsymbol{a}^{(k)}$
  - must have a consistent way of visiting samples

- Geometric Interpretation:
  - y_M misclassified by $\boldsymbol{a}^{(k)}$

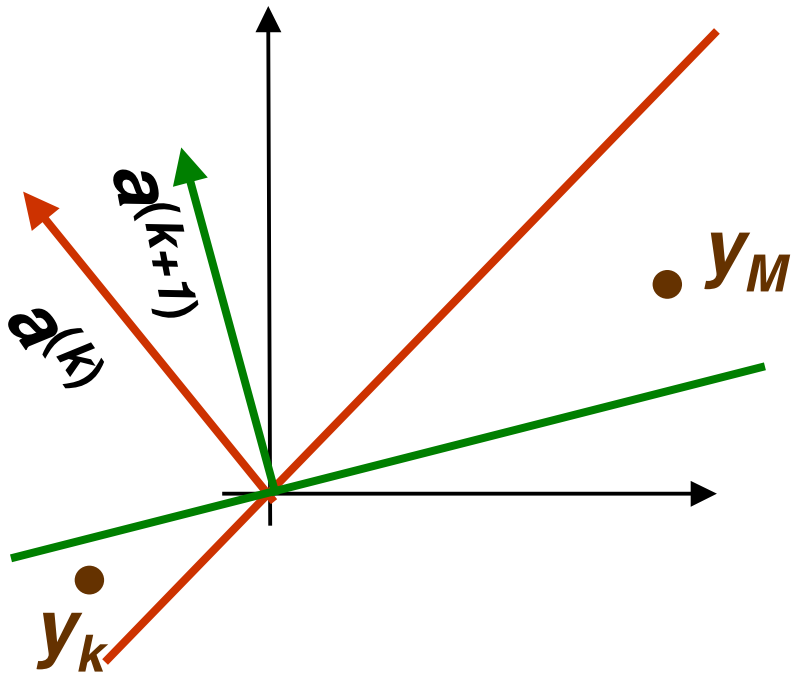
$$\left(\boldsymbol{a}^{(k)}\right)^t \boldsymbol{y}_M \leq 0$$

  - y_M is on the wrong side of decision hyperplane
  - adding $\eta \boldsymbol{y}_M$ to $\boldsymbol{a}$ moves new decision hyperplane in the right direction with respect to $\boldsymbol{y}_M$

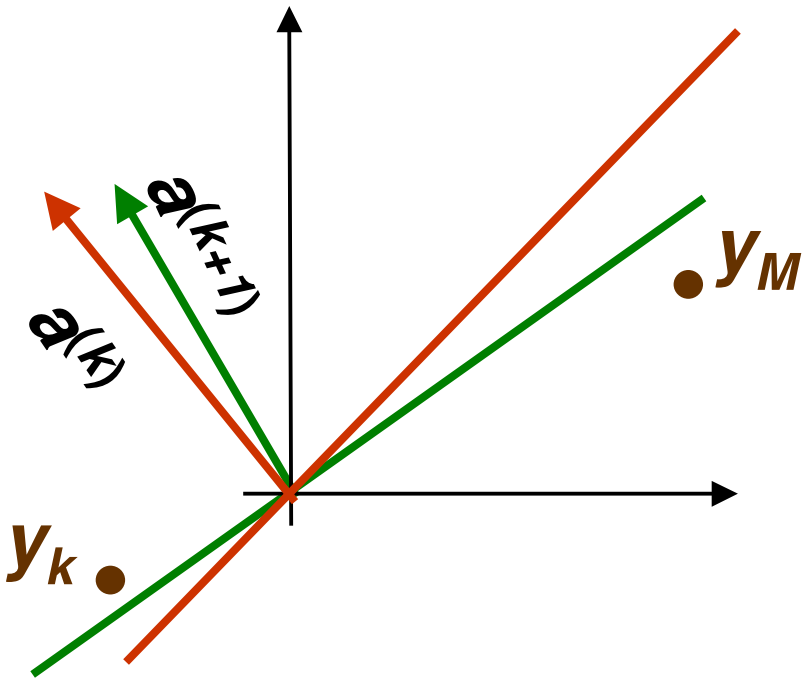


LDF: Perceptron Single Sample Rule

$$\boldsymbol{a}^{(k+1)} = \boldsymbol{a}^{(k)} + \eta^{(k)} \boldsymbol{y}_M$$



η is too large, previously correctly classified sample $\boldsymbol{y}_k$ is now misclassified



η is too small, $\boldsymbol{y}_M$ is still misclassified

LDF: Perceptron Example

	features				grade
name	***good attendance?***	***tall?***	***sleeps in class?***	***chews gum?***	
Jane	*yes (1)*	*yes (1)*	*no (-1)*	*no (-1)*	*A*
Steve	*yes (1)*	*yes (1)*	*yes (1)*	*yes (1)*	*F*
Mary	*no (-1)*	*no (-1)*	*no (-1)*	*yes (1)*	*F*
Peter	*yes (1)*	*no (-1)*	*no (-1)*	*yes (1)*	*A*

- ***class 1***: students who get grade *A*
- ***class 2***: students who get grade *F*

LDF Example: Augment feature vector

	features					grade
name	***extra***	***good attendance?***	***tall?***	***sleeps in class?***	***chews gum?***	
Jane	*1*	*yes (1)*	*yes (1)*	*no (-1)*	*no (-1)*	*A*
Steve	*1*	*yes (1)*	*yes (1)*	*yes (1)*	*yes (1)*	*F*
Mary	*1*	*no (-1)*	*no (-1)*	*no (-1)*	*yes (1)*	*F*
Peter	*1*	*yes (1)*	*no (-1)*	*no (-1)*	*yes (1)*	*A*

- convert samples $\boldsymbol{x_1},\ldots,\boldsymbol{x_n}$ to augmented samples $\boldsymbol{y_1},\ldots,\boldsymbol{y_n}$ by adding a new dimension of value 1

LDF: Perform "Normalization"

	features					grade
name	***extra***	***good attendance?***	***tall?***	***sleeps in class?***	***chews gum?***	
Jane	*1*	*yes (1)*	*yes (1)*	*no (-1)*	*no (-1)*	*A*
Steve	*-1*	*yes (-1)*	*yes (-1)*	*yes (-1)*	*yes (-1)*	*F*
Mary	*-1*	*no (1)*	*no (1)*	*no (1)*	*yes (-1)*	*F*
Peter	*1*	*yes (1)*	*no (-1)*	*no (-1)*	*yes (1)*	*A*

- Replace all examples from class c_2 by their negative

$$y_i \rightarrow -y_i \qquad \forall y_i \in c_2$$

- Seek weight vector a s.t. $a^t y_i > 0 \qquad \forall y_i$

LDF: Use Single Sample Rule

	features					grade
name	***extra***	***good attendance?***	***tall?***	***sleeps in class?***	***chews gum?***	
Jane	*1*	*yes (1)*	*yes (1)*	*no (-1)*	*no (-1)*	*A*
Steve	*-1*	*yes (-1)*	*yes (-1)*	*yes (-1)*	*yes (-1)*	*F*
Mary	*-1*	*no (1)*	*no (1)*	*no (1)*	*yes (-1)*	*F*
Peter	*1*	*yes (1)*	*no (-1)*	*no (-1)*	*yes (1)*	*A*

- Sample is misclassified if $\boldsymbol{a}^t \boldsymbol{y}_i = \sum_{k=0}^{4} a_k y_i^{(k)} < 0$
- gradient descent single sample rule: $\boldsymbol{a}^{(k+1)} = \boldsymbol{a}^{(k)} + \eta^{(k)} \sum_{\boldsymbol{y} \in \boldsymbol{Y}_M} \boldsymbol{y}$
- Set ***fixed*** learning rate to $\eta^{(k)} = 1$: $\boxed{\boldsymbol{a}^{(k+1)} = \boldsymbol{a}^{(k)} + \boldsymbol{y}_M}$

LDF: Gradient decent Example

- set equal initial weights $\boldsymbol{a}^{(1)}$=[0.25, 0.25, 0.25, 0.25]
- visit all samples sequentially, modifying the weights for after finding a misclassified example

name	$\boldsymbol{a}^t\boldsymbol{y}$	*misclassified?*
Jane	*0.25*1+0.25*1+0.25*1+0.25*(-1)+0.25*(-1) >0*	*no*
Steve	*0.25*(-1)+0.25*(-1)+0.25*(-1)+0.25*(-1)+0.25*(-1)<0*	*yes*

- new weights

$$\boldsymbol{a}^{(2)} = \boldsymbol{a}^{(1)} + \boldsymbol{y}_M = [\mathbf{0.25}\ \mathbf{0.25}\ \mathbf{0.25}\ \mathbf{0.25}\ \mathbf{0.25}] +$$
$$+[-\mathbf{1}\ -\mathbf{1}\ -\mathbf{1}\ -\mathbf{1}\ -\mathbf{1}] =$$
$$=[-\mathbf{0.75}\ -\mathbf{0.75}\ -\mathbf{0.75}\ -\mathbf{0.75}\ -\mathbf{0.75}]$$

LDF: Gradient decent Example

$$\boldsymbol{a}^{(2)} = [-0.75 \;\; -0.75 \;\; -0.75 \;\; -0.75 \;\; -0.75]$$

name	$\boldsymbol{a}^t\boldsymbol{y}$	*misclassified?*
Mary	*-0.75*(-1)-0.75*1 -0.75 *1 -0.75 *1 -0.75*(-1) <0*	*yes*

- new weights

$$\boldsymbol{a}^{(3)} = \boldsymbol{a}^{(2)} + \boldsymbol{y}_M = [-0.75 \;\; -0.75 \;\; -0.75 \;\; -0.75 \;\; -0.75] +$$
$$+[-1 \quad 1 \quad 1 \;\; 1 \;\; -1] =$$
$$= [-1.75 \quad 0.25 \quad 0.25 \;\; 0.25 \;\; -1.75]$$

LDF: Gradient decent Example

$$\boldsymbol{a}^{(3)} = [-1.75 \quad 0.25 \quad 0.25 \quad 0.25 \quad -1.75]$$

name	$\boldsymbol{a}^t\boldsymbol{y}$	*misclassified?*
Peter	*-1.75 *1 +0.25* 1+0.25* (-1) +0.25 *(-1)-1.75*1 <0*	*yes*

- new weights

$$\boldsymbol{a}^{(4)} = \boldsymbol{a}^{(3)} + \boldsymbol{y}_M = [-1.75 \quad 0.25 \quad 0.25 \quad 0.25 \quad -1.75] +$$

$$+ [1 \quad 1 \quad -1 \quad -1 \quad 1] =$$

$$= [-0.75 \quad 1.25 \quad -0.75 \quad -0.75 \quad -0.75]$$

LDF: Gradient decent Example

$$\boldsymbol{a}^{(4)} = [-0.75 \;\; 1.25 \;\; -0.75 \;\; -0.75 \;\; -0.75]$$

name	$\boldsymbol{a}^t\boldsymbol{y}$	*misclassified?*
Jane	*-0.75 *1 +1.25*1 -0.75*1 -0.75 *(-1) -0.75 *(-1)+0*	*no*
Steve	*-0.75*(-1)+1.25*(-1) -0.75*(-1) -0.75*(-1)-0.75*(-1)>0*	*no*
Mary	*-0.75 *(-1)+1.25*1-0.75*1 -0.75 *1 –0.75*(-1) >0*	*no*
Peter	*-0.75 *1+ 1.25*1-0.75* (-1)-0.75* (-1) -0.75 *1 >0*	*no*

- Thus the discriminant function is

$$g(\boldsymbol{y}) = -0.75 * \boldsymbol{y}^{(0)} + 1.25 * \boldsymbol{y}^{(1)} - 0.75 * \boldsymbol{y}^{(2)} - 0.75 * \boldsymbol{y}^{(3)} - 0.75 * \boldsymbol{y}^{(4)}$$

- Converting back to the original features $\boldsymbol{x}$:

$$g(\boldsymbol{x}) = 1.25 * \boldsymbol{x}^{(1)} - 0.75 * \boldsymbol{x}^{(2)} - 0.75 * \boldsymbol{x}^{(3)} - 0.75 * \boldsymbol{x}^{(4)} - 0.75$$

LDF: Gradient decent Example

- Converting back to the original features $\boldsymbol{x}$:

$$1.25 * x^{(1)} - 0.75 * x^{(2)} - 0.75 * x^{(3)} - 0.75 * x^{(4)} > 0.75 \Rightarrow \textit{grade A}$$

$$1.25 * x^{(1)} - 0.75 * x^{(2)} - 0.75 * x^{(3)} - 0.75 * x^{(4)} < 0.75 \Rightarrow \textit{grade F}$$

- This is just one possible solution vector
- If we started with weights $\boldsymbol{a}^{(1)}$=[0,0.5, 0.5, 0, 0], solution would be [-1,1.5, -0.5, -1, -1]

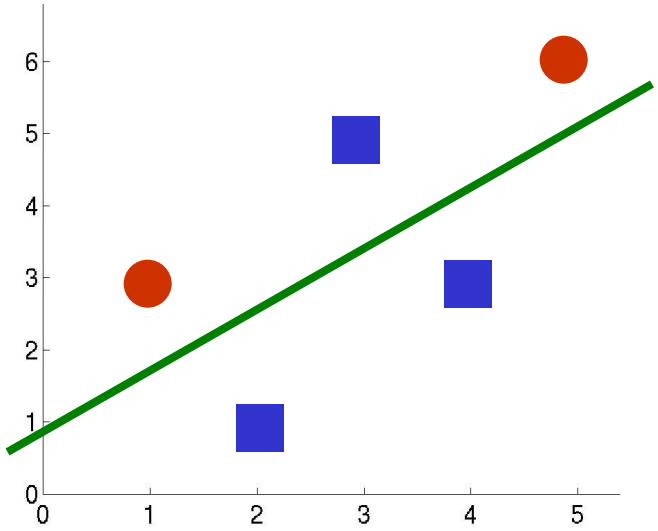
$$1.5 * x^{(1)} - 0.5 * x^{(2)} - x^{(3)} - x^{(4)} > 1 \Rightarrow \textit{grade A}$$

$$1.5 * x^{(1)} - 0.5 * x^{(2)} - x^{(3)} - x^{(4)} < 1 \Rightarrow \textit{grade F}$$

  - In this solution, being tall is the least important feature

LDF: Nonseparable Example

- Suppose we have 2 features and samples are:
  - Class 1: [2,1], [4,3], [3,5]
  - Class 2: [1,3] and [5,6]
- These samples are not separable by a line



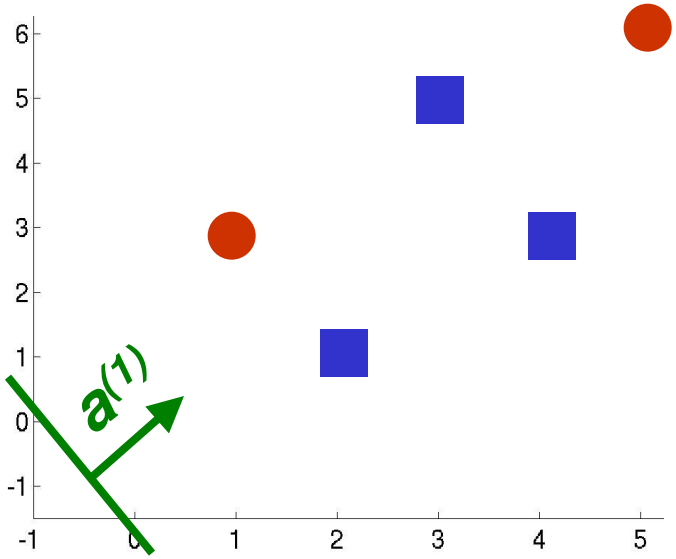
- Still would like to get approximate separation by a line, good choice is shown in green
  - some samples may be "noisy", and it's ok if they are on the wrong side of the line
- Get y_1, y_2, y_3, y_4 by adding extra feature and "normalizing"

$$y_1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad y_2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \quad y_3 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad y_4 = \begin{bmatrix} -1 \\ -1 \\ -3 \end{bmatrix} \quad y_5 = \begin{bmatrix} -1 \\ -5 \\ -6 \end{bmatrix}$$

LDF: Nonseparable Example

- Let's apply Perceptron single sample algorithm
- initial equal weights $\boldsymbol{a}^{(1)} = [1\ 1\ 1]$
  - this is line $x^{(1)}+x^{(2)}+1=0$
- fixed learning rate $\eta = 1$

$$\boldsymbol{a}^{(k+1)} = \boldsymbol{a}^{(k)} + \boldsymbol{y}_M$$



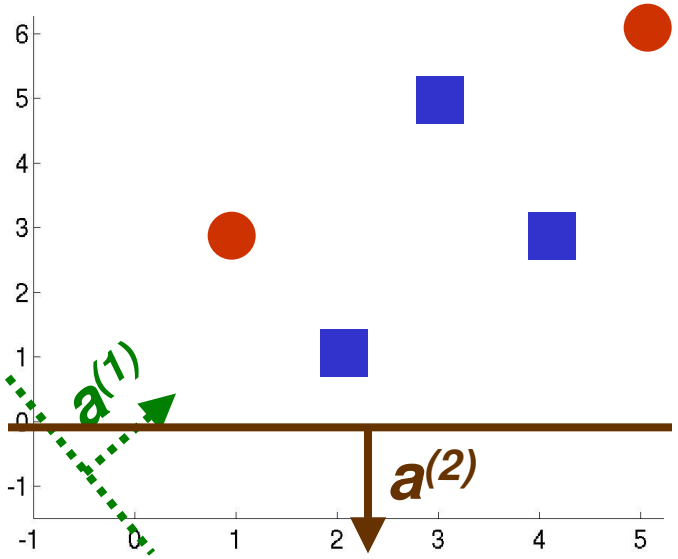
$$\boldsymbol{y}_1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad \boldsymbol{y}_2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \quad \boldsymbol{y}_3 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad \boldsymbol{y}_4 = \begin{bmatrix} -1 \\ -1 \\ -3 \end{bmatrix} \quad \boldsymbol{y}_5 = \begin{bmatrix} -1 \\ -5 \\ -6 \end{bmatrix}$$

- $\boldsymbol{y}^t_1\boldsymbol{a}^{(1)} = [1\ 1\ 1]*[1\ 2\ 1]^t > 0$ ✔
- $\boldsymbol{y}^t_2\boldsymbol{a}^{(1)} = [1\ 1\ 1]*[1\ 4\ 3]^t > 0$ ✔
- $\boldsymbol{y}^t_3\boldsymbol{a}^{(1)} = [1\ 1\ 1]*[1\ 3\ 5]^t > 0$ ✔

LDF: Nonseparable Example

$$a^{(1)} = [1\ 1\ 1] \qquad a^{(k+1)} = a^{(k)} + y_M$$

$$y_1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad y_2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \quad y_3 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad y_4 = \begin{bmatrix} -1 \\ -1 \\ -3 \end{bmatrix} \quad y_5 = \begin{bmatrix} -1 \\ -5 \\ -6 \end{bmatrix}$$



- $y^t_4 a^{(1)} = [1\ 1\ 1]*[-1\ -1\ -3]^t = -5 < 0$

$$a^{(2)} = a^{(1)} + y_M = [1\ 1\ 1] + [-1\ -1\ -3] = [0\ 0\ -2]$$

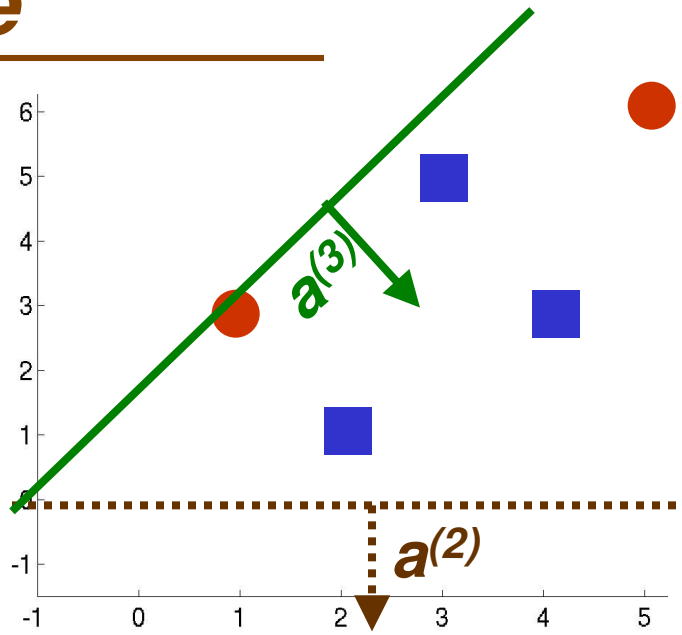
- $y^t_5\, a^{(2)} = [0\ 0\ -2]*[-1\ -5\ -6]^t = 12 > 0$ ✔
- $y^t_1\, a^{(2)} = [0\ 0\ -2]*[1\ 2\ 1]^t < 0$

$$a^{(3)} = a^{(2)} + y_M = [0\ 0\ -2] + [1\ 2\ 1] = [1\ 2\ -1]$$

LDF: Nonseparable Example

$$a^{(3)} = [1\ 2\ -1] \qquad a^{(k+1)} = a^{(k)} + y_M$$

$$y_1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad y_2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \quad y_3 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad y_4 = \begin{bmatrix} -1 \\ -1 \\ -3 \end{bmatrix} \quad y_5 = \begin{bmatrix} -1 \\ -5 \\ -6 \end{bmatrix}$$



