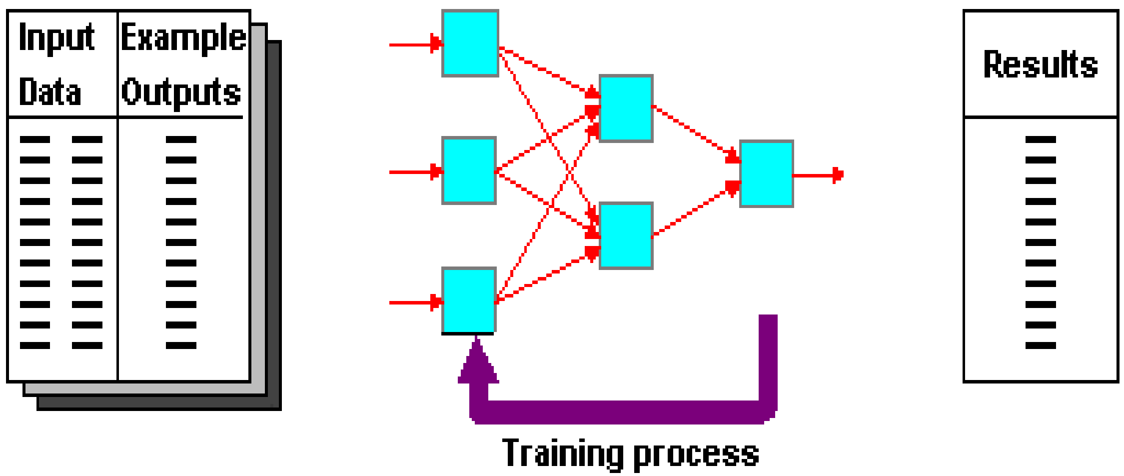
Linear Models

- Training and test data sets
- Training set: input & target
- Test set: only input

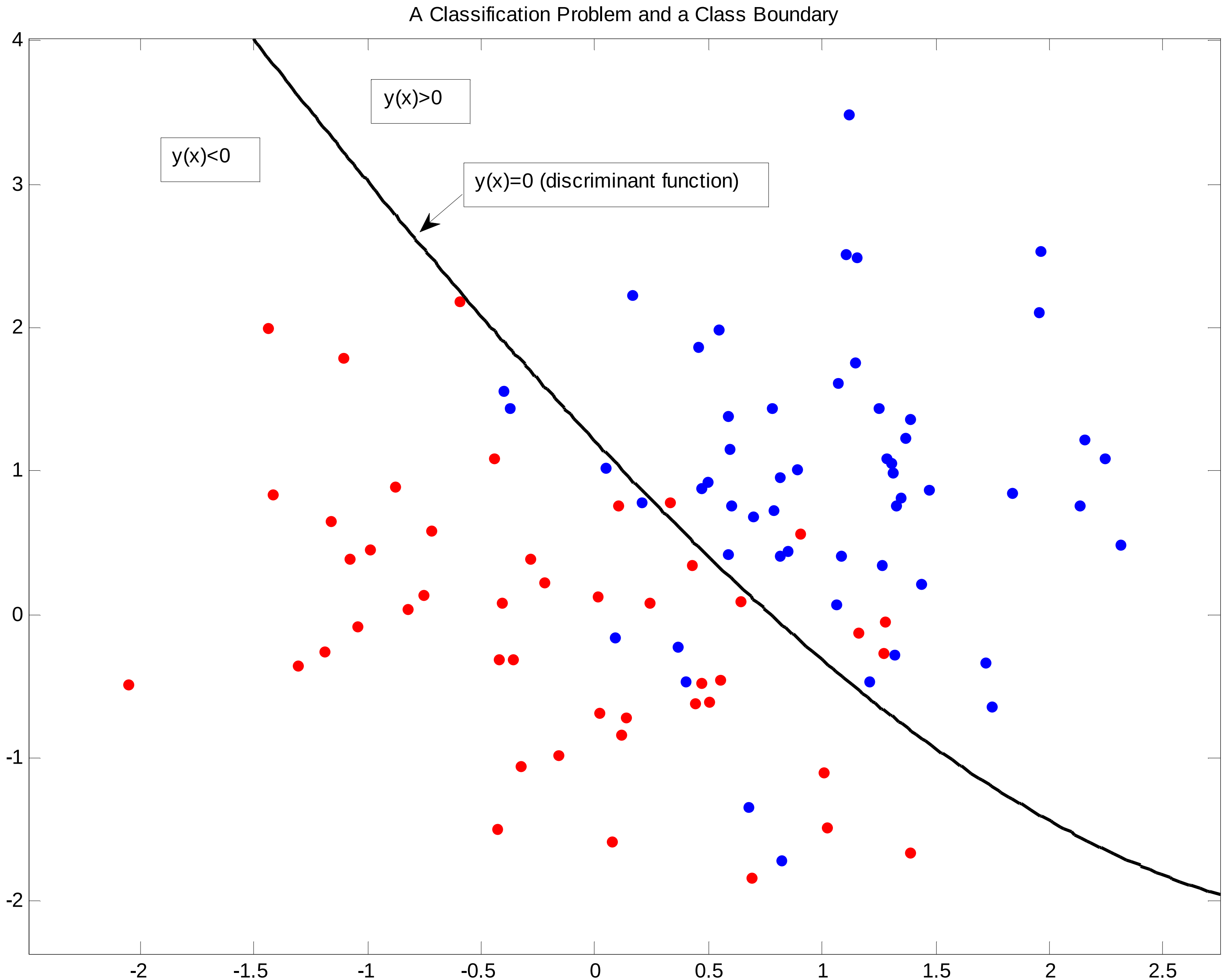


Single Layer Networks (Linear Models)

- Linear Discriminants
- Single Layer Perceptron
- Linear Separability and Cover's Theorem
- Learning Algorithms:
  - Perceptron Rule and Convergence Theorem
  - Pocket Algorithm
  - Least-Squares Method (Adaline)
  - Gradient Descent and Logistic Regression
  - Support Vector Machines
- Generalized Linear Discriminants
- Multi-class perceptron learning algorithm

Motivation

- Sometimes modeling P(Ci|X) is impossible (very few data points, very high dimensionality of data, lack of background knowledge, etc.)
- Then one can try to find class boundaries directly, assuming certain form of discriminant functions and optimizing a suitable error measure
- *The simplest case of a boundary: a line (a hyperplane) single layer networks (Perceptron; Adaline; GLM; SVM)*
- General case: multi-layer networks (Multi-layer Perceptron; RBF-newtorks)


A Classification Problem and a Class Boundary
y(x)>0
y(x)<0
y(x)=0 (discriminant function)
4
3
2
1
0
-1
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5

Discriminant Functions

- A *discriminant function for classes C_1 and C_2* is any function $y(\mathbf{x})$ such that an input vector $\mathbf{x}$ is assigned to class C_1 if $y(x) > 0$ (and to C_2 if $y(x) < 0$) [what to do with ties?]