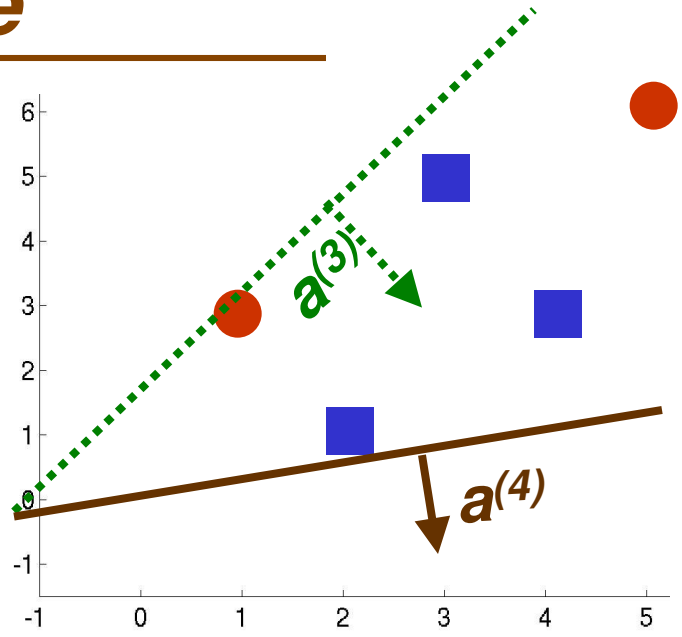
- $y^t_2\, a^{(3)} = [1\ 4\ 3]*[1\ 2\ -1]^t = 6 > 0$ ✔
- $y^t_3\, a^{(3)} = [1\ 3\ 5]*[1\ 2\ -1]^t > 0$ ✔
- $y^t_4\, a^{(3)} = [-1\ -1\ -3]*[1\ 2\ -1]^t = 0$

$$a^{(4)} = a^{(3)} + y_M = [1\ 2\ -1] + [-1\ -1\ -3] = [0\ 1\ -4]$$

LDF: Nonseparable Example

$$a^{(4)} = [0\ 1\ -4] \qquad a^{(k+1)} = a^{(k)} + y_M$$

$$y_1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad y_2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \quad y_3 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad y_4 = \begin{bmatrix} -1 \\ -1 \\ -3 \end{bmatrix} \quad y_5 = \begin{bmatrix} -1 \\ -5 \\ -6 \end{bmatrix}$$



- $y^t_2\, a^{(3)} = [1\ 4\ 3]*[1\ 2\ -1]^t = 6 > 0$ ✔
- $y^t_3\, a^{(3)} = [1\ 3\ 5]*[1\ 2\ -1]^t > 0$ ✔
- $y^t_4\, a^{(3)} = [-1\ -1\ -3]*[1\ 2\ -1]^t = 0$

$$a^{(4)} = a^{(3)} + y_M = [1\ 2\ -1] + [-1\ -1\ -3] = [0\ 1\ -4]$$

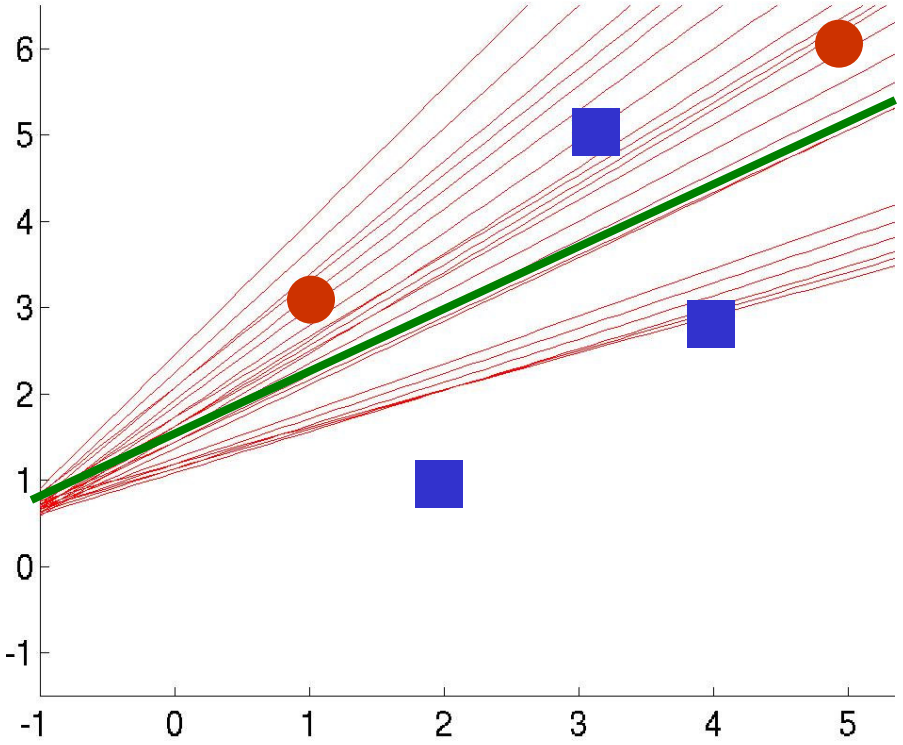
LDF: Nonseparable Example

- we can continue this forever
  - there is no solution vector ***a*** satisfying for all ***i***

$$a^t y_i = \sum_{k=0}^{5} a_k y_i^{(k)} > 0$$

- need to stop but at a good point:

- solutions at iterations 900 through 915. Some are good some are not.
- How do we stop at a good solution?



LDF: Convergence of Perceptron rules

- If classes are linearly separable, and use fixed learning rate, that is for some constant $\boldsymbol{c}$, $\eta^{(k)} = \boldsymbol{c}$
  - *both single sample and batch perceptron rules converge to a correct solution* (could be any $\boldsymbol{a}$ in the solution space)
- If classes are not linearly separable:
  - algorithm does not stop, it keeps looking for solution which does not exist
  - by choosing appropriate learning rate, can always ensure convergence: $\eta^{(k)} \rightarrow 0$ ***as*** $k \rightarrow \infty$
  - for example inverse linear learning rate: $\eta^{(k)} = \frac{\eta^{(1)}}{k}$
  - for inverse linear learning rate convergence in the linearly separable case can also be proven
  - no guarantee that we stopped at a good point, but there are good reasons to choose inverse linear learning rate

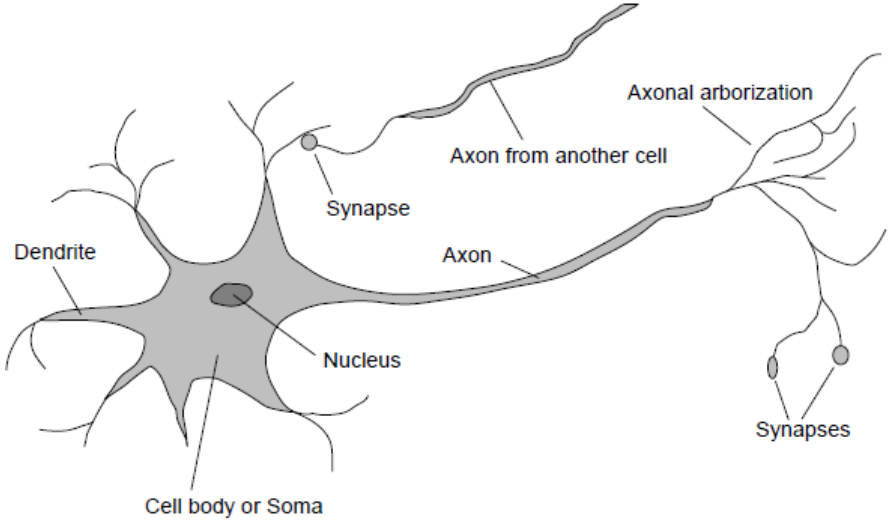
LDF: Perceptron Rule and Gradient decent

- Linearly separable data
  - perceptron rule with gradient decent works well
- Linearly non-separable data
  - need to stop perceptron rule algorithm at a good point, this maybe tricky

Batch Rule	*Single Sample Rule*
▪ Smoother gradient because all samples are used	▪ easier to analyze ▪ Concentrates more than necessary on any isolated "noisy" training examples

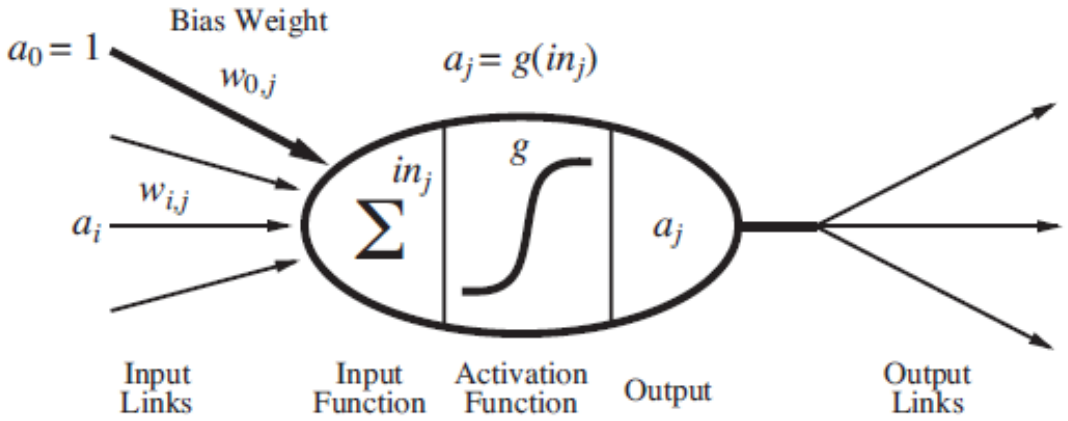
Brain, Neuron, Neural Network

- The Brain
- Mathematical models of the brain's activity
- Neuroscience hypothesis - Mental activity consists of electrochemical activity in networks of brain cells called neurons
- Inspired by this, earliest **AI** work to create **artificial neural networks**
- Other names for the field: *connectionism, parallel distributed processing and neural computation*



Mathematical model of a neuron

- A simple model by McCulloch and Pitts
- Roughly, it “fires” when a linear combination of its inputs exceeds some (hard or soft) threshold
- In another way, implements a linear classifier
- Neural Network (NN) – a collection of units connected together
- Properties determined by – its topology and the properties of the “neurons”



Neural Network Structures – *Single Neuron*

- Composed of *nodes or units* connected by directed links
- A link from unit i to unit j serves to propagates the **activation** $\mathbf{a_i}$ from i to j
- Each link also has a numeric **weight** $\mathbf{w_{i,j}}$ associated with it – determines the *strength* and *sign of the connection*
- Each unit has a dummy input $a_0=1$, with and associated weight $w_{0,j}$
- Each unit j computes a weighted sum of its inputs:

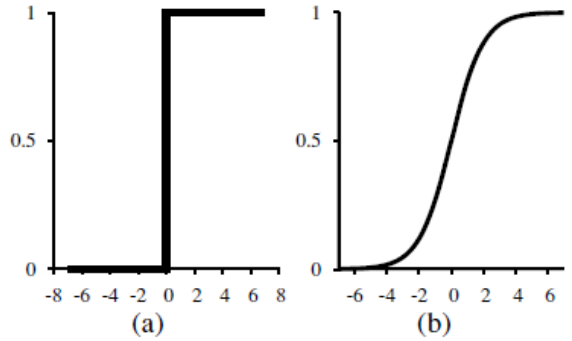
$$in_j = \sum_{i=0}^{n} w_{i,j} a_i$$

Neural Network Structures – *Single Neuron*

- Then it applies an **activation function g** to this sum to derive the output

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right)$$

- The activation function g is typically either a **hard threshold** (Fig. (a)) in which case the unit is called a **perceptron**
- or a **logistic function** (Fig. (b)), in which case the term **sigmoid perceptron** is sometimes used
- Both of these *nonlinear activation functions* ensure the important property that the entire network of units can represent **a nonlinear function**



Neural Network Structures – *Network of Neurons*

- How to connect neurons to form a network?
- Two fundamentally distinct ways: **feed-forward network** and **recurrent network**
- **Feed-forward network:** has only connections in one direction; it forms a directed acyclic graph
- Every node receives input from *"upstream"* nodes and delivers output to *"downstream"* nodes; there are *no loops*
- A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves

Neural Network Structures – *Network of Neurons*

- **recurrent network:** feeds its outputs back into its own inputs
- the response of the network to a given input depends on its initial state, which may depend on previous inputs
- Recurrent networks can support short-term memory
- More interesting as models of the brain, but also more difficult to understand
- Applications: Text processing, Machine translation, Speech recognition, Time series prediction, Handwriting recognition, Robot control and so on .

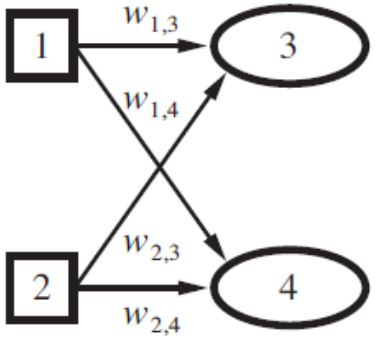
Neural Network Structures – *Network of Neurons*

- Feed-forward networks are usually arranged in **layers, such that each unit receives input** only from units in the immediately preceding layer
- Single layer network – Every unit connects directly from the network's inputs to its outputs
- Multilayer networks - have one or more layers of **hidden units that** are not connected to the outputs of the network
- Neural networks are often used in cases where multiple outputs rather than learning problems with a single output variable y
- Eg., (1) Add two input bits – Sum and Carry as outputs

  (2) learning to categorize images of handwritten digits (0-9)

Single-layer feed-forward neural networks (perceptrons)

- A network with all the inputs connected directly to the outputs
- A simple two-input, two-output perceptron network



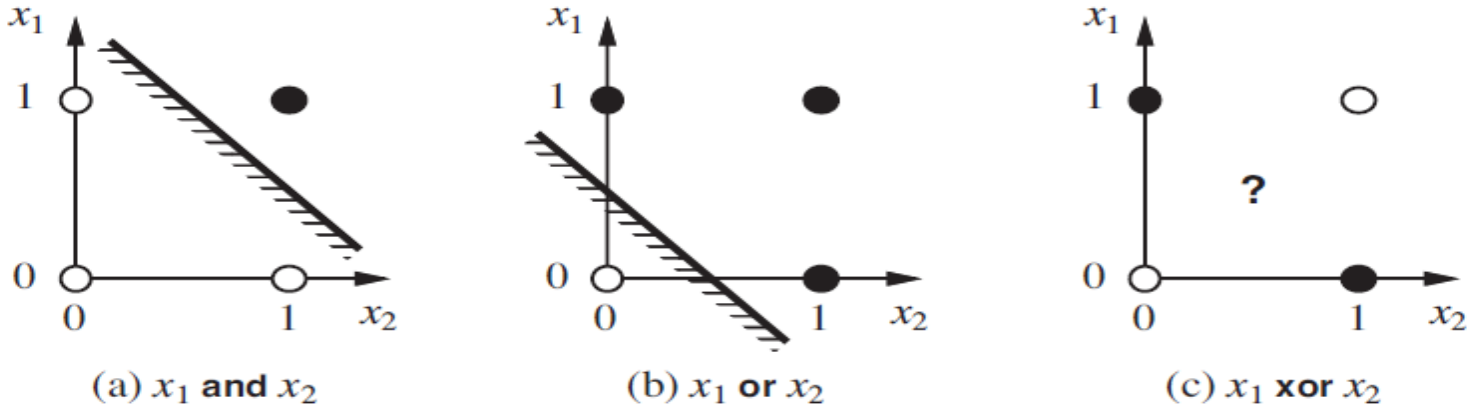
- We might hope to learn the two-bit adder function

Training data

x_1	x_2	y_3 (carry)	y_4 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

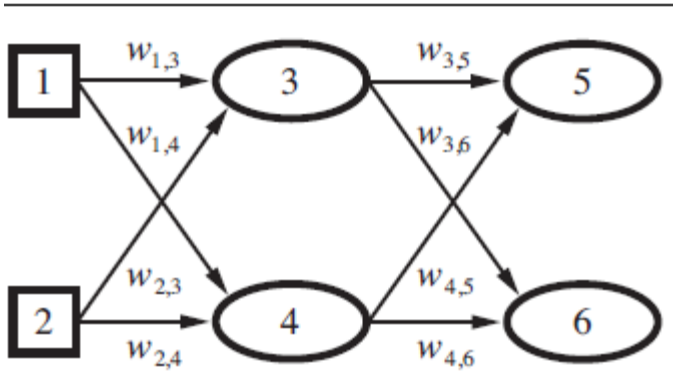
Single-layer feed-forward neural networks (perceptrons)

- A perceptron with m outputs is essentially m separate networks; so there is m separate training processes
- Depending upon the activation function: (1) perceptron learning rule (2) gradient descent rule [for logistic regression]
- If we apply either method, in the case of two-bit adder, only the carry function can be learned; never the sum function!
- The sum function is XOR of the two input bits, which is not a linearly separable, **hence perceptron can not train it**.



Multilayer feed-forward neural networks

- Assume the network as a function **$h_w(x)$ parameterized by the weights w**
- Two input units, two hidden units, and two output units (In addition, each unit has a dummy input fixed at 1.)
- Given input vector **x** = (x1, x2); so activations of input units are set as (a1, a2) = (x1, x2).



Multilayer feed-forward neural networks

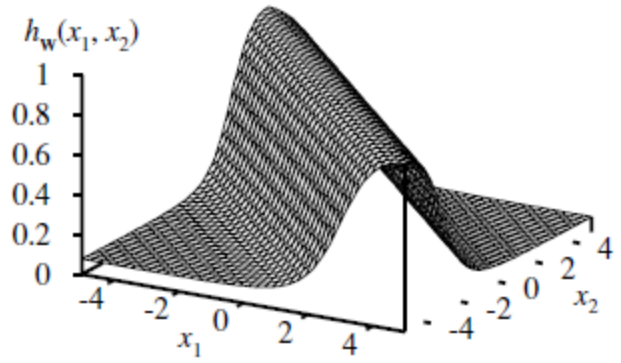
- The output at unit 5 is

$$\begin{aligned} a_5 &= g(w_{0,5,+} w_{3,5}\, a_3 + w_{4,5}\, a_4) \\ &= g(w_{0,5,+} w_{3,5}\, g(w_{0,3} + w_{1,3}\, a_1 + w_{2,3}\, a_2) + w_{4,5}\, g(w_0 4 + w_{1,4}\, a_1 + w_{2,4}\, a_2)) \\ &= g(w_{0,5,+} w_{3,5}\, g(w_{0,3} + w_{1,3}\, x_1 + w_{2,3}\, x_2) + w_{4,5}\, g(w_0 4 + w_{1,4}\, x_1 + w_{2,4}\, x_2)). \end{aligned}$$

- Thus the output expressed as a function of the inputs and the weights.
- As long as it is derivable, we can apply the **gradient-descent loss-minimization method** to train the network
- The function represented by the network can be highly nonlinear
- Nested nonlinear soft threshold functions
- Neural networks as a tool for doing **nonlinear regression**

Multilayer feed-forward neural networks

- Each unit in a sigmoid network represents a soft threshold in its input space
- With one hidden layer and one output layer: each output unit computes a soft-thresholded linear combination of several such functions
- By adding two opposite-facing soft threshold functions and thresholding the result, we can obtain a *"ridge"* function like this



Multilayer feed-forward neural networks

- In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.

- Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

Learning in multilayer networks

- Assume input as well as output as vectors
- We can not decompose into m separate learning problems! both a5 and a6 depend on all of the input-layer weights, so updates to those weights will depend on errors in both a5 and a6
- But it is possible if we add up the gradient contributions from each of the m-outputs when updating the weights: **$y-h_w(x)$** is the error vector; k number of output nodes

$$\frac{\partial}{\partial w} Loss(\mathbf{w}) = \frac{\partial}{\partial w} |\mathbf{y} - \mathbf{h}_\mathbf{w}(\mathbf{x})|^2 = \frac{\partial}{\partial w} \sum_k (y_k - a_k)^2 = \sum_k \frac{\partial}{\partial w} (y_k - a_k)^2$$

Learning in multilayer networks

- Whereas the error $y - h_w$ at the output layer is clear, **the error at the hidden layers seems** mysterious because the training data do not say what value the hidden nodes should have
- Fortunately, it turns out that we can **back-propagate the error from the output layer to the** hidden layers.
- The back-propagation process emerges directly from a derivation of the overall error gradient
- At the output layer: let Err_k be the k^{th} component of the error vector $\mathbf{y - h_w}$. Define a modified error $\Delta_k = Err_k \times g'(in_k)$, so that the weight update rule becomes:

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k$$

Learning in multilayer networks

- To update the connections between the input units and the hidden units: we do the **error backpropagation**

- The idea is that hidden node j is *"responsible"* for some fraction of the error Δ_k in each of the output nodes to which it connects.

- Thus, the Δ_k values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer.

$$\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k$$

Learning in multilayer networks

- The weight-update rule for the weights between the inputs and the hidden layer is essentially identical to the update rule for the output layer

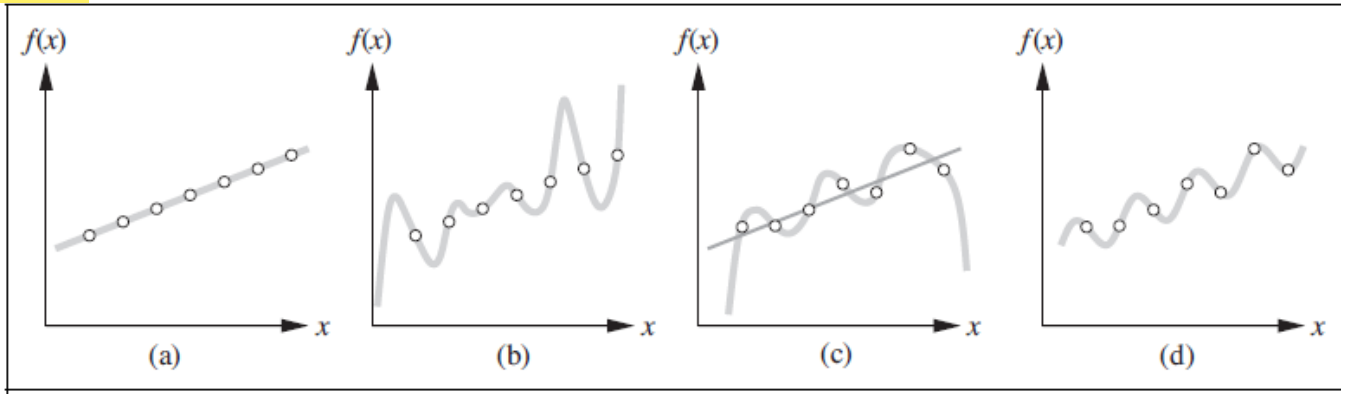
$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j$$

• The back-propagation process can be summarized as follows:

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
  - Propagate the Δ values back to the previous layer.
  - Update the weights between the two layers.

Learning the neural network structures

- So far we have considered learning the weights, given a fixed network structure.
- Now how to find the best network structure?
- If a network that is too big, it will be able to memorize all the examples; but **will not generalize** well to inputs that have not seen before!
- In other words, like all statistical models, neural networks are subject to **overfitting** when there are too many parameters in the model



Learning the neural network structures

- **If we stick to fully connected network** – the only choices are (1) number of hidden layers and (2) their sizes.
- The usual approach is to try several and keep the best (cross validation techniques)
- We choose the network architecture that gives the highest prediction accuracy on the validation sets.

- **If not fully connected network -** some effective search method through the very large space of possible connection topologies

Learning the neural network structures

- The **optimal brain damage algorithm** begins with a fully connected network and removes connections from it
- After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped
- The network is then retrained, and if its performance has not decreased then the process is repeated

- In addition to removing connections, it is also possible to remove units that are not contributing much to the result

Learning the neural network structures

- Another approach: growing a larger network from a smaller one
- The **tiling algorithm** - the idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible.
- Subsequent units are added to take care of the examples that the first unit got wrong.
- The algorithm adds only as many units as are needed to cover all the examples

MULTILAYER NEURAL NETWORKS

Jeff Robble, Brian Renzenbrink, Doug Roberts

Multilayer Neural Networks

We learned in Chapter 5 that clever choices of nonlinear φ functions we can obtain arbitrary decision boundaries that lead to minimum error.

Identifying the appropriate nonlinear functions to be used can be difficult and incredibly expensive.

We need a way to *learn* the non-linearity at the same time as the linear discriminant.

Multilayer Neural Networks, in principle, do exactly this in order to provide the optimal solution to arbitrary classification problems.

Multilayer Neural Networks implement linear discriminants in a space where the inputs have been mapped non-linearly.

The form of the non-linearity can be learned from simple algorithms on training data.

Note that neural networks require continuous functions to allow for gradient descent

Multilayer Neural Networks

Training multilayer neural networks can involve a number of different algorithms, but the most popular is *the back propagation algorithm* or generalized delta rule.

Back propagation is a natural extension of the LMS algorithm.
The back propagation method is simple for models of arbitrary complexity.
This makes the method very flexible.

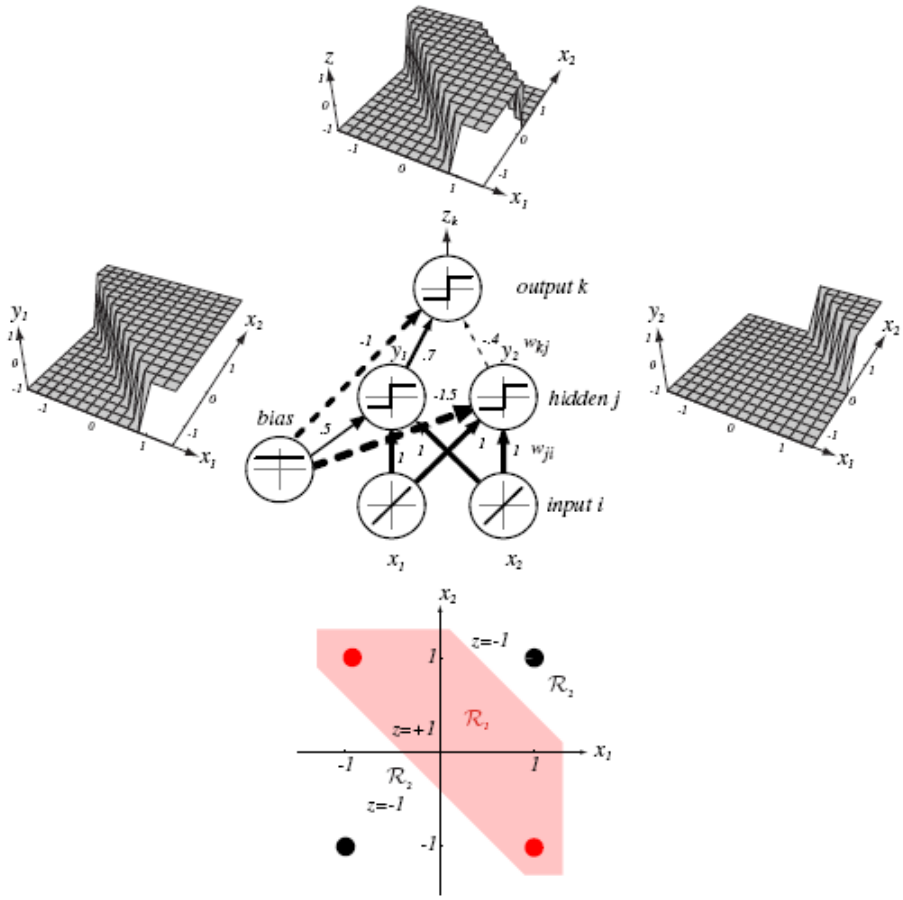
One of the largest difficulties with developing neural networks is regularization, or adjusting the complexity of the network.

Too many parameters = poor generalization

Too few parameters = poor learning

Example: Exclusive OR (XOR)

x – The feature vector
y – The vector of hidden layer outputs
k – The output of the network
w_{ji} – weight of connection between input unit i and hidden unit j
w_{kj} – weight of connection between input hidden unit j and output unit k
bias – A numerical bias used to make calculation easier



Example: Exclusive OR (XOR)

XOR is a Boolean function that is true for two variables if and only if one of the variables is true and the other is false.

This classification can not be solved with linear separation, but is very easy for a neural network to generate a non-linear solution to.

The hidden unit computing y_1 acts like a two-layer Perceptron.

It computes the boundary $x_1 + x_2 + bias_1 = 0$. If $x_1 + x_2 + bias_1 > 0$, then the hidden unit sets $y_1 = 1$, otherwise y_1 is set equal to –1. Analogous to the OR function.

The other hidden unit computes the boundary for $x_1 + x_2 + bias_2 = 0$, setting $y_2 = 1$ if $x_1 + x_2 + bias_2 > 0$. Analogous to the negation of the AND function.

The final output node emits a positive value if and only if both y_1 and y_2 equal 1.

Note that the symbols within the nodes graph the nodes activation function.

This is a 2-2-1 fully connected topology.

Feedforward Operation and Classification

Figure 6.1 is an example of a simple three layer neural network

The neural network consists of:

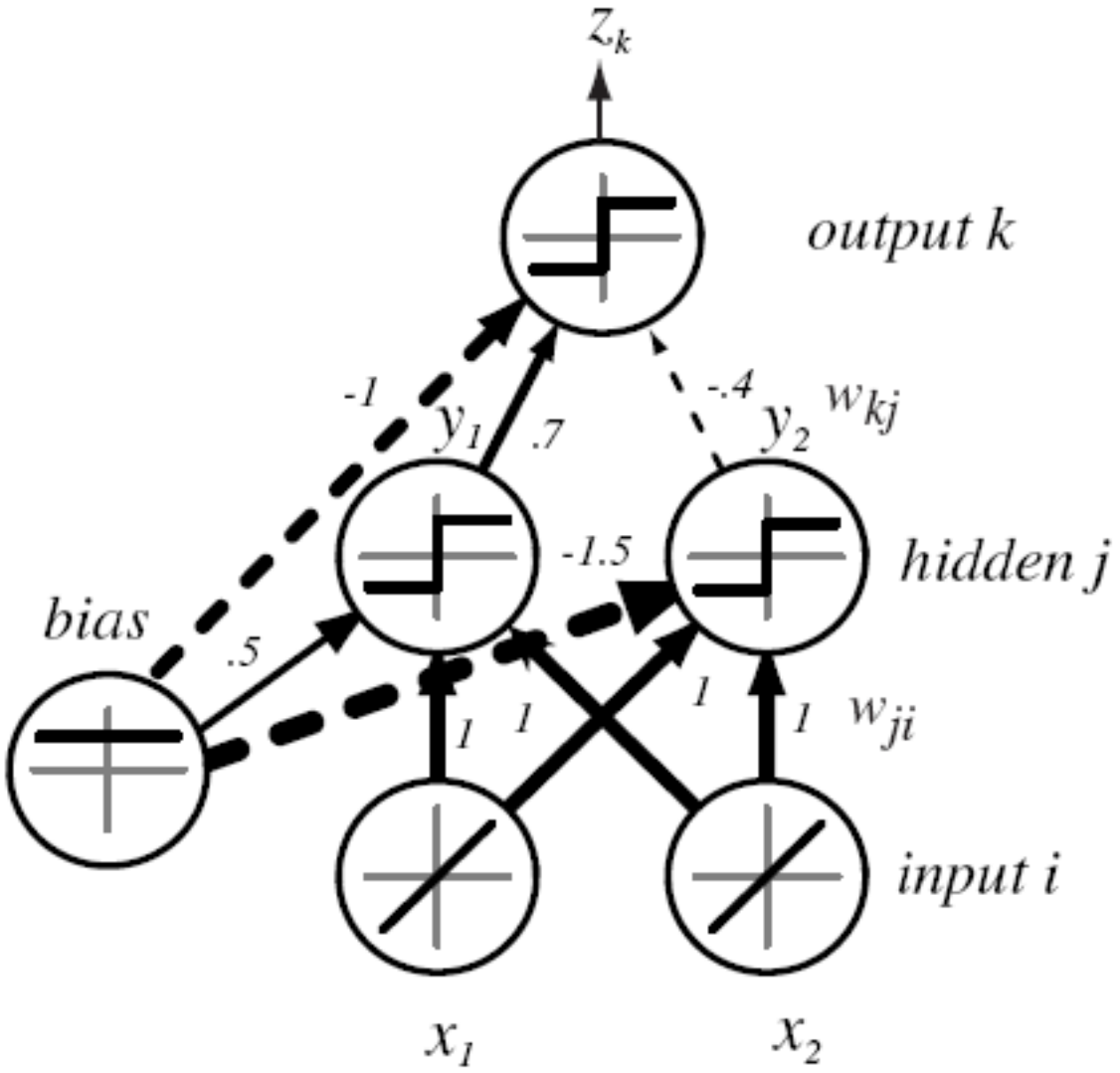
- An input layer
- A hidden layer
- An output layer

Each of the layers are interconnected by modifiable weights, which are represented by the links between layers

Each layer consists of a number of units (neurons) that loosely mimic the properties of biological neurons

The hidden layers are mappings from one space to another. The goal is to map to a space where the problem is linearly separable.

Feedforward Operation and Classification



Activation Function

The input layer takes a two dimensional vector as input.

The output of each input unit equals the corresponding component in the vector.

Each unit of the hidden layer computes the weighted sum of its inputs in order to form a a scalar net activation (net) which is the inner product of the inputs with the weights at the hidden layer.

Activation Function

$$net_j = \sum_{i=0}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv \mathbf{w}_j^t \mathbf{x} \qquad (1)$$

Where

- *i* indexes units in the input layer
- *j* indexes units in the hidden layer
- w_{ij} denotes the input-to-hidden layer weights at the hidden unit *j*

Activation Function

Each hidden unit emits an output that is a nonlinear function of its activation, f(net), that is:

$$y_j = f(net_j) \quad (2)$$

One possible activation function is simply the sign function:

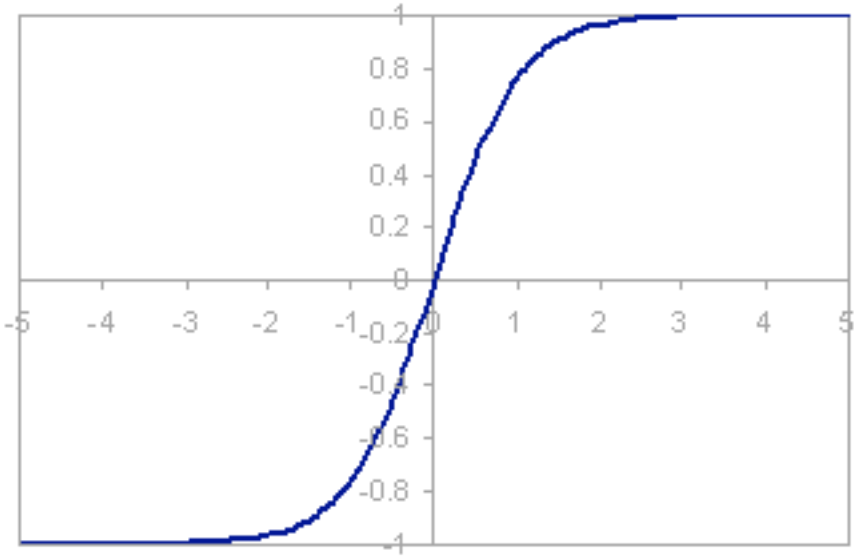
$$f(net) = Sgn(net) \equiv \begin{cases} 1 & if \quad net \geq 0 \\ -1 & if \quad net \leq 0 \end{cases} \qquad (3)$$

The activation function represents the *nonlinearity* of a unit.

The activation function is sometimes referred to as a *sigmoid* function, a *squashing* function, since its primary purpose is to limit the output of the neuron to some reasonable range like a range of -1 to +1, and thereby inject some degree of non-linearity into the network.

Activation Function

LeCun suggests the hyperbolic tangent function as a good activation function. *tanh* is completely symmetric:



Because *tanh* is asymptotic when producing outputs of ±1, the network should be trained for an intermediate value such as ±.8.

Also, the derivative of *tanh* is simply $1\text{-}tanh^2$, so if

$f(net) = tanh(net)$, then the derivative is simply $1\text{-}tanh^2(net)$.

When we discuss the update rule you will see why an activation function with an easy-to-compute derivative is desirable.

Activation Function

Each output unit similarly computes its net activation based on the hidden unit signals as:

$$net_k = \sum_{j=1}^{n_H} x_j w_{kj} + w_{k0} = \sum_{j=1}^{n_H} x_j w_{kj} \equiv \mathbf{w}_k^t \mathbf{y} \qquad (4)$$

$$net_j = \sum_{i=0}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv \mathbf{w}_j^t \mathbf{x} \qquad (1)$$

Where:

- k indexes units in the output layer
- n_h denotes the number of hidden units

Equation 4 is basically the same as Equation 1. The only difference is are the indexes.

Activation Function

An output unit computes the nonlinear function of its *net*

$$z_k = f(net_k) \quad (5)$$

The output z_k can be thought of as a function of the input feature vector **x**

If there are c output units, we think of the network as computing c discriminant functions $z_k = g_k(\boldsymbol{x})$

Inputs can be classified according to which discriminant function is determined to be the largest

General Feedforward Operation

Given a sufficient number of hidden units of a general type any function can be represented by the hidden layer

This also applies to:

- More inputs
- Other nonlinearities
- Any number of output units

Equations 1, 2, 4, and 5 can be combined to express the discriminant function $g_k(\boldsymbol{x})$:

$$g_k(\mathbf{x}) \equiv z_k = f\left(\sum_{j=1}^{n_H} w_{kj} f\left(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}\right) + w_{k0}\right) \qquad (7)$$

Expressive Power

Any continuous function from input to output can be implemented in a three layer network given

- A sufficient number of of hidden units nH
- Proper nonlinearities
- Weights

Kolmogorov proved that any continuous function $g(\mathbf{x})$ defined on the hypercube I^n ($I = [0,1]$ and $n \geq 2$) can be represented in the form

$$g(\mathbf{x}) = \sum_{j=1}^{2n+1} \Xi_j \left(\sum_{i=1}^{d} \Psi_{ij}(x_i) \right) \qquad (8)$$

Expressive Power

Equation 8 can be expressed in neural network terminology as follows:

- Each of the $2n + 1$ hidden units takes as input a sum of d nonlinear functions, one for each input feature x_i
- Each hidden unit emits a nonlinear function Ξ of its total input
- The output unit emits the sum of the contributions of the hidden units

Expressive Power

Figure 6.2 represents a 2-4-1 network, with a bias.