- In case of N classes, we need N discriminant functions $y_1(\mathbf{x}), y_2(\mathbf{x}), \ldots, y_N(\mathbf{x})$, such the k-th function $y_k(\mathbf{x})$ discriminates class C_k from all other classes, for k=1, …, N, i.e.,

  x is assigned to class C_k iff $y_k(x) > y_n(x)$ for all $k \neq n$

- Examples:
  - $y_k(x)=P(C_k|x)$ (posterior probability)
  - $y_k(x)=P(x|C_k)*P(C_k)$ (normalization not needed)
  - $y_k(x)= \ln(P(x|C_k)+\ln P(C_k)$ (monotonic transformation doesn't affect final decisions!!!)

- In case of two classes (C_1 and C_2) we often use $y(x)=y_1(x)-y_2(x)$ as a discriminant; then the sign of y(x) decides on the class:

  if $y(x)>0$ then x in C_1 else x in C_2

Linear Discriminant Functions

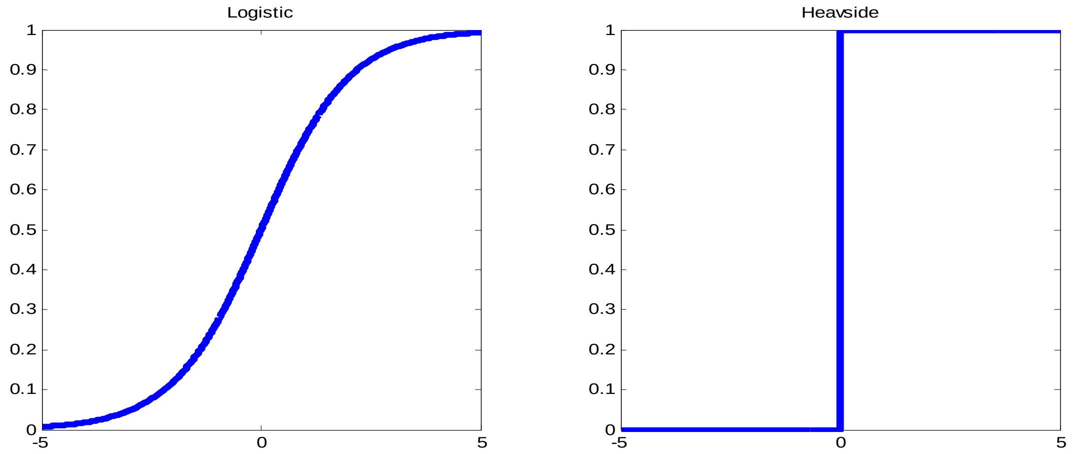
A *linear discriminant function in n-dimensional space* is a function $y(\mathbf{x})$ *in the form:*

$$y(\boldsymbol{x})=w_0+w_1x_1+\ldots+w_nx_n = w_0+w^T\boldsymbol{x}$$

It defines a point (in R^1), a line (in R^2), a plane (in R^3), a hyperplane (in R^n), that splits the space into “positive” and “negative” half-spaces.

A composition of a linear discriminant $y(x)$ with a monotonic function $g(a)$ also defines a linear boundary: $g(y(x))>0$ iff $y(x)>g^{-1}(0)$.

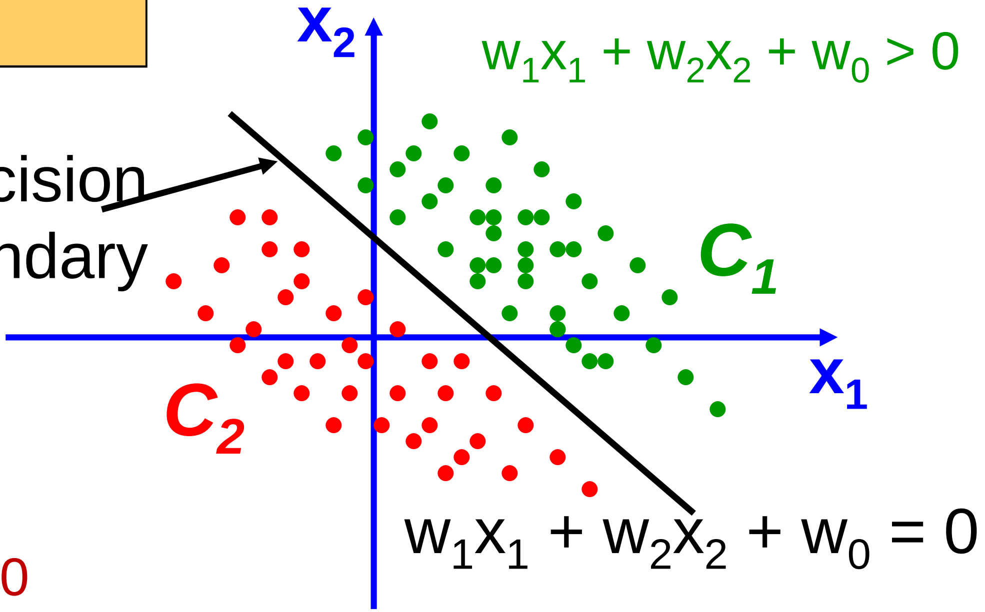
Common choices of g(a) are: $g(a)=1/(1+\exp(-a))$ or $g(a)=0$ for $a<0$; 1 otherwise



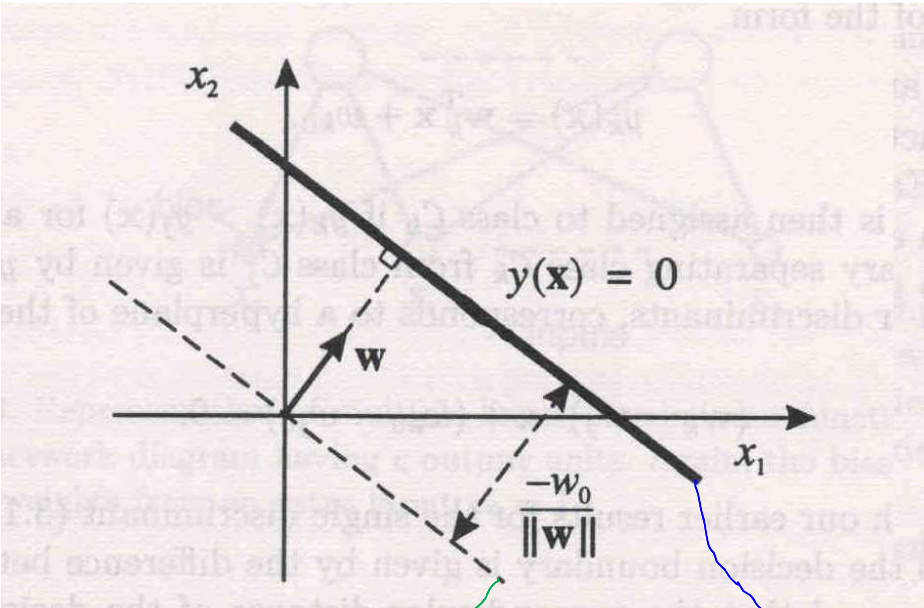
Geometric View in 2D

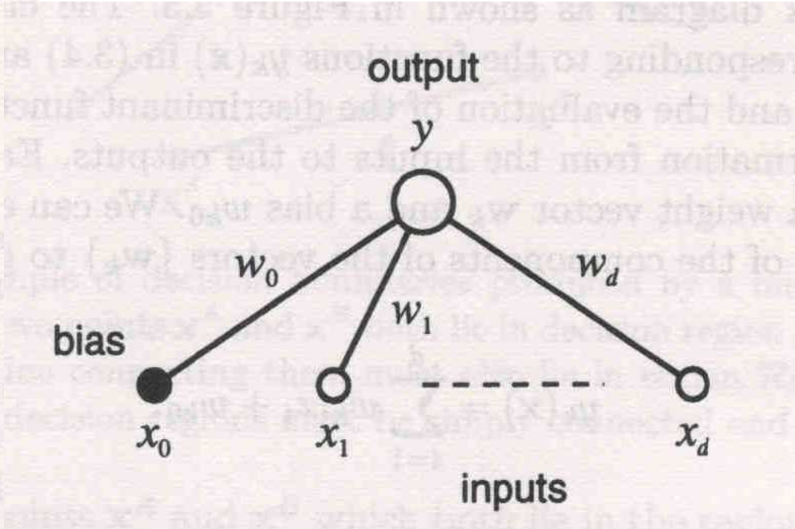
The equation below describes a (hyper-)plane in the input space consisting of real valued 2D vectors. The plane splits the input space into two regions, each of them describing one class.

$$\boxed{\sum_{i=1}^{2} w_i x_i + w_0 = 0}$$



Linear Discriminant and Perceptron





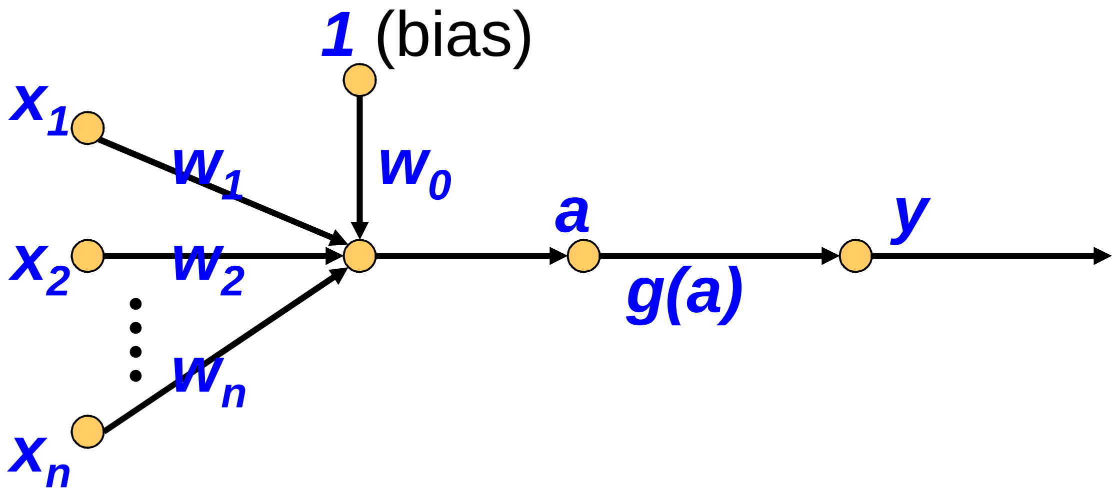
$w_1x_1 + w_2x_2 = 0$
$\langle w_1, w_2 \rangle$ is *perpendicular* to the boundary!

$w_1x_1 + w_2x_2 + w_0 = 0$

Perceptron: McCulloch-Pitts Model

The (McCulloch-Pitts) **perceptron** is a single node NN (or a single layer NN) with a non-linear function **g(a)**:

$$a = w_0 + \sum w_i x_i; \qquad g(a) = \begin{cases} +1 \text{if } a \geq 0 \\ -1 \text{if } a < 0 \end{cases}$$



Perceptron Training

- How can we train a perceptron for a classification task?

- We try to find suitable values for the weights in such a way that the training examples are correctly classified.

- Geometrically, we try to find a hyper-plane that separates the examples of the two classes.

- Two classes C_1 and C_2 are *linearly separable* if there exists a hyperplane that separates them.

Perceptron learning algorithm

initialize **w** randomly;

while (there are misclassified training examples)

Select a misclassified example ($\mathbf{x}$,d)

$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta d\mathbf{x}$;

end-while;

$\eta>0$ is a learning rate parameter (step size);

Motivation:

If x missclassified and d=1 => **wx** should be bigger,

If x missclassified and d=-1 => **wx** should be smaller

$$(w+\eta dx)x = wx + \eta dx^2$$

This rule does exactly what we want (x^2 is >0) !!!

Conventions:

- w is a row vector;
- x is a column vector
- x^2 is $x^T * x$

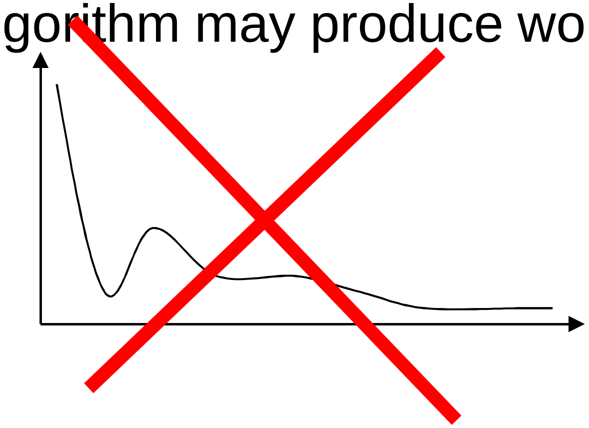
Main properties

1) Perceptron Convergence Theorem:

If the classes C_1, C_2 are linearly separable (that is, there exists a hyper-plane that separates them) then the perceptron algorithm applied to $C_1 \cup C_2$ terminates successfully after a finite number of iterations. [*the value of the learning rate η is not essential*]

2) Bad Behavior:

If the classes C_1, C_2 are **not** linearly separable then the perceptron algorithm may produce worse and worse results:



Improvement: (Naive) Pocket Algorithm

0. Start with a random set of weights;
   put them in a 'pocket'
1. Select at random a pattern
2. If it is misclassified then

   2A. apply the perceptron update rule

   2B. check whether new weights are better than those kept in the pocket; if so put new weights to the pocket (and remove the old ones)

3. goto 1.

Testing if new weights are better (less errors) is very expensive !!!