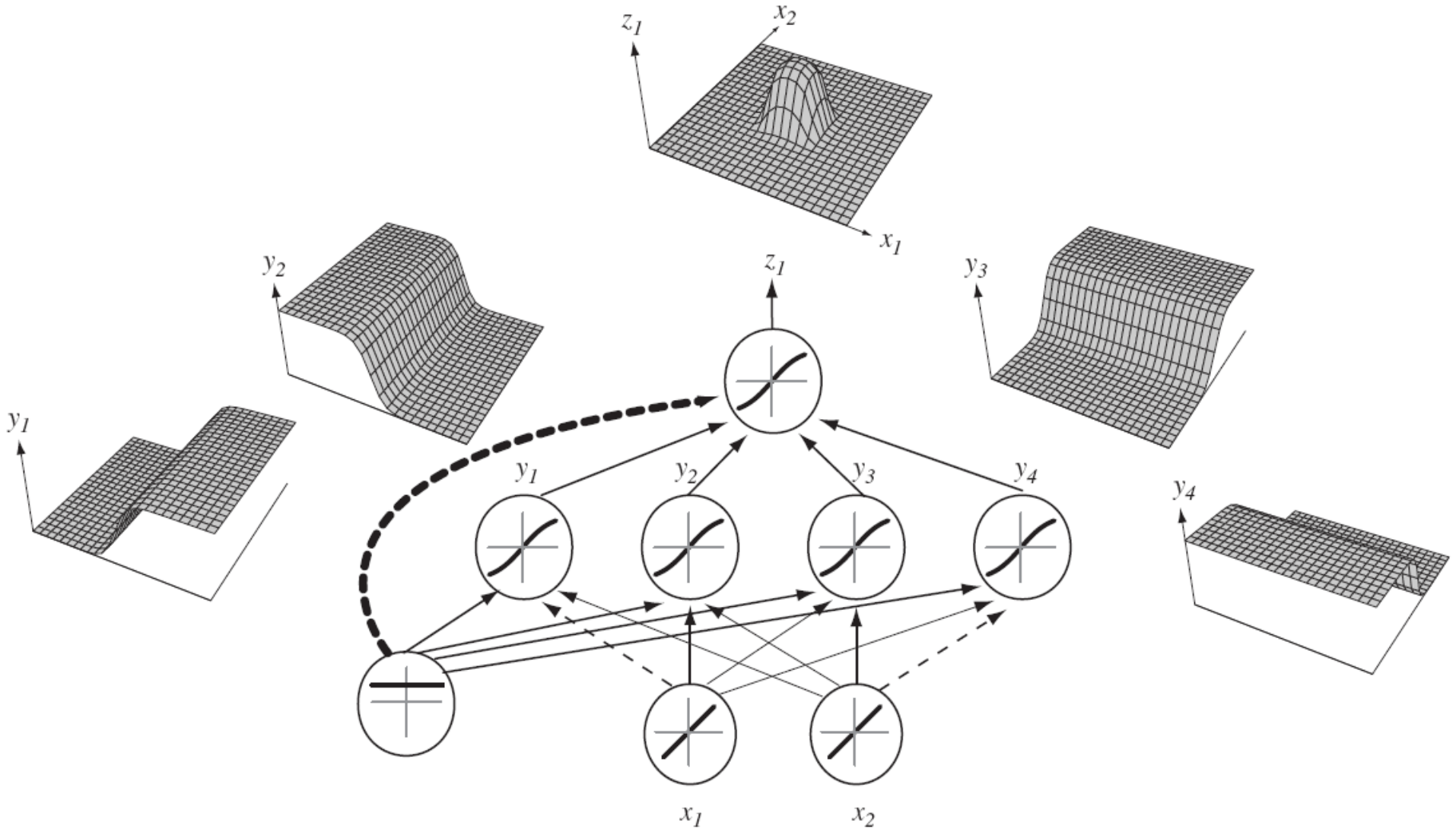
Each hidden output unit has a sigmoidal activation function $f(\cdot)$

The hidden output units are paired in opposition and

thus produce a “bump” at the output unit.

Given a sufficiently large number of hidden units, any continuous function from input to output can be approximated arbitrarily well by such a network.

Expressive Power



Backpropagation Learning Rule

"Backpropagation of Errors" – during training an error must be propagated from the output layer back to the hidden layer in order learn *input-to-hidden weights*

Credit Assignment Problem – there is no explicit teacher to tell us what the hidden unit's output should be

Backpropagation is used for the *supervised learning* of networks.

Modes of Operation

Networks have 2 primary modes of operation:

- Feed-forward Operation – present a pattern to the input units and pass signals through the network to yield outputs from the output units (ex. XOR network)

- Supervised Learning – present an input pattern and change the network parameters to bring the actual outputs closer to desired *target values*

3-Layer Neural Network Notation

d – dimensions of input pattern **x**

x_i – signal emitted by input unit i

(ex. pixel value for an input image)

n_H – number of hidden units

f() – nonlinear activation function

w_{ji} – weight of connection between input unit i and hidden unit j

net_j – inner product of input signals with weights w_{ij} at the hidden unit j

y_j – signal emitted by hidden unit j,

$y_j = f(net_j)$

w_{kj} – weight of connection between input hidden unit j and output unit k

net_k – inner product of hidden signals with weights w_{kj} at the output unit k

z_k – signal emitted by output unit k

(one for each classifier), $z_k = f(net_k)$

t – target vector that output signals **z** are compared with to find differences between actual and desired values

c – number of classes (size of **t** and **z**)

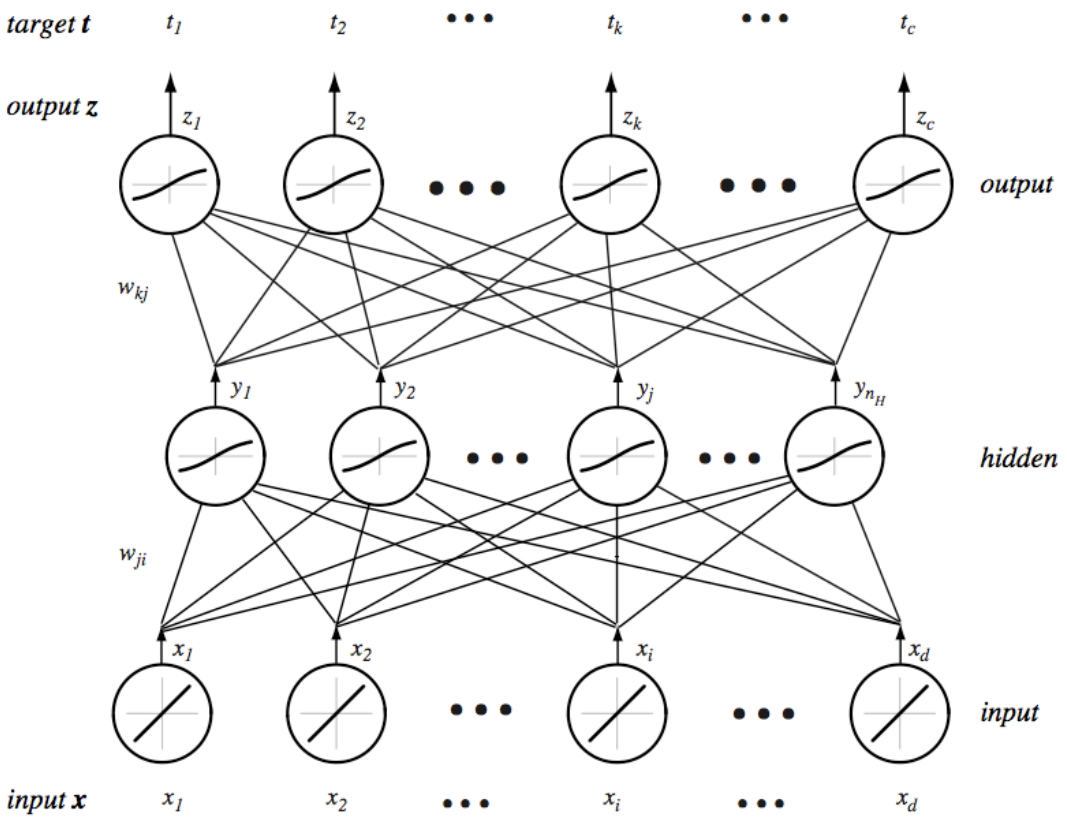


FIGURE 6.4. A d-n_H-c fully connected three-layer network

Training Error

Using **t** and **z** we can determine the training error of a given pattern using the following criterion function:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 = \frac{1}{2}\|\mathbf{t} - \mathbf{z}\|^2 \quad (9)$$

Which gives us half the sum of the squared difference between the desired output t_k and the actual output z_k over all outputs. This is based on the vector of current weights in the network, **w**.

Looks very similar to the the Minimum Squared Error criterion function:

$$J_s(\mathbf{a}) = \sum_{i=1}^{n}(\mathbf{a}^t\mathbf{y}_i - b_i)^2 = \|\mathbf{Ya} - \mathbf{b}\|^2 \quad (44)$$

Y– n-by- $\hat{d}$ matrix of x space feature points to y space feature points in $\hat{d}$ dimensions

a – $\hat{d}$ dimensional weight vector

b – margin vector

Update Rule: Hidden Units to Output Units

Weights are initialized with random values and changed in a direction that will reduce the error:

$$\Delta \mathbf{w} = -\eta \frac{\partial J}{\partial \mathbf{w}} \quad (10) \qquad \Delta w_{pq} = -\eta \frac{\partial J}{\partial w_{pq}} \quad (11)$$

η – learning rate that controls the relative size of the change in weights

The weight vector is updated per iteration m, as follows:

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta \mathbf{w}(m) \quad (12)$$

Let's evaluate Δ**w** for a 3-layer network for the output weights w_{kj}:

$$net_k = \sum_{j=0}^{n_H} y_j w_{kj} \quad (4)$$

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} \frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}} \quad (13)$$

(y_j)

Sensitivity of unit k describes how the overall error changes with respect to the unit's net activation:

$$\delta_k = -\frac{\partial J}{\partial net_k} \quad (14)$$

Update Rule: Hidden Units to Output Units

$$J(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 \quad (9)$$

$$z_k = f(net_k) \quad (5)$$

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k) \quad (15)$$

After plugging it all in we have the weight update / learning rule:

$$\Delta w_{kj} = -\eta\left(\frac{\partial J}{\partial w_{kj}}\right) = -\eta(-\delta_k y_j) = \eta\delta_k y_j = \eta(t_k - z_k) f'(net_k) y_j \quad (17)$$

If we get the correct outputs, $t_k = z_k$, then $\Delta w_{kj} = 0$ as desired

Iterative form:

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta\mathbf{w}(m) = \mathbf{w}(m) + \eta(m)(t_k - z_k) f'(net_k) y_j$$

Looks a lot like the Least Mean Squared Rule when $f'(net_k)=1$:

$$\mathbf{a}(k+1) = \mathbf{a}(k) + \Delta\mathbf{a}(k) = \mathbf{a}(k) + \eta(k)\,(b(k) - \mathbf{a}^{\mathbf{t}}(k) y^k) y^k \quad (61)$$

Update Rule: Input Units to Hidden Units

The update rule for a connection weight between hidden unit j and output unit k :

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

Working backwards from outputs to inputs, we need to find the update rule for a connection weight between input unit i and hidden unit j:

$$\Delta w_{ji} = \eta \delta_j x_i = \eta \left(\left[\sum_{k=1}^{c} w_{kj} \delta_k \right] f'(net_j) \right) x_j \quad (21)$$

Intuitive reasoning

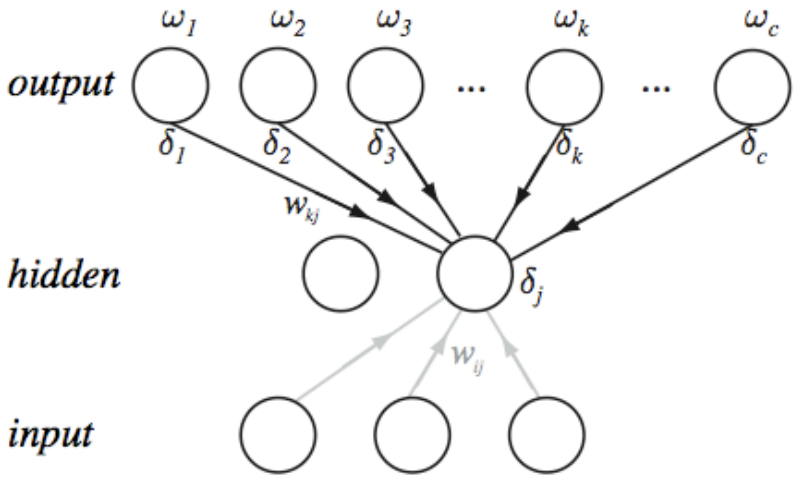


FIGURE 6.5. The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units: $\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k$. The output unit sensitivities are thus propagated "back" to the hidden units.

Backpropagation Analysis

What if we set the all the initial weights, w_{kj}, to 0 in the update rule?

$$\Delta w_{ji} = \eta \delta_j x_i = \eta \left(\left[\sum_{k=1}^{c} w_{kj} \delta_k \right] f'(net_j) \right) x_j \quad (21)$$

Clearly, $\Delta w_{ji} = 0$ and the weights would never change.

It would be bad of all network weights were ever equal to 0.

This is why we start the network with random weights.

The update rules discussed only apply to a 3-layer network where all of the connections were between a layer and it's preceeding layer.

The concepts can be generalized to apply to a network with n hidden layers, to include bias units, to handle a different learning rate for each unit, etc.

It's more difficult to apply these concepts to recurrent networks where there are connections from higher layers back to lower layers/

Training Protocols

Training consists of presenting to the network the collection of patters whose category we know, collectively known as the training set. Then we go about finding the output of the network and adjusting the weights to make the next output more like the desired target values.

The three most useful training protocols are:

- Stochastic
- On-Line
- Batch

In the stochastic and batch training methods, we will usually make several passes through the training data. For on-line training, we will use each pattern once and only once.

We use the term *epoch* to describe the overall number of training set passes. The number of epochs is an indication of the relative amount of learning.

Stochastic Training

In stochastic training patterns are chosen at random from the training set and network weights are updated for each pattern presentation.

Stochastic Back propagation Algorithm

1. begin initialize n_H , **w**, criterion θ, η, $m \leftarrow 0$
2. do m $\leftarrow$ m + 1
3. $\mathbf{x}^m \leftarrow$ randomly chosen pattern
4. $w_{ji} \leftarrow w_{ji} + \eta \delta_j x_i$; $w_{kj} \leftarrow w_{kj} + \eta \delta_k y_j$
5. until $\|\nabla J(\mathbf{w})\| < \theta$
6. return **w**
7. end

On-Line Training

In on-line training, each pattern is presented once and only once.
There is no use of memory for storing the patterns.

Useful for live training with human testers.

On-line Back propagation Algorithm

1. begin initialize n_H , **w**, criterion θ, η, $m \leftarrow 0$
2. do m$\leftarrow$ m+1
3. $\mathbf{x}^m \leftarrow$ select a new, unique pattern to be used
4. $w_{ji} \leftarrow w_{ji} + \eta \delta_j x_i \quad w_{kj} \leftarrow w_{kj} + \eta \delta_k y_j$
5. until $\|\nabla J(\mathbf{w})\| < \theta$
6. return **w**
7. end

Batch Training

In batch training, the entire training set is presented first and their corresponding weight updates are summed; from there the actual weights in the network are updated. This process is iterated until the stopping criteria is met.

The total training error over *n* individual patterns can be written as: $\sum_{p=1}^{n} J_p$

Batch Back propagation Algorithm

1. begin initialize n_H , **w**, criterion θ , η , $m \leftarrow 0$
2. do r $\leftarrow$ m+1 (increment epoch)
3. m $\leftarrow$ 0; $\Delta w_{ji} \leftarrow 0; \Delta w_{kj} \leftarrow 0;$
4. do m $\leftarrow$ m+1
5. x^m = select pattern
6. $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \delta_j x_i; \Delta w_{kj} \leftarrow \Delta w_{kj} + \eta \delta_{kj} y_{ji};$
7. until m=n
8. $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}; w_{kj} \leftarrow w_{kj} + \Delta w_{kj}$
9. until $\|\nabla J(\mathbf{w})\| < \theta$
10. return **w**
11. end

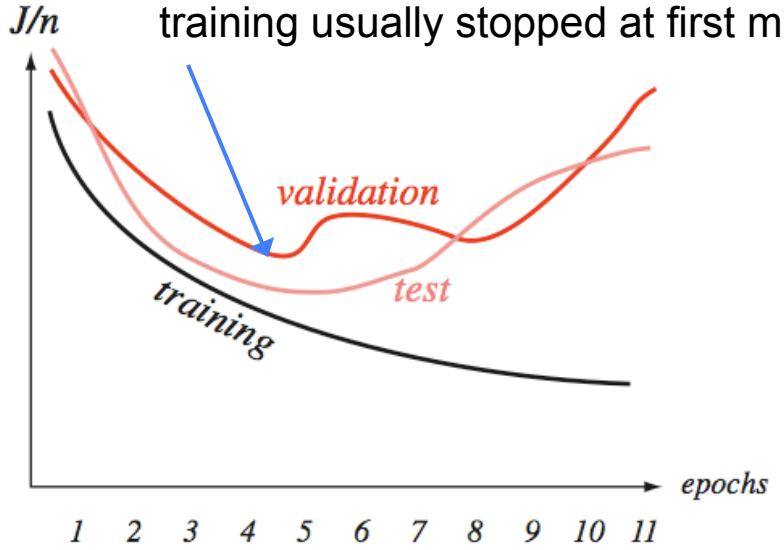
Learning Curves

training set – network hones weights using update rule

test set – estimate generalization error due to weight training; performance of fielded network.

validation set – represent novel patterns not yet classified

training usually stopped at first minimum of validation set



low expressive power (few network weights)

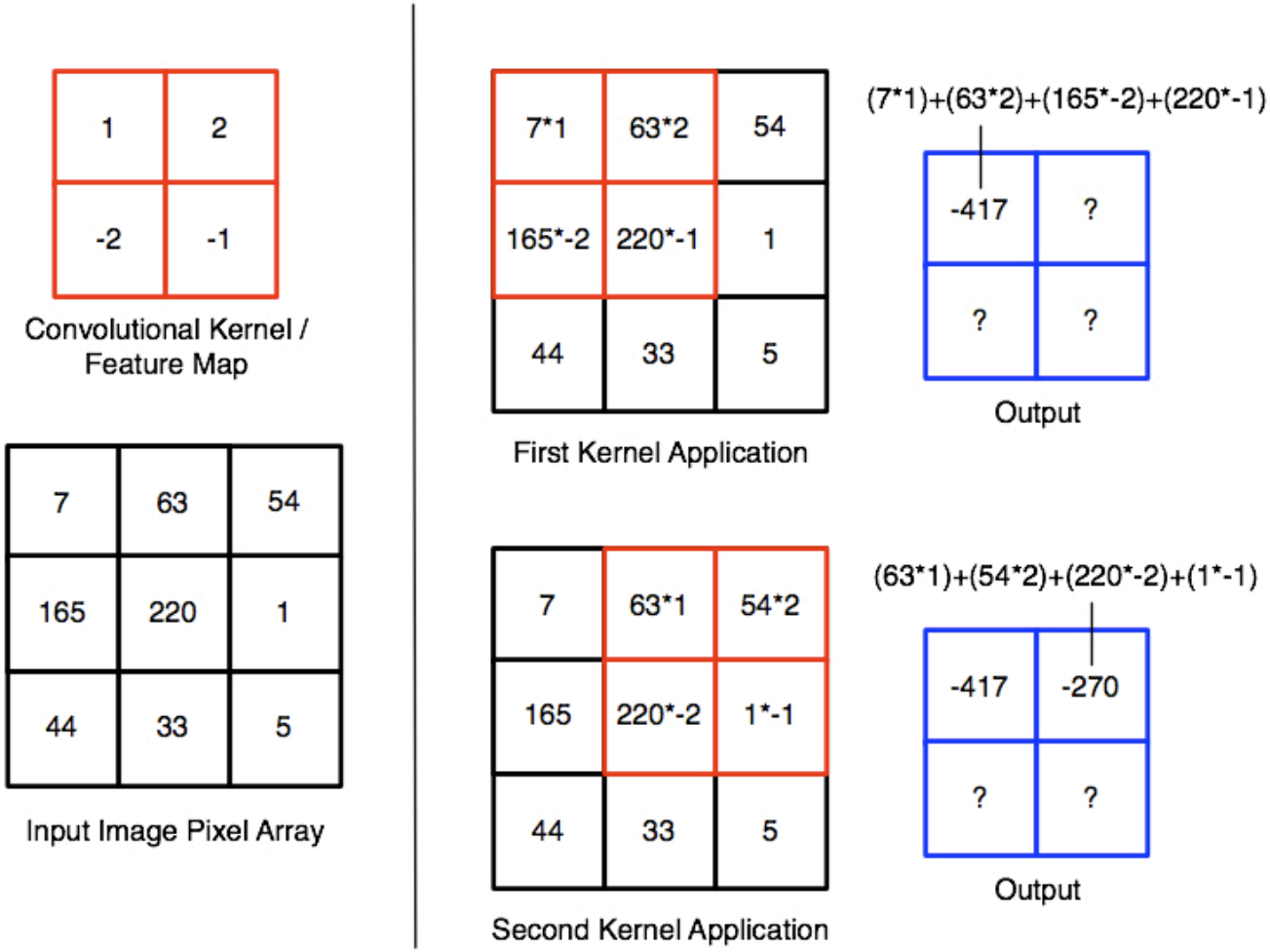
+ high Bayes error

= high asymptotic training error

FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n \sum_{p=1}^{n} J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set.