Gallant's Pocket Algorithm

main idea: measure the quality of weights by counting the number of <u>*consecutive correct classifications*</u> ('current_run')

0. Start with a random set of weights; put them in the 'pocket';

   ***best_run* :=0**; ***current_run* :**=0;

1. Select a pattern at random

2. If it is misclassified then

   apply the update rule; ***current_run* :**=0;

   else

   current_run++;
   if ***current_run*** > ***best_run*** then
   put new weights to the pocket; ***best_run* :=*current_run***

3. Goto 1

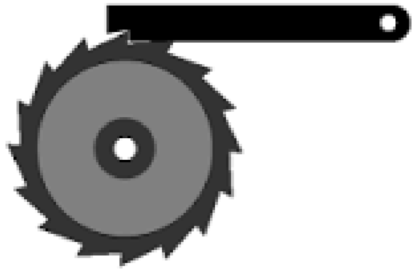
Gallant's Pocket Algorithm with ratchet

0. Start with a random setting of weights; put it in the 'pocket'

1. ***best_run:=0***; ***current_run:=0***;

2. Select at random a pattern

3. **IF** it is correctly classified **THEN**



current_run:=current_run+1

IF ***best_run<current_run*** **AND** current weights are better than those kept in the pocket **THEN**

pocket:=current weights; ***best_run=current_run;***

ELSE

current_run=0;

update weights

Pocket convergence theorem

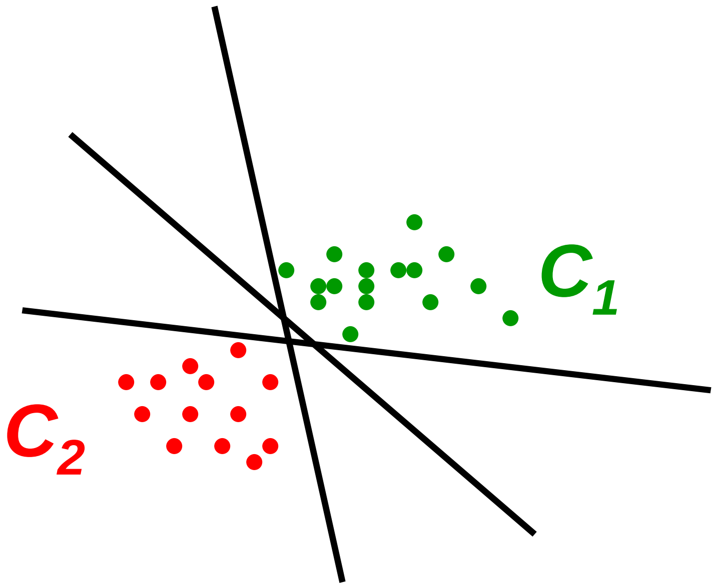
Pocket Convergence Theorem:

The pocket algorithm converges with probability 1 to optimal weights (even if sets are not separable!)

Practice:

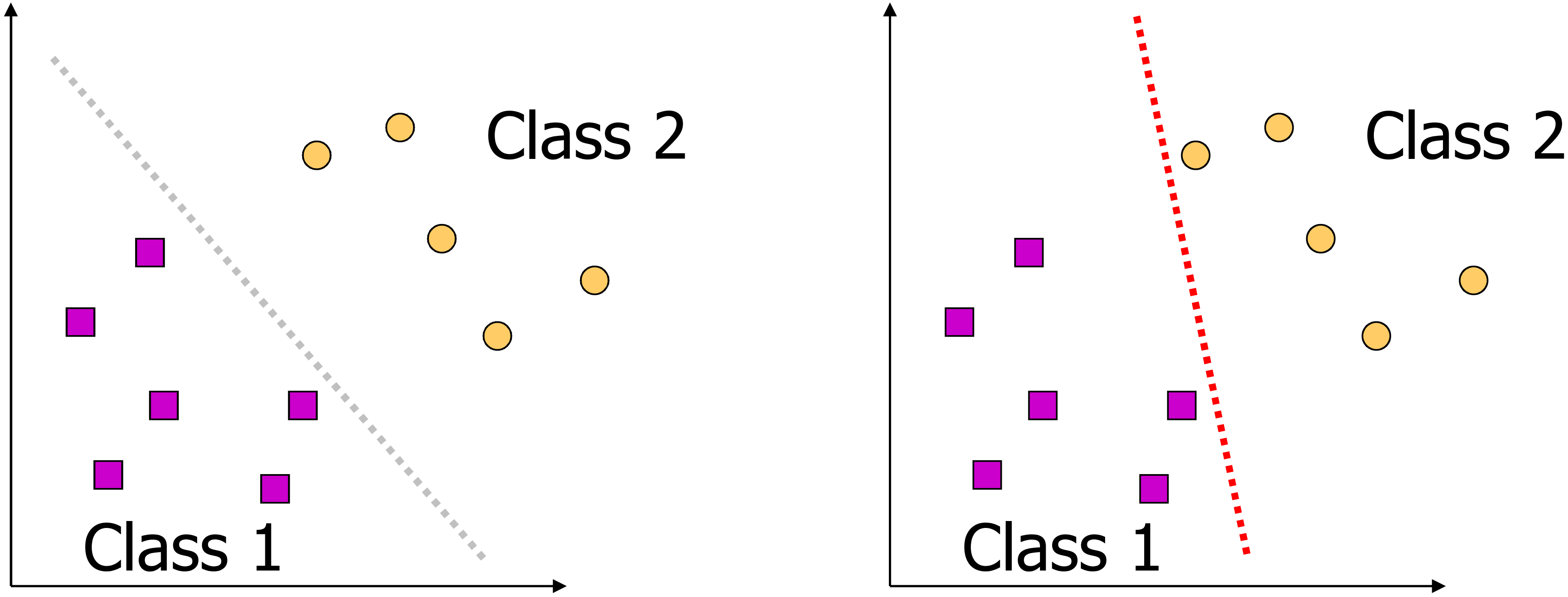
1) The convergence rate is quite high
2) Pocket algorithm is better than the Perceptron algorithm
3) Both algorithms have very limited use
4) There may be many separating hyperplanes ...

Which separating hyperplane is best?



We are interested in accuracy on the test set!

Examples of Bad Decision Boundaries



Idea:

define a “continuous error measure” and try to minimize it !

Cover's Theorem (1965):

What is the chance that a **randomly labeled set of N points** (in general position) **in d-dimensional space, is linearly separable?**

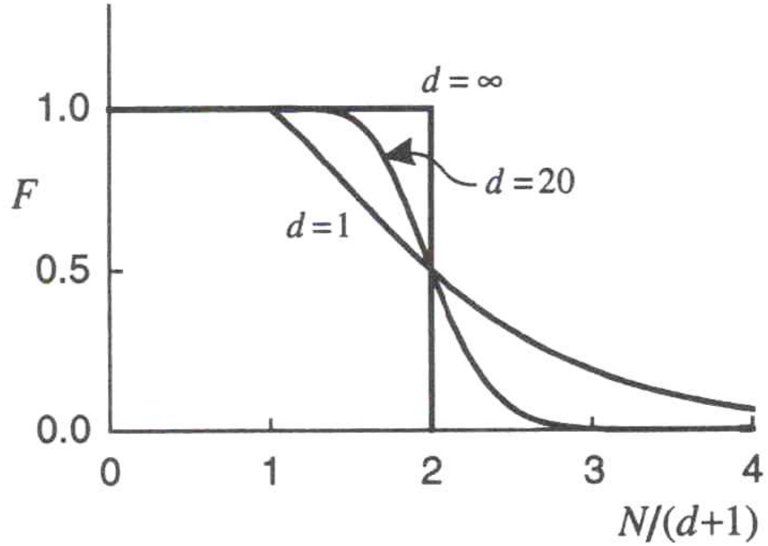


Figure 3.7. Plot of the fraction $F(N, d)$ of the dichotomies of N data points in d dimensions which are linearly separable, as a function of $N/(d+1)$, for various values of d.

$$F(N,d) = \begin{cases} 1 & \text{when } N \leq d+1 \\ \dfrac{1}{2^{N-1}} \displaystyle\sum_{i=0}^{d} \binom{N-1}{i} & \text{when } N \geq d+1 \end{cases} \tag{3.30}$$

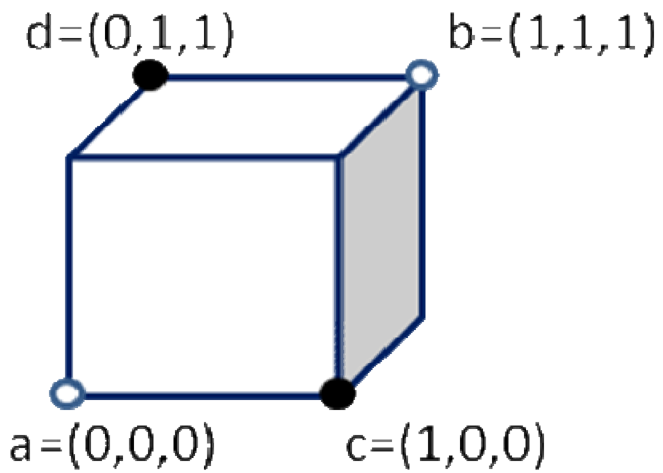
Cover's Theorem in highly dimensional spaces:

1) if the number of points in d-dimensional space is smaller than 2*d then they are almost always linearly separable

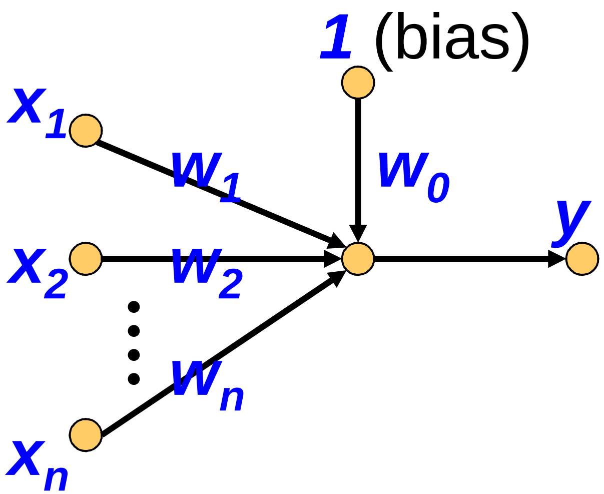
2) if the number of points in d-dimensional space is bigger than 2*d then they are almost always linearly non-separable

A quick check:

Are points {a, b} linearly separable from {c, d}?
How does it relate to the Cover's theorem?



Adaline: Adaptive Linear Element



$$y = w_0 + \sum w_i x_i$$

- Activation function g(x)=x (identity)
- The desired outputs are -1's or 1's, the actual outputs (y's) are "real numbers"
- Main idea: minimize the squared error:

$$Error = \sum_{Examples} (y_i - d_i)^2$$

Error function

on pattern $\boldsymbol{i}$: $E(i)=(d_i-y_i)^2$

on all patterns: $E= \Sigma(d_i-y_i)^2$

But

$$y_i = w_0+w_1x_1+ \ldots +w_kx_k,$$

so

$$E=\Sigma(d_i-(w_0+w_1x_1+ \ldots +w_kx_k))^2$$

thus

$\boldsymbol{E}$ is a function of $w_0,w_1, \ldots, w_k$.

How can we find the minimum of $E(w_0,w_1, \ldots ,w_k)$?

By the gradient descent algorithm ...

Gradient Descent Algorithm

How to find a minimum of a function $f(x,y)$?

1. Start with an arbitrary point (x_0, y_0)

2. Find a direction in which f is decreasing most rapidly

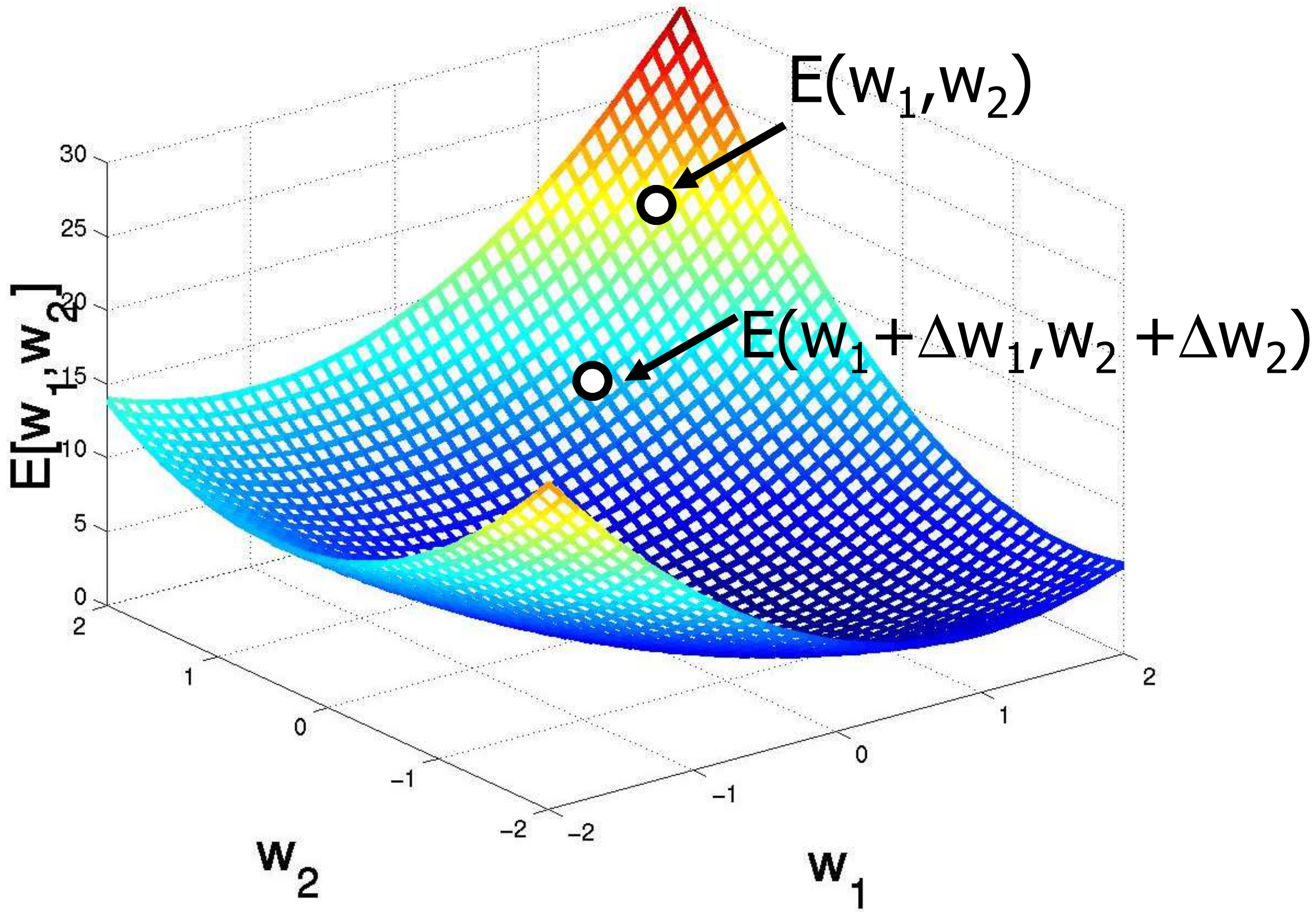
$$-\left[\frac{\partial f(x_0, y_0)}{\partial x}; \frac{\partial f(x_0, y_0)}{\partial y}\right]$$