Convolutional Neural Network

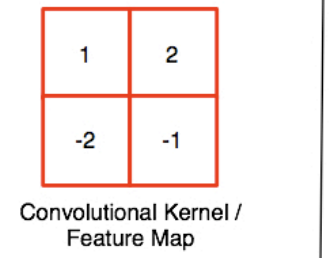
Convolutional kernel application example:



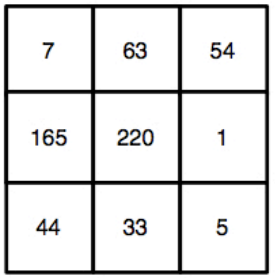
Convolutional Neural Network

Each pixel in the sample MNIST image becomes an input node.

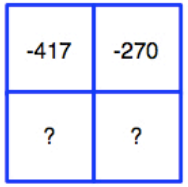
Convolutional kernel is applied to the input image in the second layer:



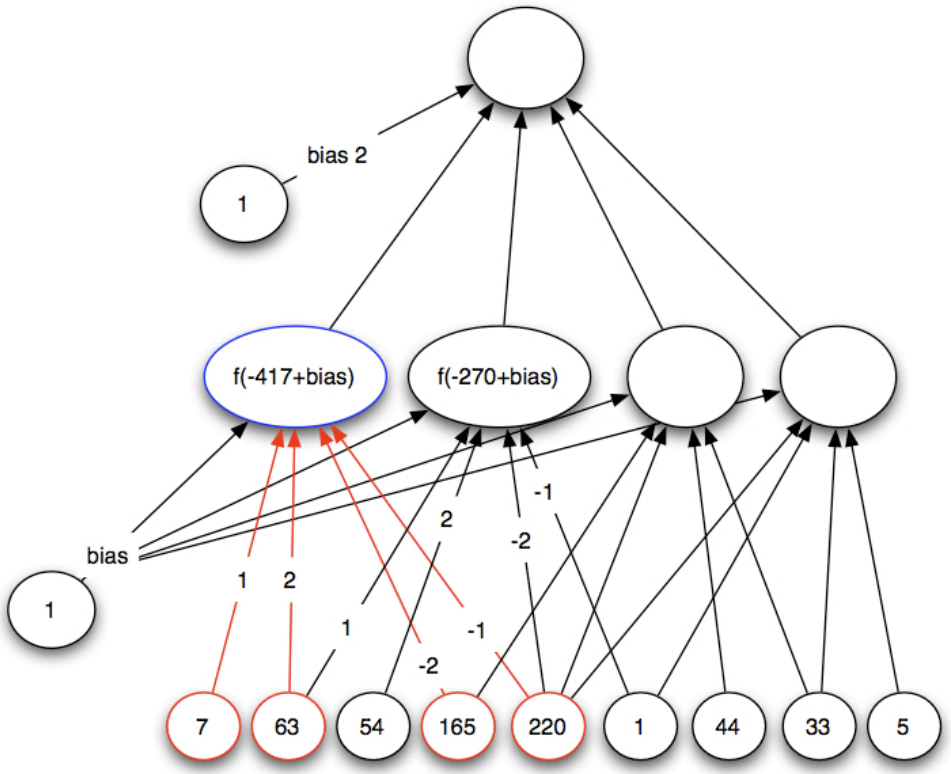
Convolutional Kernel / Feature Map



Input Image Pixel Array



Output



Convolutional Neural Network Applied to Handwritten Digit Recognition

Simard recommends a convolutional neural network consisting of 5 layers.

Input layer consists of one neuron per pixel in a 29x29 padded MNIST sample image

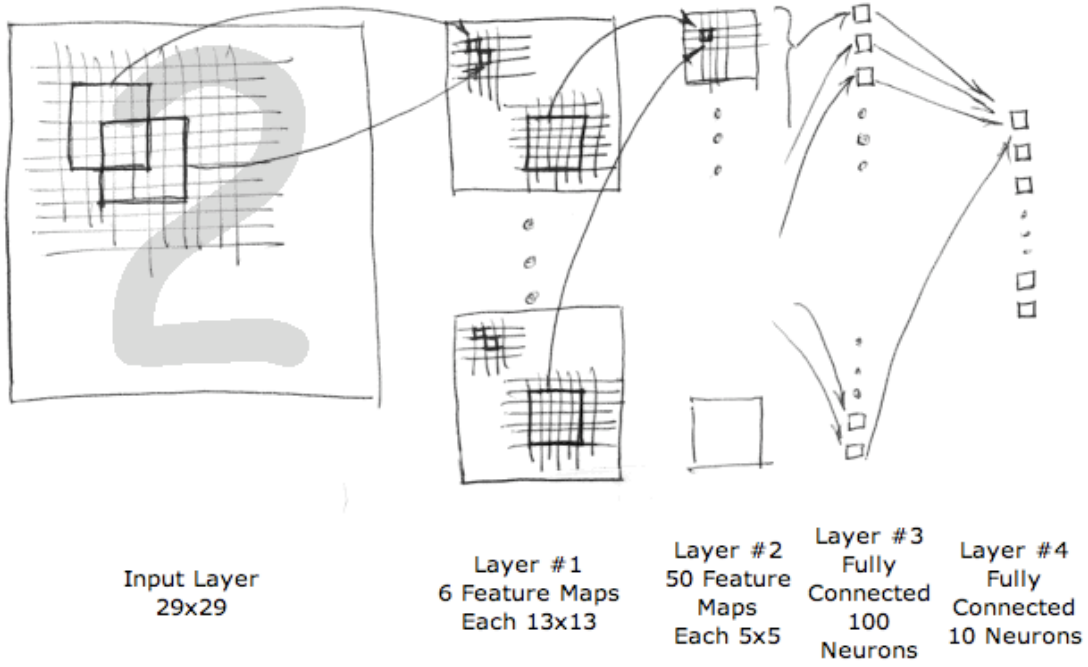
First layer applies 6 feature maps to the input layer. Each map is a randomly distributed 5x5 kernel.

Second layer applies 50 feature maps to all 6 previous maps. Each map is a randomly distributed 5x5 kernel.

First and second layers referred to as trainable feature extractor.

Third and fourth layers are fully connected and compose a universal classifier with 100 hidden units.

Third and fourth layers referred to as a trainable feature classifier.



http://www.codeproject.com/KB/library/NeuralNetRecognition/IllustrationNeuralNet.gif

Structural Adaptability

One of the biggest issues with neural networks is the selection of a suitable structure for the network. This is especially true in unknown environments.

Lee Tsu-Chang developed a method for altering the structure of the neural network during the learning process.

If, during training, the error has stabilized but is larger than our target error, we will generate a new hidden layer neuron.

A neuron can be annihilated when it is no longer a functioning element of the network. This occurs if the neuron is a redundant element, has a constant output, or is entirely dependent on another neuron.

These criteria can be checked by monitoring the input weight vectors of neurons in the same layer. If two of them are linear dependent, then the represent the same hyper plane in the data space spanned by their receptive field and are totally dependent on each other. If the output is a constant value, then the output entropy is approximately zero.

This methodology is incredibly useful for finding the optimal design for a neural network without requiring extensive domain knowledge.

Structural Adaptability

In the graph to the right, the letters A, B, and C represent the times when structural adaptability caused a hidden layer neuron to be removed. Note that these occurred at performance plateaus and did in fact improve the error rate.

The error rate was calculated for an OCR network over each upper-case English letter.

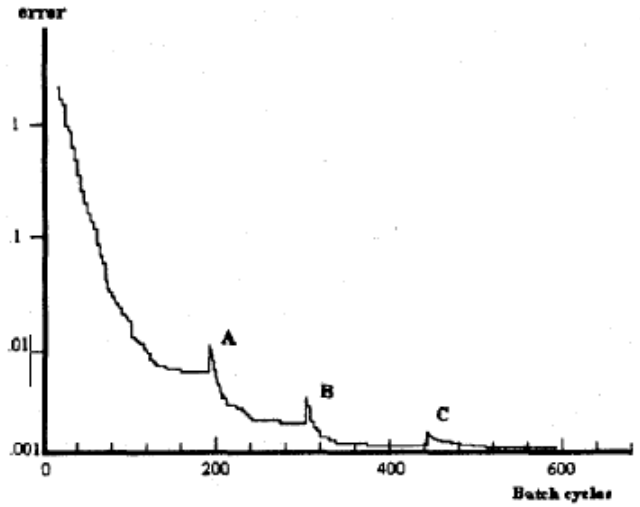


Figure 2. Error convergence; structural adaptability.

References

Duda, R., Hart, P., Stork, D. Pattern Classification, 2nd ed. John Wiley & Sons, 2001.

Y. LeCun, L. Bottou, G.B. Orr, and K.-R. Muller. Efficient Backprop. Neural Networks: Tricks of the Trade, Springer Lecture Notes in Computer Sciences, No. 1524, pp. 5-50, 1998.

P. Simard, D. Steinkraus, and J. Platt. Best Practice for Convolutional Neural Networks Applied to Visual Document Analysis. International Conference on Document Analysis and Recognition, ICDAR 2003, IEEE Computer Society, Vol. 2, pp. 958-962, 2003.

L. Tsu-Chang, *Structure Level Adaptation for Artificial Neural Networks, Kluwer Academic* Publishers, Boston, 1991.

Ramirez, J.M.; Gomez-Gil, P.; Baez-Lopez, D., "On structural adaptability of neural networks in character recognition," *Signal Processing, 1996., 3rd International Conference on* , vol.2, no., pp.1481-1483 vol.2, 14-18 Oct 1996

Chapter 6: Multilayer Neural Networks (Sections 1-5, 8)

1. Introduction
2. Feedforward Operation and Classification
3. Backpropagation Algorithm
4. Error Surfaces
5. Backpropagation as Feature Mapping

8. Practical Techniques for Improving Backpropagation

1. Introduction

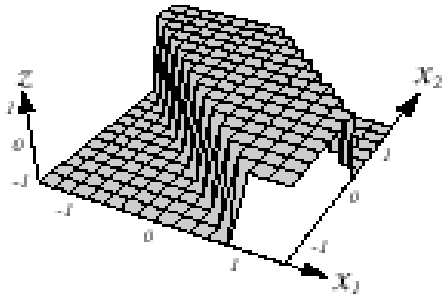
- Goal: Classify objects by learning nonlinearity
  - There are many problems for which linear discriminants are insufficient for minimum error
  - In previous methods, the central difficulty was the choice of the appropriate nonlinear functions
  - A “brute” approach might be to select a complete basis set such as all polynomials; such a classifier would require too many parameters to be determined from a limited number of training samples

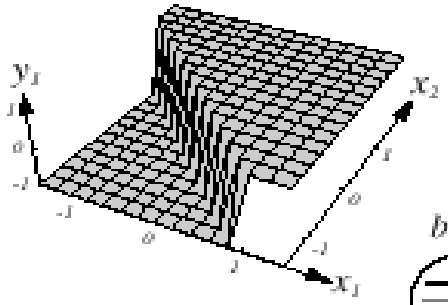
- There is no automatic method for determining the nonlinearities when no information is provided to the classifier

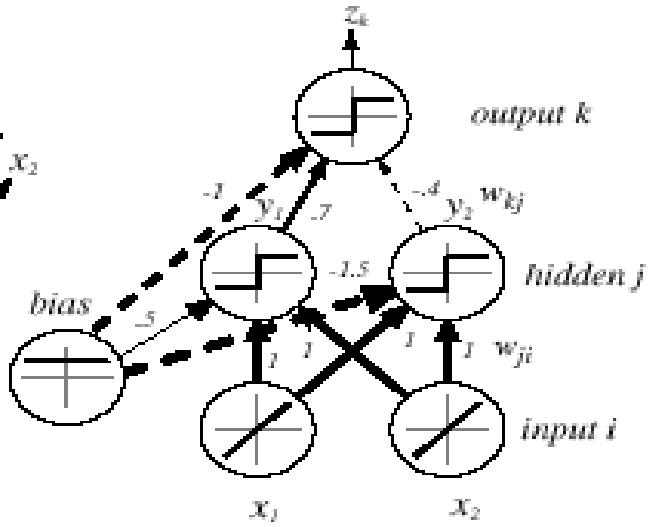
- In using the multilayer Neural Networks, the form of the nonlinearity is learned from the training data

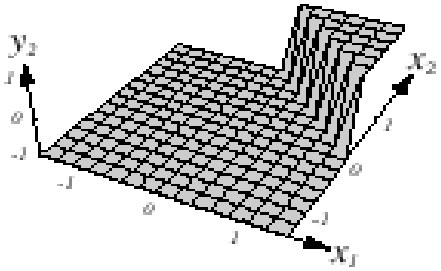
2. Feedforward Operation and Classification

- A three-layer neural network consists of an input layer, a hidden layer and an output layer interconnected by modifiable weights represented by links between layers


z
x2
x1


y1
x2
x1


zk
output k
wkj
hidden j
wji
input i
bias
y1
y2
x1
x2


y2
x2
x1

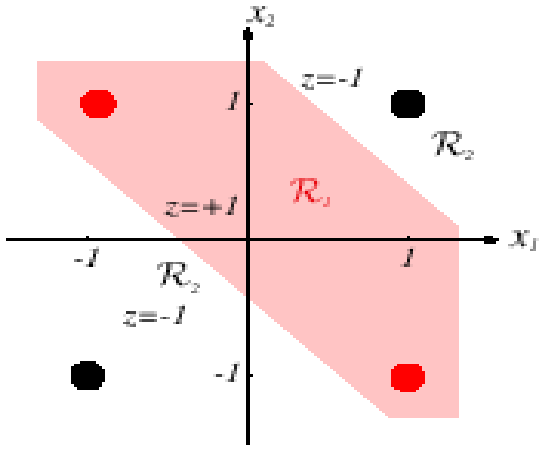

x2
z=-1
z=+1
x1
z=-1

FIGURE 6.1. The two-bit parity or exclusive-OR problem can be solved by a three-layer network. At the bottom is the two-dimensional feature $x_1\,x_2$-space, along with the four patterns to be classified. The three-layer network is shown in the middle. The input units are linear and merely distribute their feature values through multiplicative weights to the hidden units. The hidden and output units here are linear threshold units, each of which forms the linear sum of its inputs times their associated weight to yield *net*, and emits a +1 if this *net* is greater than or equal to 0, and −1 otherwise, as shown by the graphs. Positive or "excitatory" weights are denoted by solid lines, negative or "inhibitory" weights by dashed lines; each weight magnitude is indicated by the line's thickness, and is labeled. The single output unit sums the weighted signals from the hidden units and bias to form its *net*, and emits a +1 if its *net* is greater than or equal to 0 and emits a −1 otherwise. Within each unit we show a graph of its input-output or activation function—$f(net)$ versus *net*. This function is linear for the input units, a constant for the bias, and a step or sign function elsewhere. We say that this network has a 2-2-1 fully connected topology, describing the number of units (other than the bias) in successive layers. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

- A single “bias unit” is connected to each unit other than the input units
- Net activation: $$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv w_j^t . x,$$

  where the subscript i indexes units in the input layer, *j* in the hidden; w_{ji} denotes the input-to-hidden layer weights at the hidden unit *j*. (In neurobiology, such weights or connections are called “synapses”)

- Each hidden unit emits an output that is a nonlinear function of its activation, that is: $y_j = f(net_j)$

Figure 6.1 shows a simple threshold function

$$f(net) = sgn(net) \equiv \begin{cases} 1 & \text{if } net \geq 0 \\ -1 & \text{if } net < 0 \end{cases}$$

- The function *f(.)* is also called the activation function or "nonlinearity" of a unit. There are more general activation functions with desirables properties

- Each output unit similarly computes its net activation based on the hidden unit signals as:

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = w_k^t . y,$$

where the subscript *k* indexes units in the ouput layer and n_H denotes the number of hidden units.

- More than one output are referred z_k. An output unit computes the nonlinear function of its net, emitting

$$z_k = f(net_k)$$

- In the case of c outputs (classes), we can view the network as computing c discriminants functions $z_k = g_k(x)$ and classify the input x according to the largest discriminant function $g_k(x) \quad \forall k = 1, \ldots, c$

- The three-layer network with the weights listed in fig. 6.1 solves the XOR problem

- The hidden unit y_1 computes the boundary:

$x_1 + x_2 + 0.5 = 0$

$\geq 0 \Rightarrow y_1 = +1$

$< 0 \Rightarrow y_1 = -1$

- The hidden unit y_2 computes the boundary:

$x_1 + x_2 - 1.5 = 0$

$\leq 0 \Rightarrow y_2 = +1$

$> 0 \Rightarrow y_2 = -1$

- The final output unit emits $z_1 = +1 \Leftrightarrow y_1 = +1$ *and* $y_2 = +1$

  $z_k = y_1$ and not $y_2 = (x_1$ *or* $x_2)$ and not $(x_1$ *and* $x_2) = x_1$ *XOR* x_2 which provides the nonlinear decision of fig. 6.1

- General Feedforward Operation – case of c output units

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

$$(k = 1,\ldots,c)$$

  - Hidden units enable us to express more complicated nonlinear functions and thus extend the classification

  - The activation function does not have to be a sign function, it is often required to be continuous and differentiable

  - We can allow the activation in the output layer to be different from the activation function in the hidden layer or have different activation for each individual unit

  - We assume for now that all activation functions to be identical

- Expressive Power of multi-layer Networks

Question: Can every decision be implemented by a three-layer network described by equation (1) ?

Answer: Yes (due to A. Kolmogorov)

"Any continuous function from input to output can be implemented in a three-layer net, given sufficient number of hidden units n_H, proper nonlinearities, and weights."

$$g(x) = \sum_{j=1}^{2n+1} \delta_j \left(\Sigma \beta_{ij} (x_i) \right) \qquad \forall x \in I^n (I = [0,1]; n \geq 2)$$

for properly chosen functions δ_j and β_{ij}

- Each of the *2n+1* hidden units δ_j takes as input a sum of *d* nonlinear functions, one for each input feature x_i

- Each hidden unit emits a nonlinear function δ_j of its total input

- The output unit emits the sum of the contributions of the hidden units

Unfortunately: Kolmogorov's theorem tells us very little about how to find the nonlinear functions based on data; this is the central problem in network-based pattern recognition

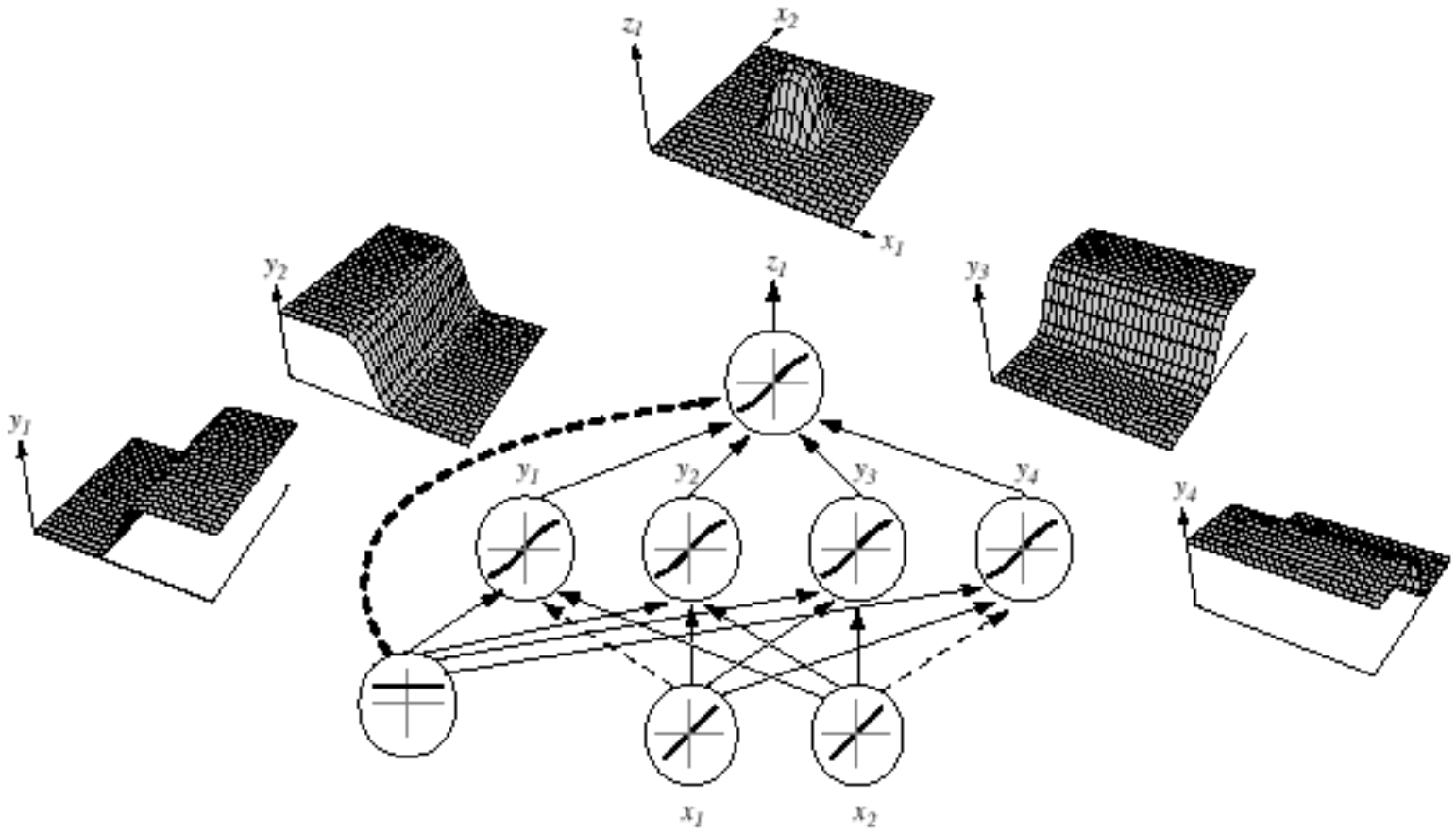


FIGURE 6.2. A 2-4-1 network (with bias) along with the response functions at different units; each hidden output unit has sigmoidal activation function $f(\cdot)$. In the case shown, the hidden unit outputs are paired in opposition thereby producing a "bump" at the output unit. Given a sufficiently large number of hidden units, any continuous function from input to output can be approximated arbitrarily well by such a network. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

3. Backpropagation Algorithm

- Any function from input to output can be implemented as a three-layer neural network

- These results are of greater theoretical interest than practical, since the construction of such a network requires the nonlinear functions and the weight values which are unknown!

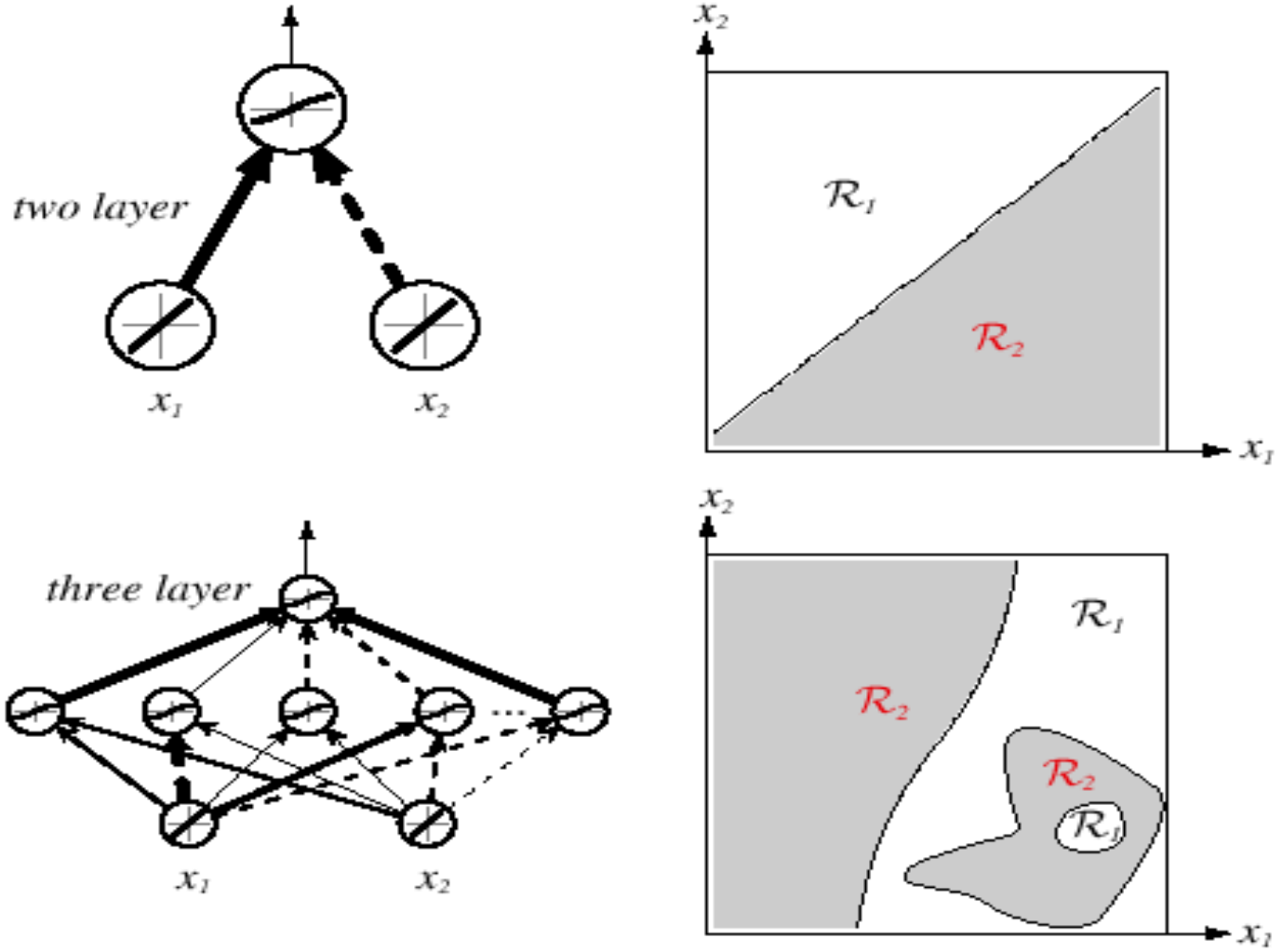


FIGURE 6.3. Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex or simply connected. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

- Our goal now is to set the interconnexion weights based on the training patterns and the desired outputs

- In a three-layer network, it is a straightforward matter to understand how the output, and thus the error, depend on the hidden-to-output layer weights

- The power of backpropagation is that it enables us to compute an effective error for each hidden unit, and thus derive a learning rule for the input-to-hidden weights, this is known as:

The credit assignment problem

- Network have two modes of operation:

  - Feedforward

    The feedforward operations consists of presenting a pattern to the input units and passing (or feeding) the signals through the network in order to get outputs units (no cycles!)

  - Learning

    The supervised learning consists of presenting an input pattern and modifying the network parameters (weights) to reduce distances between the computed output and the desired output

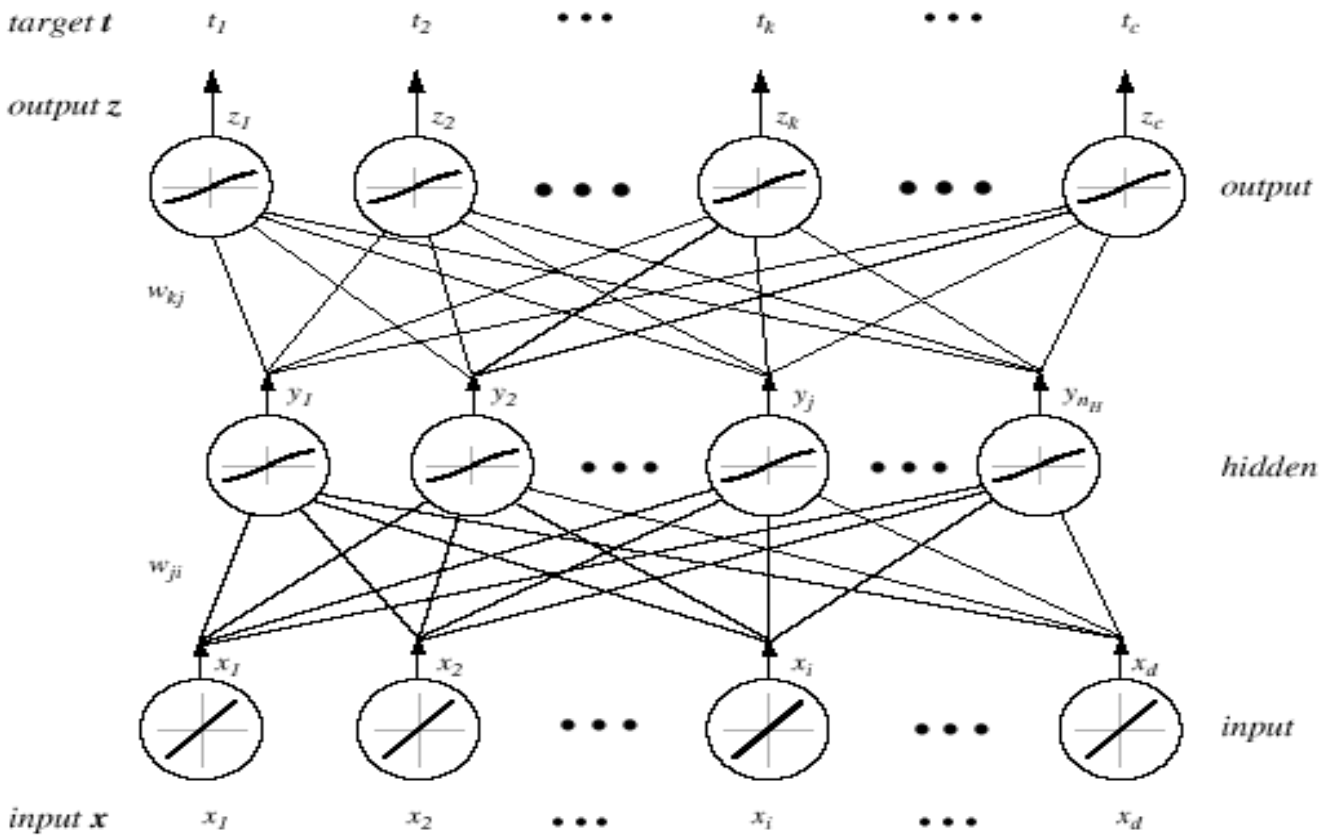


FIGURE 6.4. A d-n_H-c fully connected three-layer network and the notation we shall use. During feedforward operation, a d-dimensional input pattern $\mathbf{x}$ is presented to the input layer; each input unit then emits its corresponding component x_i. Each of the n_H hidden units computes its net activation, net_j, as the inner product of the input layer signals with weights w_{ji} at the hidden unit. The hidden unit emits $y_j = f(net_j)$, where $f(\cdot)$ is the nonlinear activation function, shown here as a sigmoid. Each of the c output units functions in the same manner as the hidden units do, computing net_k as the inner product of the hidden unit signals and weights at the output unit. The final signals emitted by the network, $z_k = f(net_k)$, are used as discriminant functions for classification. During network training, these output signals are compared with a teaching or target vector $\mathbf{t}$, and any difference is used in training the weights throughout the network. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

- Network Learning
  - Let tk be the k-th target (or desired) output and zk be the k-th computed output with $k = 1, \ldots, c$ and w represents all the weights of the network
  - The training error: $$J(w) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 = \frac{1}{2}\|t - z\|^2$$
  - The backpropagation learning rule is based on gradient descent
    - The weights are initialized with pseudo-random values and are changed in a direction that will reduce the error:

$$\Delta w = -\eta \frac{\partial J}{\partial w}$$

where η is **the learning rate** which indicates the relative size of the change in weights

$$w(m+1) = w(m) + \Delta w(m)$$

where m is the m-th pattern presented

- Error on the hidden–to-output weights

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}$$

where the sensitivity of unit k is defined as: $\delta_k = -\frac{\partial J}{\partial net_k}$

and describes how the overall error changes with the activation of the unit's net

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k} \cdot \frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k)$$

Since $net_k = w_k^t.y$ therefore:

$$\frac{\partial net_k}{\partial w_{kj}} = y_j$$

Conclusion: the weight update (or learning rule) for the hidden-to-output weights is:

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

- Error on the input-to-hidden units

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j} \cdot \frac{\partial y_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}}$$

However,

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

$$= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial net_k}\cdot\frac{\partial net_k}{\partial y_j} = -\sum_{k=1}^{c}(t_k - z_k)f'(net_k)w_{kj}$$

Similarly as in the preceding case, we define the sensitivity for a hidden unit:

$$\delta_j \equiv f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k$$

which means that:"The sensitivity at a hidden unit is simply the sum of the individual sensitivities at the output units weighted by the hidden-to-output weights w_{kj}; all multipled by $f'(net_j)$"

Conclusion: The learning rule for the input-to-hidden weights is:

$$\Delta w_{ji} = \eta x_i \delta_j = \eta\underbrace{\left[\sum w_{kj}\delta_k\right]}_{\delta_j} f'(net_j)x_i$$

- Starting with a pseudo-random weight configuration, the stochastic backpropagation algorithm can be written as:

```
Begin  initialize    n_H; w, criterion θ, η, m ← 0
       do m ← m + 1
            x^m ← randomly chosen pattern
            w_ji ←  w_ji + ηδ_j x_i; w_kj ← w_kj + ηδ_k y_j
       until ||∇J(w)|| < θ
        return w
End
```

- Stopping criterion
  - The algorithm terminates when the change in the criterion function J(w) is smaller than some preset value θ
  - There are other stopping criteria that lead to better performance than this one
  - So far, we have considered the error on a single pattern, but we want to consider an error defined over the entirety of patterns in the training set
  - The total training error is the sum over the errors of n individual patterns

$$J = \sum_{p=1}^{n} J_p \qquad (1)$$

- Stopping criterion (cont.)
  - A weight update may reduce the error on the single pattern being presented but can increase the error on the full training set
  - However, given a large number of such individual updates, the total error of equation (1) decreases

- Learning Curves
  - Before training starts, the error on the training set is high; through the learning process, the error becomes smaller
  - The error per pattern depends on the amount of training data and the expressive power (such as the number of weights) in the network
  - The average error on an independent test set is always higher than on the training set, and it can decrease as well as increase
  - A validation set is used in order to decide when to stop training ; we do not want to overfit the network and decrease the power of the classifier generalization

    “we stop training at a minimum of the error on the validation set”

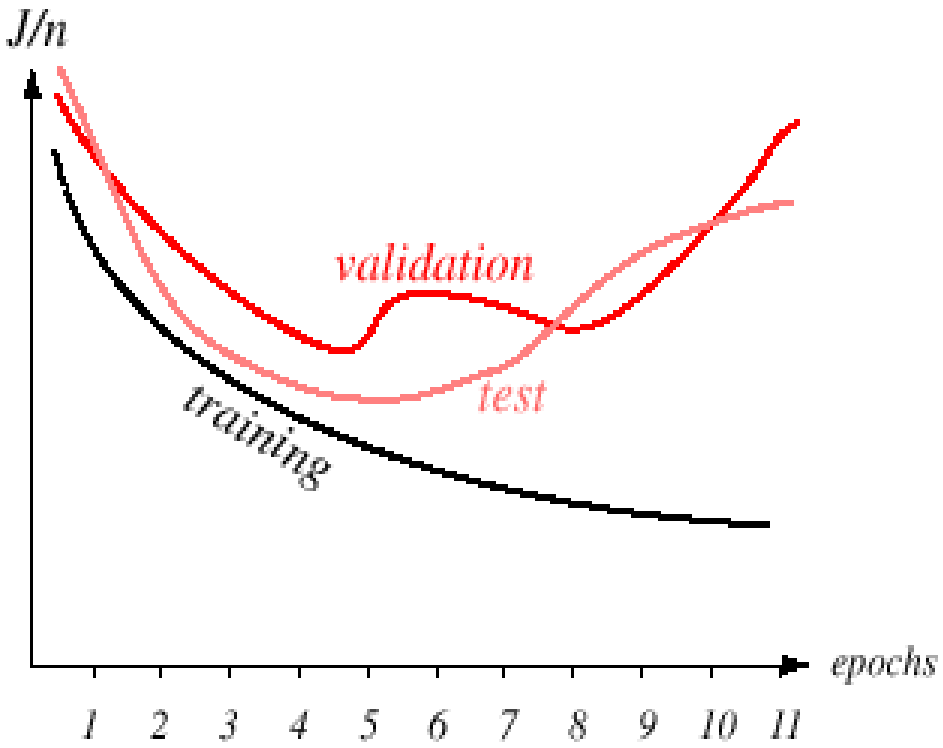


FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n \sum_{p=1}^{n} J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

4. Error Surfaces

- We can gain understanding and intuition about the backpropagation algorithm by studying error surfaces
  - The network tries to find the global minimum
  - But local minima can be a problem
  - The presence of plateaus can also be a problem

Consider the simplest three-layer nonlinear network with a single global minimum

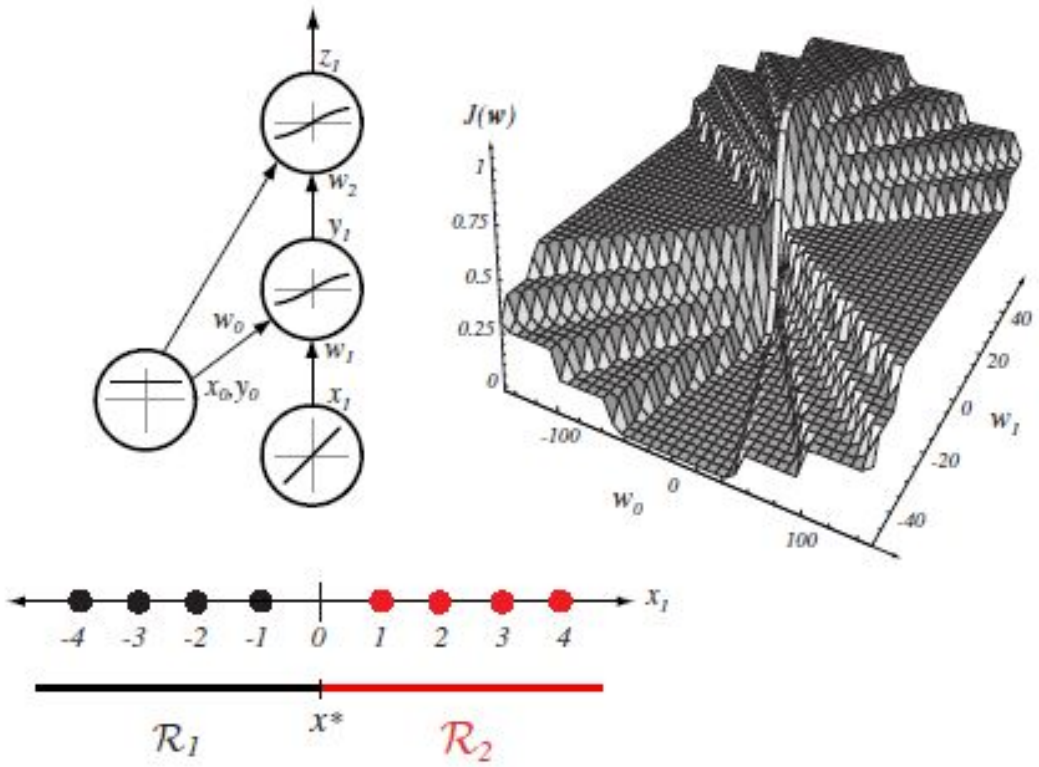


FIGURE 6.7. Eight one-dimensional patterns (four in each of two classes) are to be learned by a 1-1-1 network with steep sigmoidal hidden and output units with bias. The error surface as a function of w_0 and w_1 is also shown, where the bias weights are assigned their final values. The network starts with random weights; through stochastic training, it descends to a global minimum in error. Note especially that a low error solution exists, which indeed leads to a decision boundary separating the training points into their two categories. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Consider the same network with a problem that is not linearly separable