3. Make a small step in this direction

$$(x_0, y_0) = (x_0, y_0) - \eta\left[\frac{\partial f(x_0, y_0)}{\partial x}; \frac{\partial f(x_0, y_0)}{\partial y}\right]$$

4. Repeat the whole process

Gradient Descent



Adaline: Gradient Descent

- Find w_i's that minimize the squared error

  $E(w_0,\ldots,w_m) = \Sigma(d-y)^2$

- Gradient:

  $\nabla E[w]=[\partial E/\partial w_0,\ldots\ \partial E/\partial w_m]$; $\Delta w=-\eta\ \nabla E[w]$

  $\partial E/\partial w_i = \partial/\partial w_i\ \Sigma(d-y)^2 = \partial/\partial w_i\ \Sigma(d-\Sigma_i\, w_i\, x_i)^2 = 2\ \Sigma(d-y)(-x_i)$

  (summation over all examples)

- The weights should be updated by: $w_i=w_i + \eta\Sigma(d-y)x_i$

- Summation over all cases? Split the summation by examples! I.e., for each training example, $w_i=w_i + \eta(d-y)x_i$

Incremental versus Batch Learning

- Batch mode : gradient descent

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

  $E_D[w] = \Sigma_d (t_d - o_d)^2$

- Incremental mode: gradient descent

  $w = w - \eta \nabla E_d[w]$ over individual training examples d: $E_d[w] = (t_d - o_d)^2$

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

Adaline Training Algorithm

1. start with a random set of weights $\boldsymbol{w}$
2. select a pattern $\boldsymbol{x}$ (e.g., at random)
3. update weights:

$$\boldsymbol{w} := \boldsymbol{w} + \eta \boldsymbol{x}(\boldsymbol{d} - \boldsymbol{y}), \ \textit{(Adaline rule)}$$

where $\boldsymbol{d}$ - desired output on input $\boldsymbol{x}$ and $\boldsymbol{y} = \boldsymbol{wx}$

4. goto 2

The step size η should be relatively small

Adaline and Linear Regression

- Perceptron can be used for classification problems only
- Adaline doesn't require that outputs are -1's or 1's - arbitrary values are allowed
- Therefore Adaline can be used for solving Linear Regression Problems
- There is a direct (fast) algorithm for Linear Regression!
- **But:** Adaline requires no memory: it "learns on-the-fly";
  it's biologically justified (?)

Linear Regression

- Assume

$$y=w_0+w_1x_1+w_2x_2+\ ...\ +w_nx_n$$

- Find values for $w_0, \ldots, w_n$ for which the squared error:

$$error=(d_1-y_1)^2+(d_2-y_2)^2+\ ...\ +(d_k-y_k)^2$$

  is minimized

  $d_1, \ldots, d_k$ are desired values,

  $y_1, \ldots, y_k$ are predictions

Finding Linear Regression Model: Least Squares Method

- Error function is a quadratic function of n+1 variables:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

$$error = (d_1 - y_1)^2 + (d_2 - y_2)^2 + \ldots + (d_k - y_k)^2$$

$$error(\mathbf{w}) = (d_1 - y_1(\mathbf{w}))^2 + (d_2 - y_2(\mathbf{w}))^2 + \ldots + (d_k - y_k(\mathbf{w}))^2$$

- all partial derivatives=0 ==> error is minimal
- partial derivatives are linear functions of ***w***

⇓

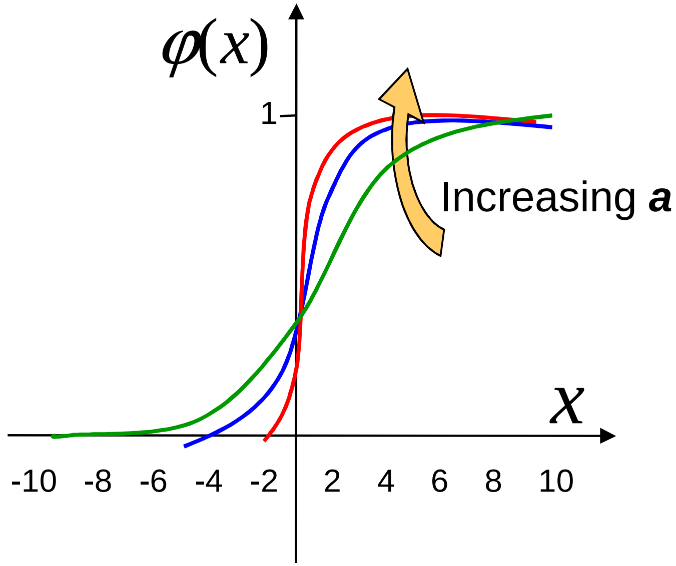
finding optimal *w* reduces to the problem of solving

a system of (*n+1)* linear equations with (*n+1)* variables (no matter how big the training set)

"Smooth Perceptron" =>Logistic Regression

- Main Idea:
  Replace the sign function by its "smooth approximation" and use the steepest descent algorithm to find weights that minimize the error (as with ADALINE)

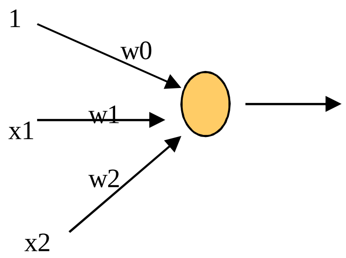
$$\varphi(\mathrm{x}) = \frac{1}{1+e^{-ax}} \quad \text{with} \quad a > 0$$



- The function to be optimized is a bit more complicated than in ADALINE case
- FORTUNATELY: the update rules are simple !