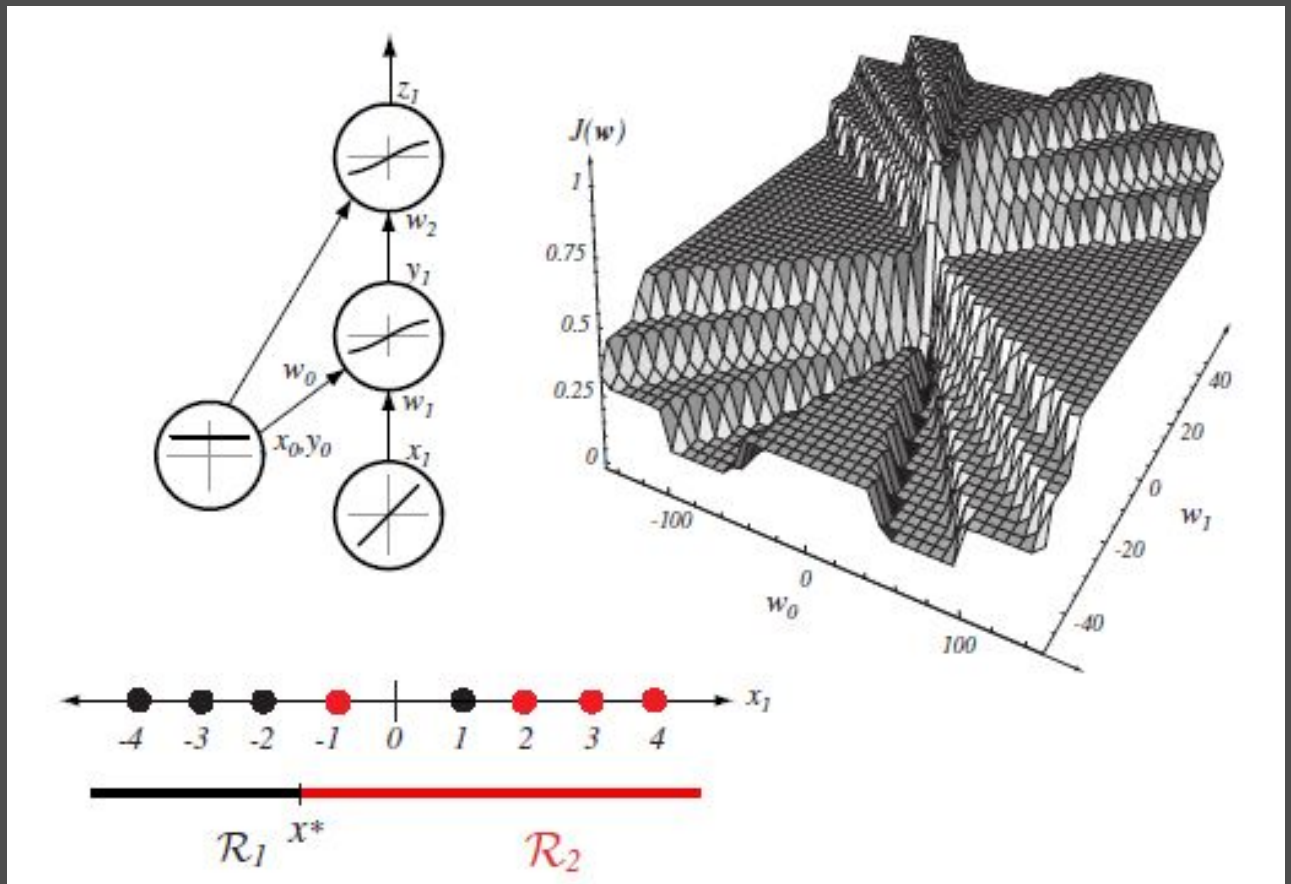


FIGURE 6.8. As in Fig. 6.7, except here the patterns are not linearly separable; the error surface is slightly higher than in that figure. Note too from the error surface that there are two forms of minimum error solution; these correspond to $-2 < x^* < -1$ and $1 < x^* < 2$, in which one pattern is misclassified. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Consider a solution to the XOR problem

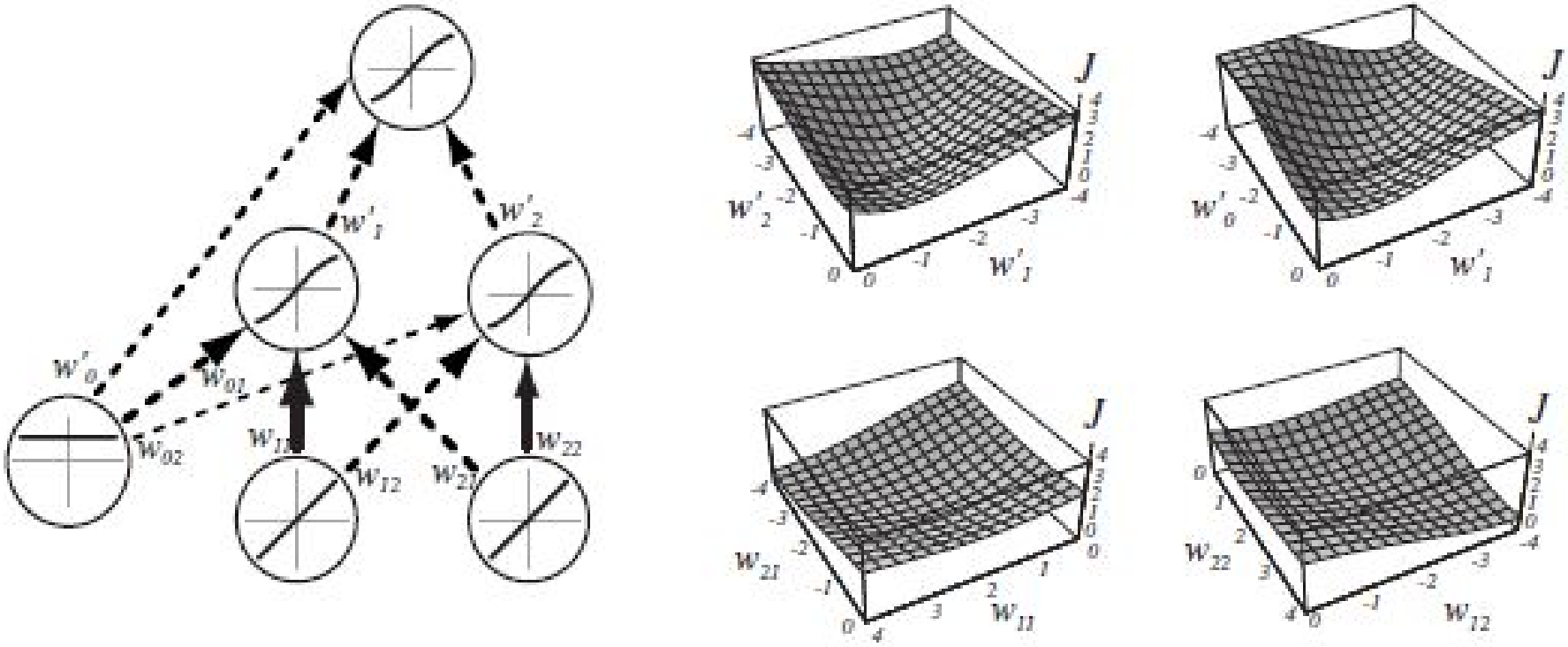


FIGURE 6.9. Two-dimensional slices through the nine-dimensional error surface after extensive training for a 2-2-1 network solving the XOR problem. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

5. Backpropagation as Feature Mapping

- Because the hidden-to-output layer leads to a linear discriminant, the novel computational power provided by multilayer neural nets can be attributed to the nonlinear warping of the input to the representation at the hidden units

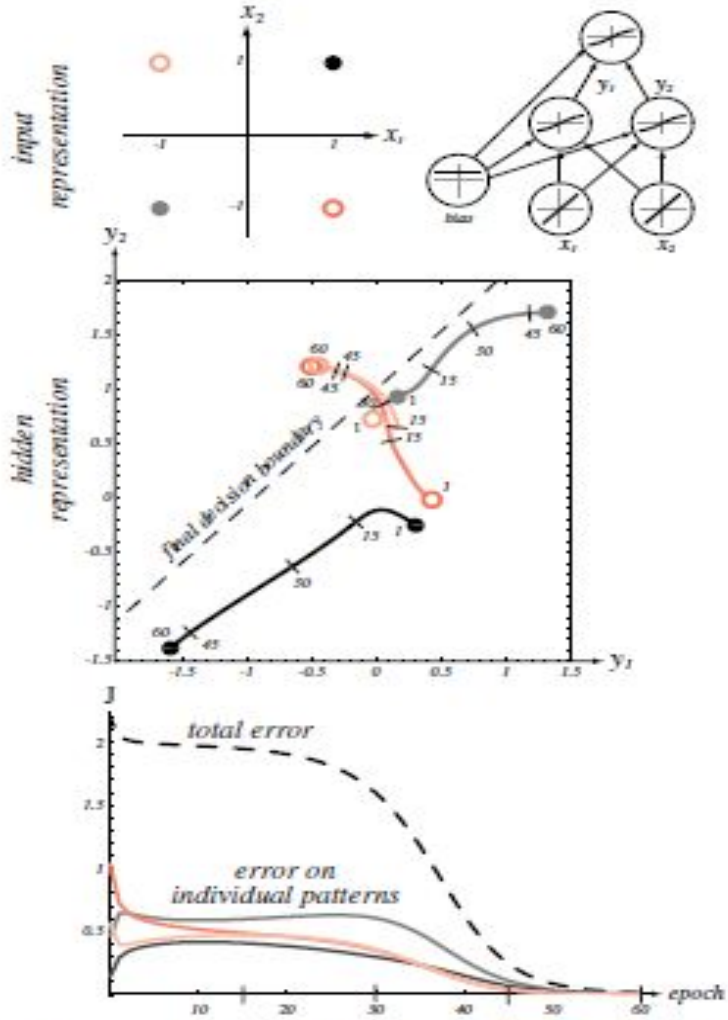


FIGURE 6.10. A 2-2-1 backpropagation network with bias and the four patterns of the XOR problem are shown at the top. The middle figure shows the outputs of the hidden units for each of the four patterns; these outputs move across the $y_1 y_2$-space as the network learns. In this space, early in training (epoch 1) the two categories are not linearly separable. As the input-to-hidden weights learn, as marked by the number of epochs, the categories become linearly separable. The dashed line is the linear decision boundary determined by the hidden-to-output weights at the end of learning; indeed the patterns of the two classes are separated by this boundary. The bottom graph shows the learning curves—the error on individual patterns and the total error as a function of epoch. Note that, as frequently happens, the total training error decreases monotonically, even though this is not the case for the error on each individual pattern. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

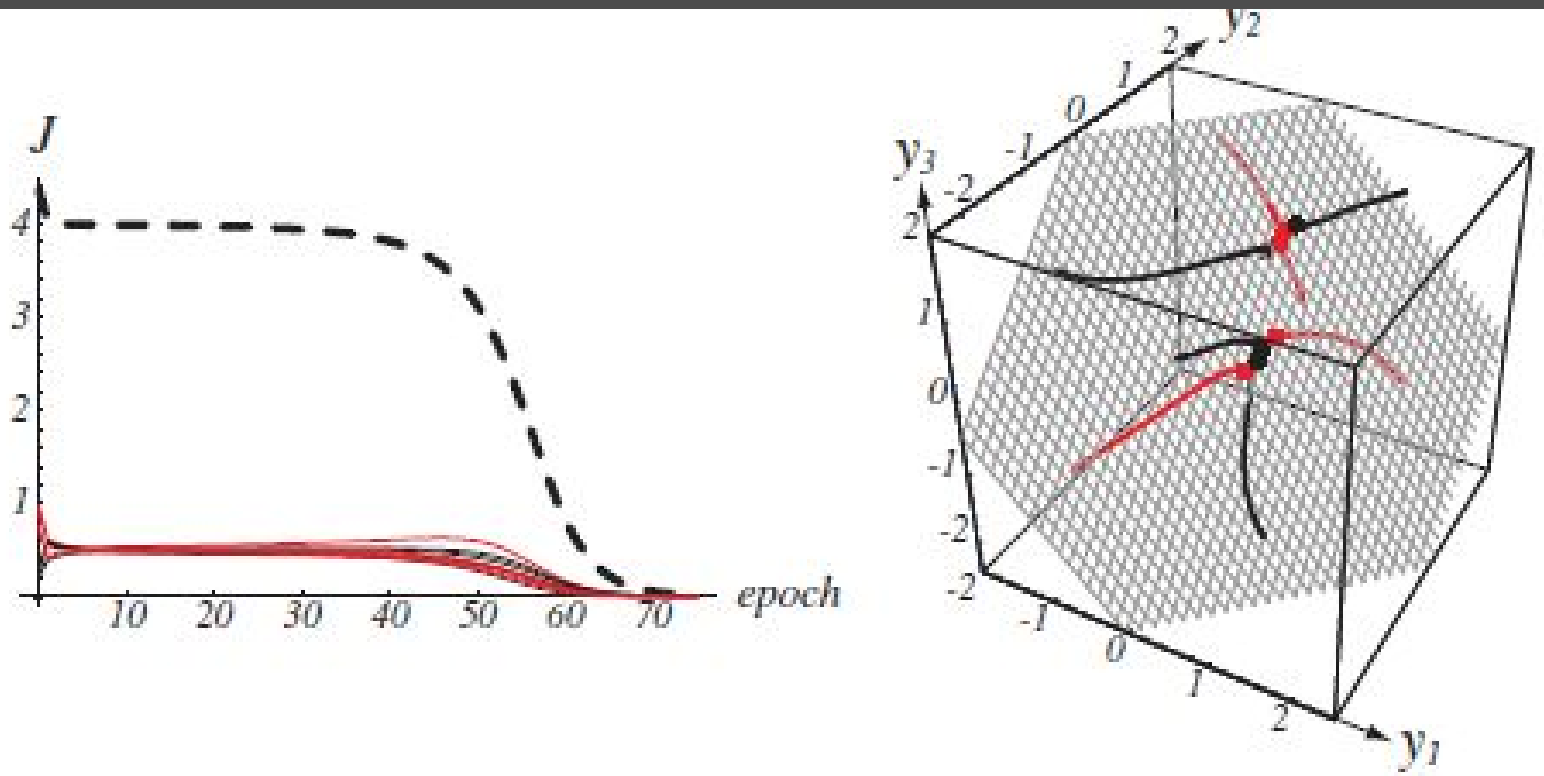


FIGURE 6.11. A 3-3-1 backpropagation network with bias can indeed solve the three-bit parity problem. The representation of the eight patterns at the hidden units ($y_1 y_2 y_3$-space) as the system learns and the planar decision boundary found by the hidden-to-output weights at the end of learning. The patterns of the two classes are indeed separated by this plane, as desired. The learning curve shows the error on individual patterns and the total error J as a function of epoch. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

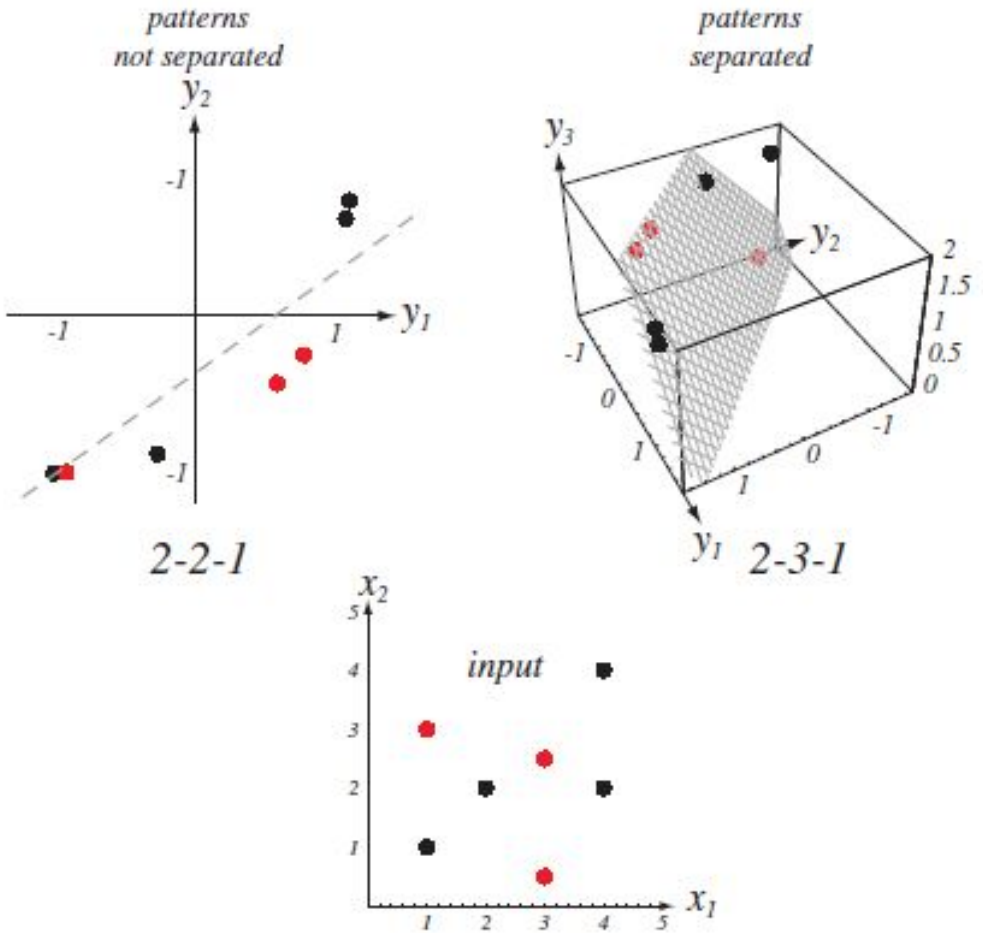


FIGURE 6.12. Seven patterns from a two-dimensional two-category nonlinearly separable classification problem are shown at the bottom. The figure at the top left shows the hidden unit representations of the patterns in a 2-2-1 sigmoidal network with bias fully trained to the global error minimum; the linear boundary implemented by the hidden-to-output weights is marked as a gray dashed line. Note that the categories are almost linearly separable in this $y_1 y_2$-space, but one training point is misclassified. At the top right is the analogous hidden unit representation for a fully trained 2-3-1 network with bias. Because of the higher dimension of the hidden layer representation, the categories are now linearly separable; indeed the learned hidden-to-output weights implement a plane that separates the categories. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Because the hidden-to-output weights => linear discriminant, the input-to-hidden weights are most instructive

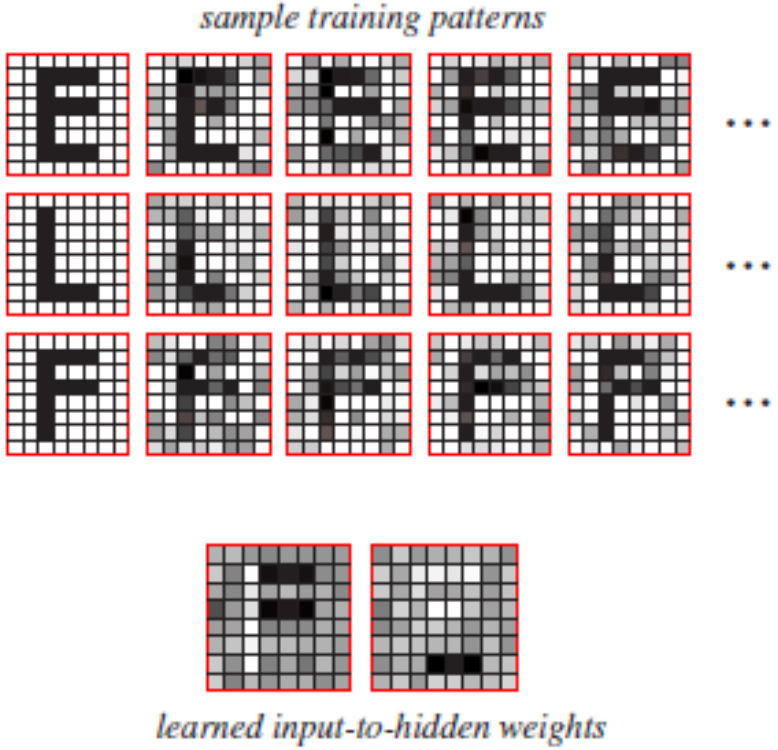


FIGURE 6.13. The top images represent patterns from a large training set used to train a 64-2-3 sigmoidal network for classifying three characters. The bottom figures show the input-to-hidden weights, represented as patterns, at the two hidden units after training. Note that these learned weights indeed describe feature groupings useful for the classification task. In large networks, such patterns of learned weights may be difficult to interpret in this way. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

8. Practical Techniques for Improving Backpropagation

1. Activation Function
   - Must be nonlinear
   - Must saturate – have a maximum and minimum
   - Must be continuous and smooth
   - Must be monotonic
   - Sigmoid class of functions provide these properties