Derivation of update rules for simple net

- Derive the Delta rule for the following network



$$y = \varphi(net) = \varphi(w_0 + w_1 x_1 + w_2 x_2)$$

(*net* = linear combination of inputs)

$$E(w_0, w_1, w_2) = \frac{1}{2}(y - d)^2 = \frac{1}{2}(\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)^2$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \text{ for } i \text{ in } \{0,1,2\}$$

- We need to find $\dfrac{\partial E}{\partial w_0}, \dfrac{\partial E}{\partial w_1}, \dfrac{\partial E}{\partial w_2}$

Derivation of Delta Rule

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_1} = \frac{1}{2} \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)^2}{\partial w_1}$$

$$= \frac{1}{2} 2(\varphi(w_0 + w_1 x_1 + w_2 x_2) - d) \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)}{\partial w_1}$$

$$= (\varphi(net) - d)\varphi'(net) \frac{\partial (w_0 + w_1 x_1 + w_2 x_2)}{\partial w_1}$$

$$= (\varphi(net) - d)\varphi'(net) x_1$$

- From similar calculations we get:

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_2} = (\varphi(net) - d)\varphi'(net) x_2$$

- and $$\frac{\partial E(w_0, w_1, w_2)}{\partial w_0} = (\varphi(net) - d)\varphi'(net)$$

Concluding:

$$\Delta w_0 = \eta(d - \varphi(net))\varphi'(net)$$

$$\Delta w_1 = \eta(d - \varphi(net))\varphi'(net)x_1$$

$$\Delta w_2 = \eta(d - \varphi(net))\varphi'(net)x_2$$

$$\Delta \mathbf{w} = \eta(d - \varphi(net))\varphi'(net)\mathbf{x}$$

- It's good to know that for the logistic sigmoid function: φ(x)=*1/(1+exp(-x))* we have: φ′(x)=φ(x)(1-φ(x))=*output(1-output)*
- *Adaline: Linear Regression*
- *"A Neuron": "Logistic Regression" (what is the error function?)*

Logistic Regression

- A single neuron is equivalent to logistic regression: y=g(**wx**+w_0), where g(a)=1/(1+exp(-x))

- In case P(x|C_1) and P(x|C_2) can be modeled by Gaussians with the same matrix Σ, the logistic regression models the posterior probability:

$$P(C_1 \mid x) = \frac{p(x \mid C_1)P(C_1)}{p(x \mid C_1)P(C_1) + p(x \mid C_2)P(C_2)} =$$

$$= \frac{1}{1+\exp(-a)} = g(a), \text{ where } \; a = \ln \frac{p(x \mid C_1)P(C_1)}{p(x \mid C_2)P(C_2)}$$

Summary of learning rules:

Perceptron learning rule:

$$\Delta\mathbf{w} = \eta*\mathbf{x}*(\mathbf{d}-\mathbf{out})\ (\textbf{for misclassified})$$

x = input vector
out = output of the network;
out = *f(net)*, where:
net = linear comb. of inputs

Perceptron: *f(net) = sign(net)*

Adaline : *f(net) = net*

"Neuron": *f(net) = 1/(1+exp(-net))*

Adaline learning rule:

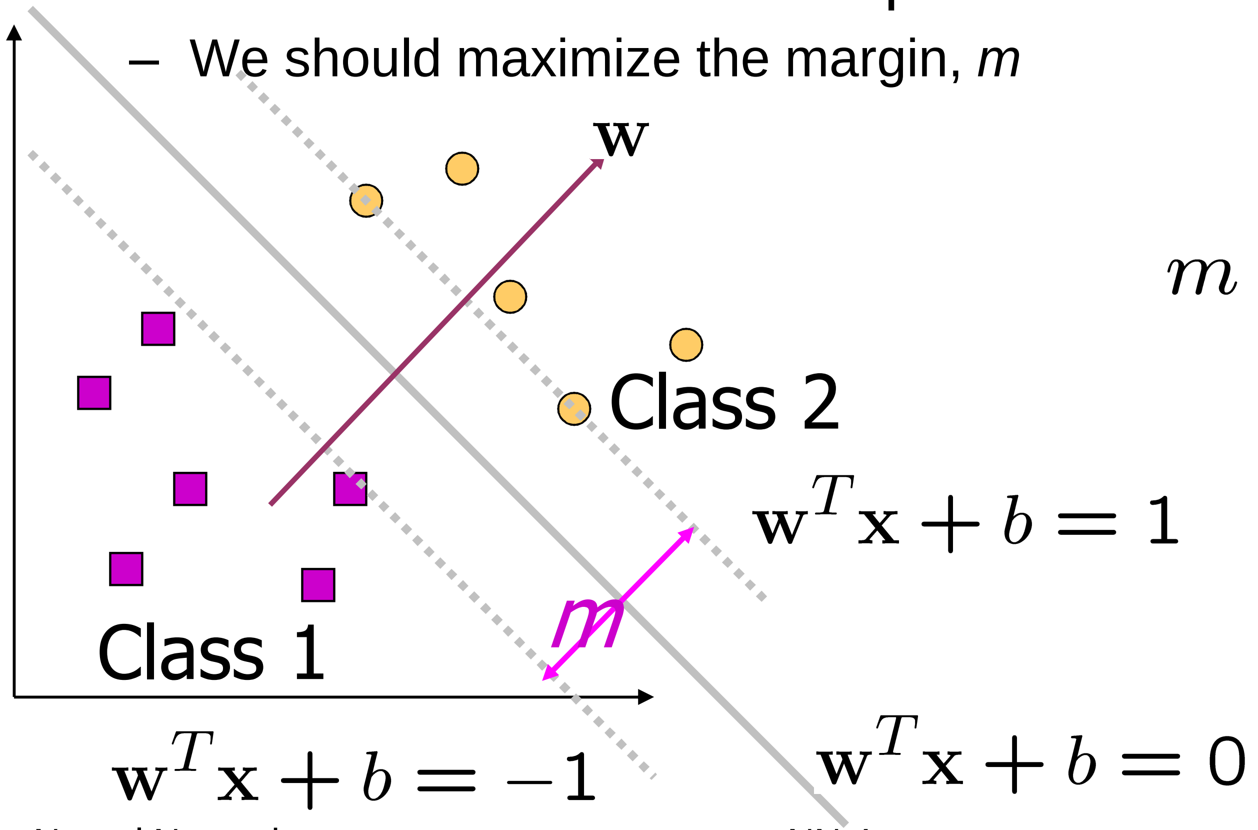
$$\Delta\mathbf{w} = \eta*\mathbf{x}*(\mathbf{d} - \mathbf{out})$$