2. Parameters for the Sigmoid

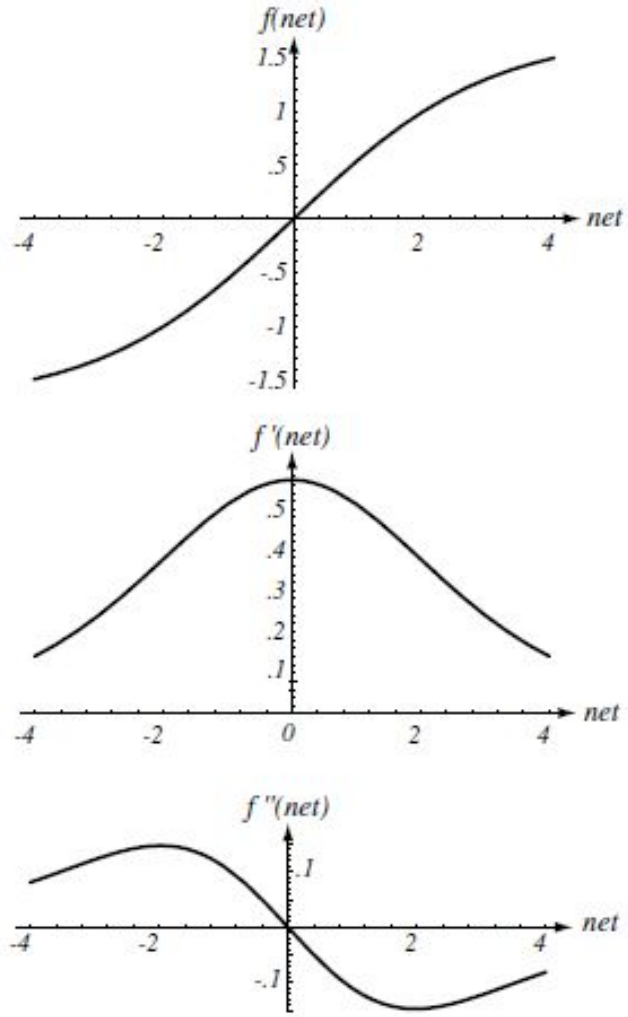


FIGURE 6.14. A useful activation function $f(net)$ is an anti-symmetric sigmoid. For the parameters given in the text, $f(net)$ is nearly linear in the range $-1 < net < +1$ and its second derivative, $f''(net)$, has extrema near $net \simeq \pm 2$. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

3. Scaling Input

Data standardization:

- The full data set should be scaled to have the same variance in each feature component

Example of importance of standardization

- In the fish classification chapter 1 example, suppose the fish weight is measured in grams and the fish length in meters
- The weight would be orders of magnitude larger than the length

4. Target Values

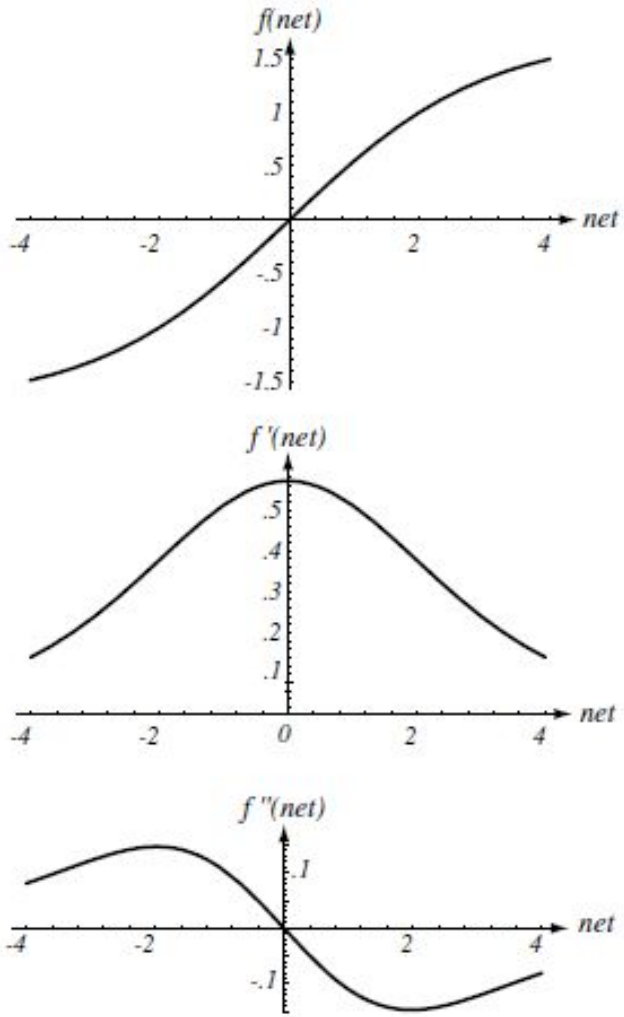


FIGURE 6.14. A useful activation function $f(net)$ is an anti-symmetric sigmoid. For the parameters given in the text, $f(net)$ is nearly linear in the range $-1 < net < +1$ and its second derivative, $f''(net)$, has extrema near $net \simeq \pm 2$. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Reasonable target values are +1 and -1

For example, for a pattern in class 3, use target vector (-1, -1, +1, -1)

5. Training with Noise

When the training set is small, additional (surrogate) training patterns can be generated by adding noise to the true training patterns.

This helps most classification methods except for highly local classifiers such as kNN

6. Manufacturing Data

As with training with noise, manufactured data can be used with a wide variety of pattern classification methods.

Manufactured training data, however, usually adds more information than Gaussian noise.

- For example, in OCR additional training patterns could be generated by slight rotations or line thickening of the the true training patterns

7. Number of Hidden Units

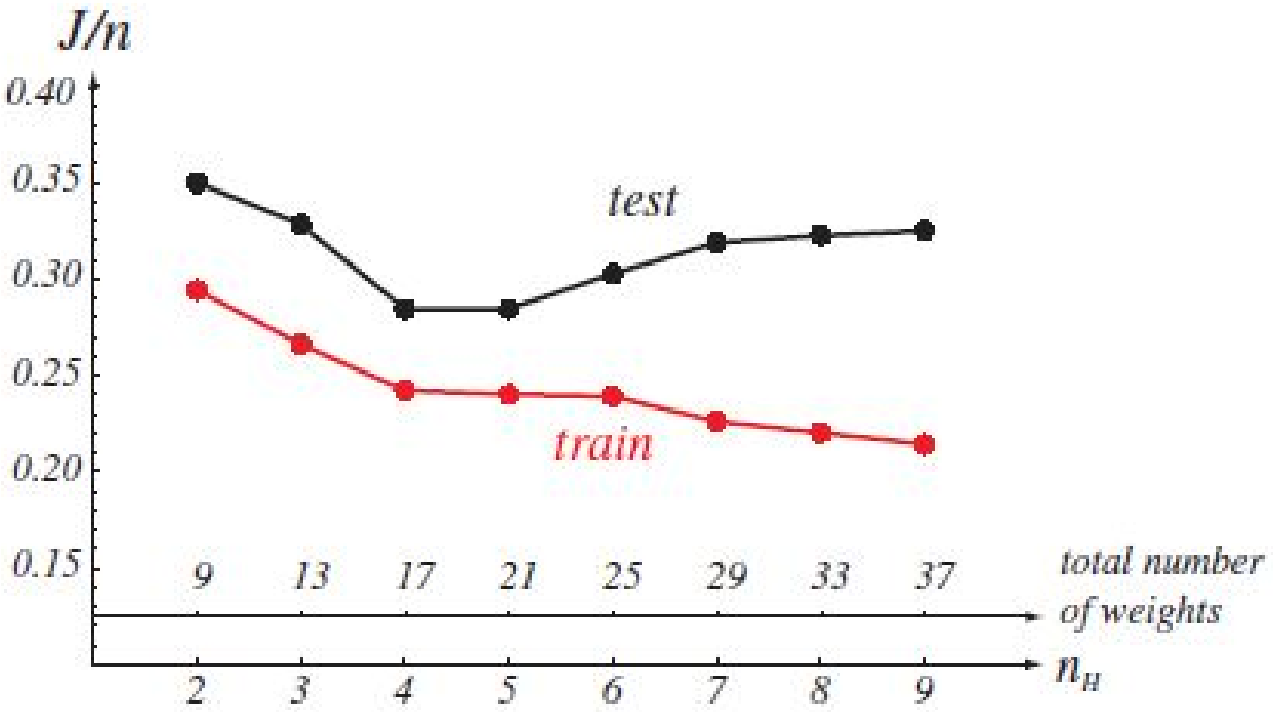


FIGURE 6.15. The error per pattern for networks fully trained but differing in the numbers of hidden units, n_H. Each $2 - n_H - 1$ network with bias was trained with 90 two-dimensional patterns from each of two categories, sampled from a mixture of three Gaussians, and thus $n = 180$. The minimum of the test error occurs for networks in the range $4 \leq n_H \leq 5$, i.e., the range of weights 17 to 21. This illustrates the rule of thumb that choosing networks with roughly $n/10$ weights often gives low test error. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

8. Initializing Weights

Initial weights cannot be zero because no learning would take place

In setting weights in a given layer, usually choose weights randomly from a single distribution to help ensure uniform learning

- Uniform learning is when all weights reach their final equilibrium values at about the same time

9. Learning Rates

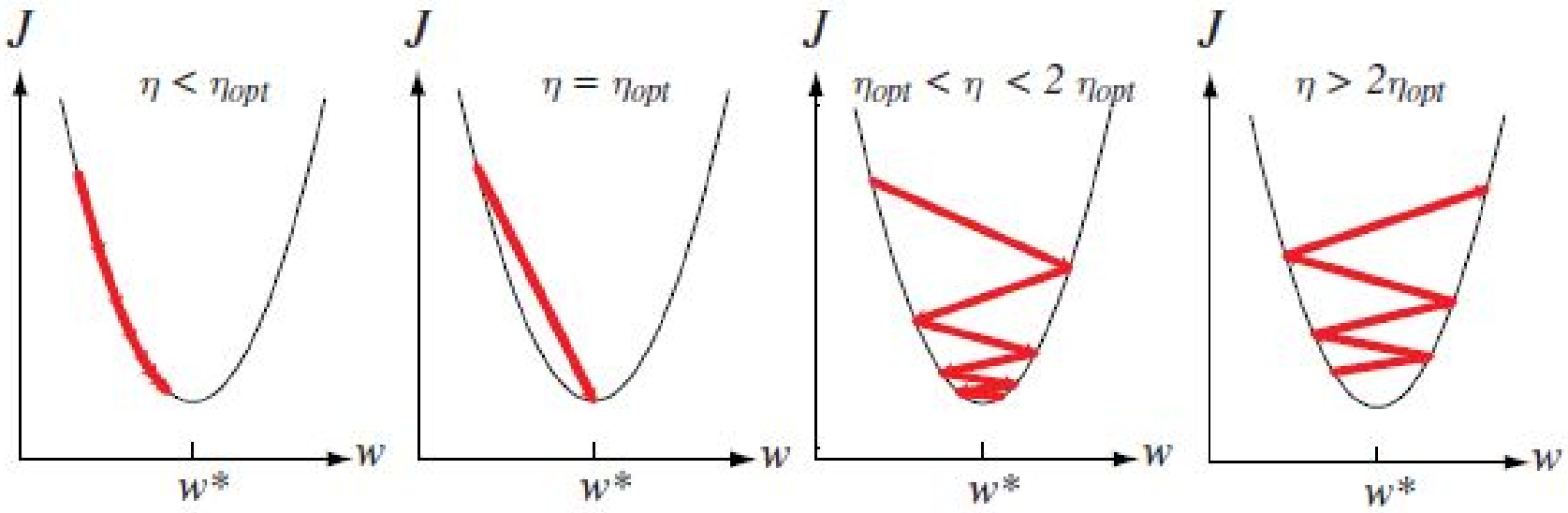


FIGURE 6.16. Gradient descent in a one-dimensional quadratic criterion with different learning rates. If $\eta < \eta_{opt}$, convergence is assured, but training can be needlessly slow. If $\eta = \eta_{opt}$, a single learning step suffices to find the error minimum. If $\eta_{opt} < \eta < 2\eta_{opt}$, the system will oscillate but nevertheless converge, but training is needlessly slow. If $\eta > 2\eta_{opt}$, the system diverges. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

10. Momentum

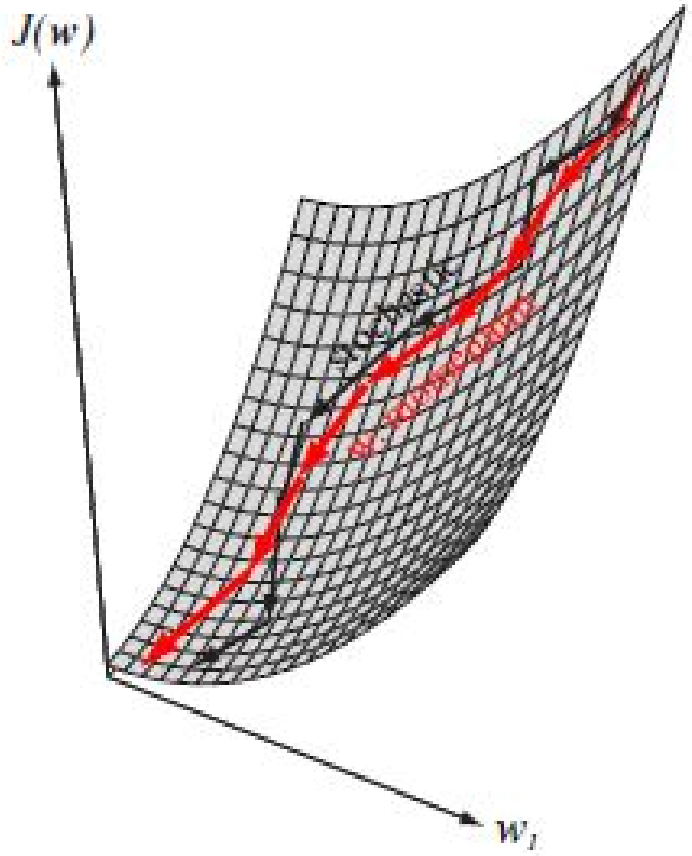


FIGURE 6.18. The incorporation of momentum into stochastic gradient descent by Eq. 37 (red arrows) reduces the variation in overall gradient directions and speeds learning. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

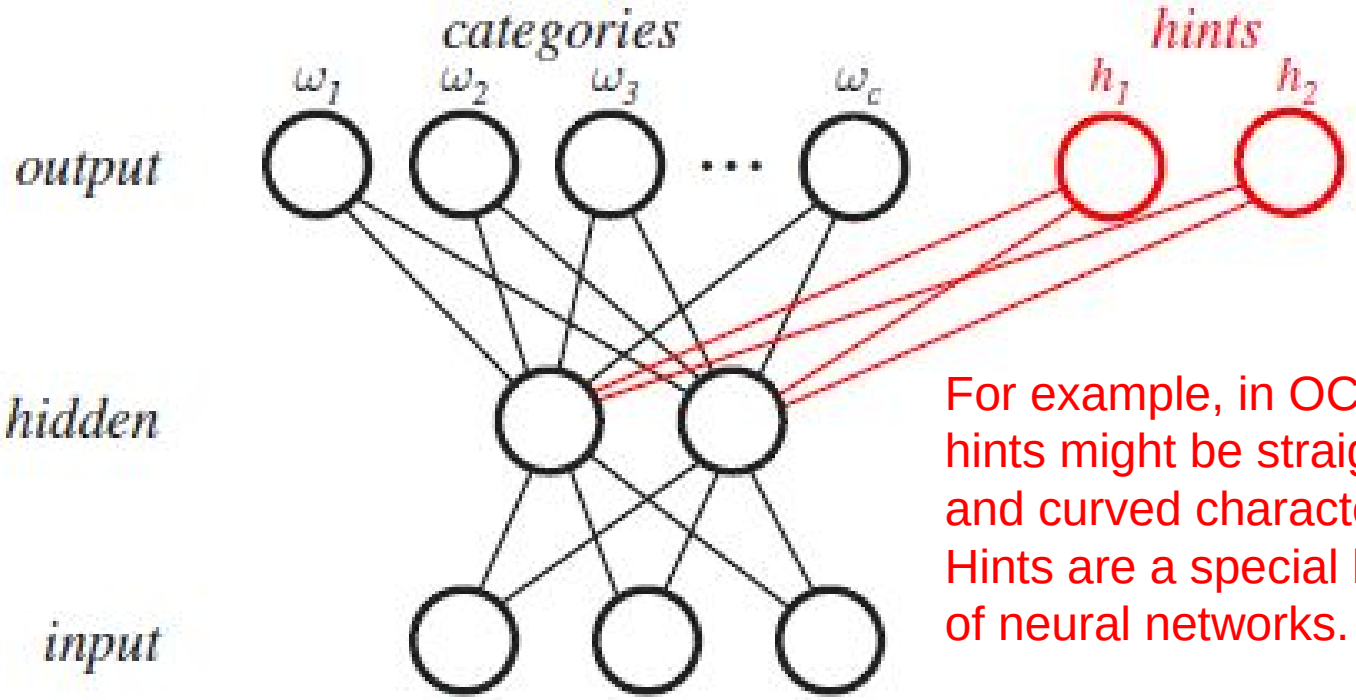
11. Weight Decay

After each weight update, every weight is simply "decayed" or shrunk

$$w^{new} = w^{old}\,(1 - \varepsilon)$$

Although there is no principled reason, weight decay helps in most cases

12. Hints



For example, in OCR the hints might be straight line and curved characters. Hints are a special benefit of neural networks.

FIGURE 6.19. In learning with hints, the output layer of a standard network having c category units is augmented with hint units. During training, the target vectors are also augmented with signals for the hint units. In this way the input-to-hidden weights learn improved feature groupings. The hint units are not used during classification, and thus they and their hidden-to-output weights are removed from the trained network. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

13. On-line, Stochastic or Batch Training?

Each of the three leading training protocols has strengths and weaknesses

- Online is usually only used when the training data is huge or memory costs high
- Batch is typically slower than stochastic learning
- For most applications stochastic training is preferred

14. Stopped Training

Best to stop training when error rate on a validation set reaches a minimum

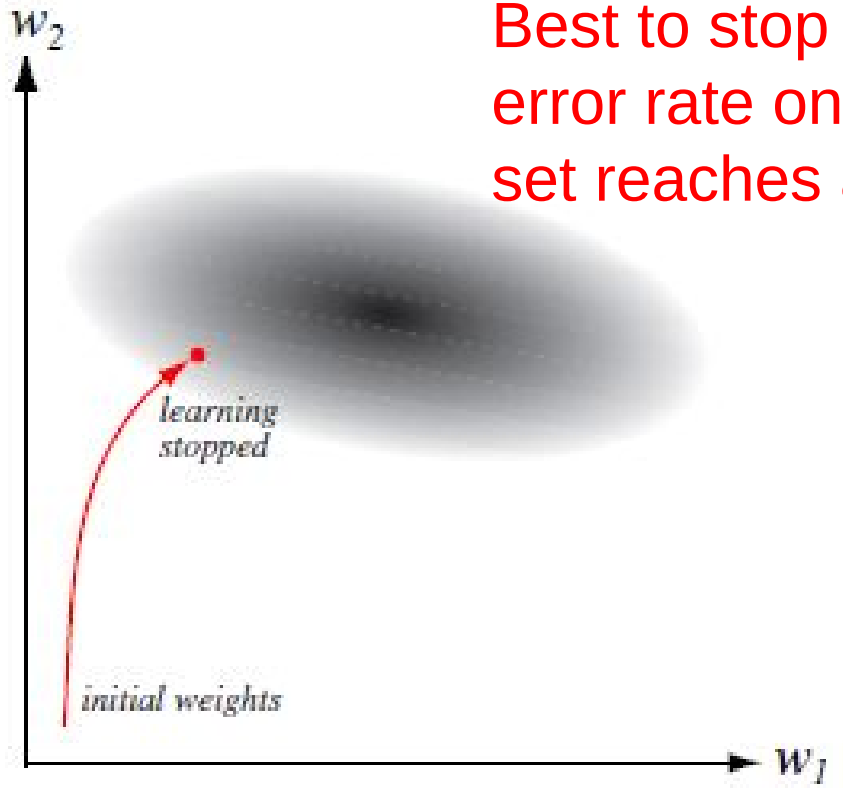


FIGURE 6.20. When weights are initialized with small magnitudes, stopped training leads to final weights that are smaller than they would be after extensive training. As such, stopped training behaves much like a form of weight decay. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

15. Number of Hidden Layers

Although three layers suffice to implement any arbitrary function, additional layers can facilitate learning

- For example, it is easier for a four-layer net to learn translations than a three-layer net

16. Criterion Function

The most common training criterion is the squared error (Eq. 9)

Alternatives are cross entropy, Minkowski error, etc.