"A Neuron" learning rule:

$$\Delta\mathbf{w} = \eta*\mathbf{x}*(\mathbf{d} - \mathbf{out})*\mathbf{out}'(\mathbf{x})$$

A yet another approach: Support Vector Machines (SVM)

- The decision boundary should be as far away from the data of both classes as possible
  - We should maximize the margin, *m*



$$m = \frac{2}{||\mathbf{w}||}$$

Finding the Decision Boundary

- Let $\{x_1, ..., x_n\}$ be our data set and let $y_i \in \{1,-1\}$ be the class label of x_i
- The decision boundary should classify all points correctly $\Rightarrow$ $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad \forall i$
- The decision boundary can be found by solving the following constrained optimization problem:

$$\text{Minimize } \frac{1}{2}||\mathbf{w}||^2$$
$$\text{subject to } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \quad \forall i$$

- A Quadratic Programming Optimization Problem (not so easy, but doable…)
- Support Vectors: the training points that are nearest to the separating hyperplane.

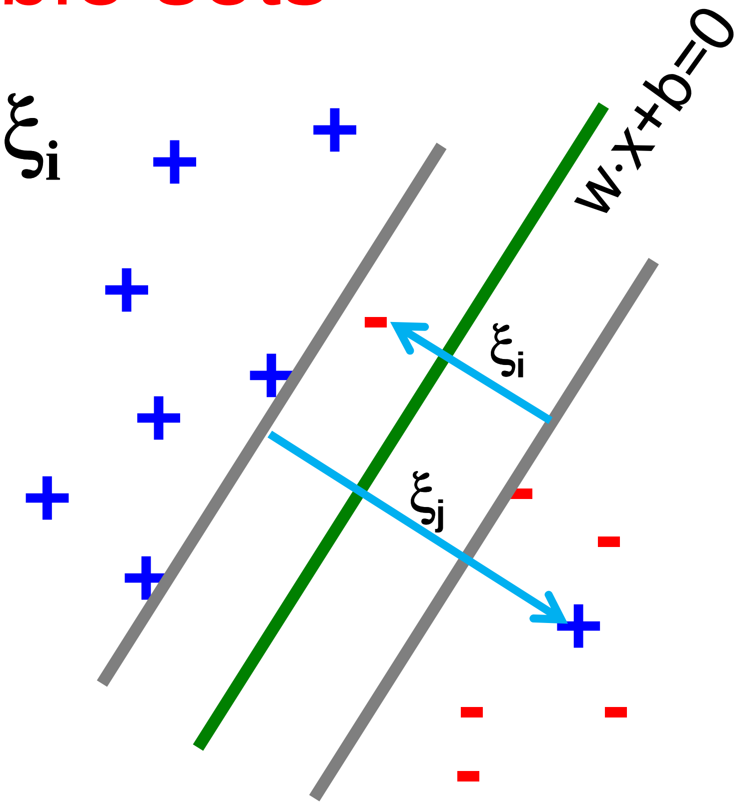
SVM for non-separable sets

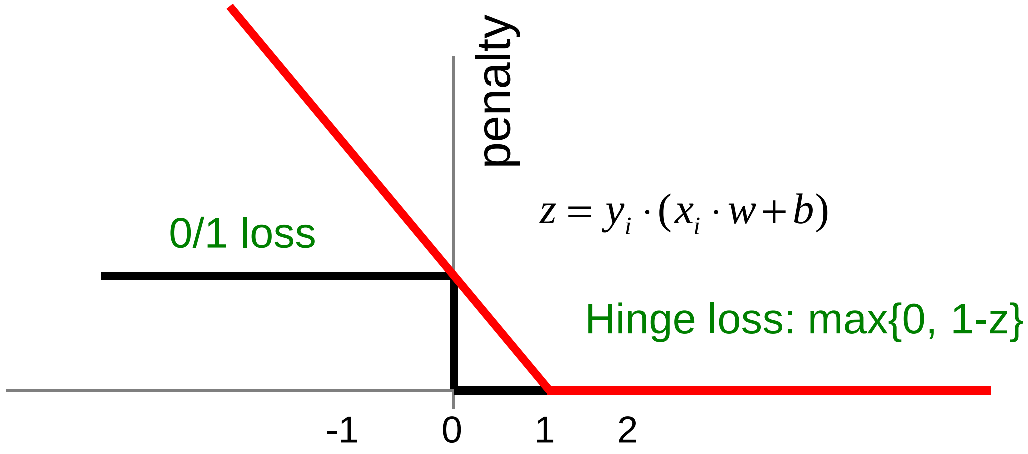
- **Introduce** **slack variables** ξ_i

$$\min_{w,b,\xi_i \geq 0} \frac{1}{2}\|w\|^2 + C \cdot \sum_{i=1}^{n} \xi_i$$

$$s.t. \forall i, y_i (w \cdot x_i + b) \geq 1 - \xi_i$$

- If point x_i is on the wrong side of the margin then it gets penalty ξ_i





For each data point:
If margin ≥ 1, don't care
If margin < 1, pay linear penalty
(can be expressed by linear cons.)

Support Vector Machines

- State-of-the-art classification procedure
- Performs very well on highly dimensional data (character recognition, text mining, bioinformatics)
- Computationally very expensive (quadratic programming) an alternative: *Stochastic Gradient Descend (approximated)*
- *Generalization to non-linear boundaries* (with help of kernel functions), *and more than 2 classes*
- Not mentioned in Bishop's book (too recent!)
- *Covered in the MMDS textbook, Ch. 12* (http://www.mmds.org/)

Learning non-linear boundaries with linear models

- How could we train a single perceptron to separate the interior of a circle from its exterior?

- It's easy: use as inputs x^2 and y^2: the discriminant function:

  $y(x^2, y^2)=x^2+y^2-r^2$ linear in its arguments!

- When using as input x, y, x^2 and x^2 we could even "learn" an ellipse!

- What about using x, y, x^2, x^2 and xy? What could we model then?

- Generalized Linear Discriminants:

$$y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj} \Phi_j(\mathbf{x}) + w_{k0}$$

- The basis functions $\Phi_j(\mathbf{x})$ are fixed (e.g., polynomials, radial functions)

Generalized Linear Discriminants

- Allowing non-linear basis functions as inputs for linear models dramatically increases the power of these models: in theory such networks can model any boundary (or function), provided we use an appropriate set of basis functions

⇓

Linear Models (Single Layer Networks) are very important!

- An important class of basis functions are "radial functions" that measure the distance of x to specific "reference points" x_k. This leads to the concept of Radial Basis Function Networks (RBF-networks)

- RBF networks will be discussed later …

Perceptron for multi-class problems

- So far, we were considering binary classification problems: how to separate two sets of points with a (linear) model? So we were looking for a single (linear) discriminant function…

- Multi-class classification problems: we want to separate c>2 sets of points

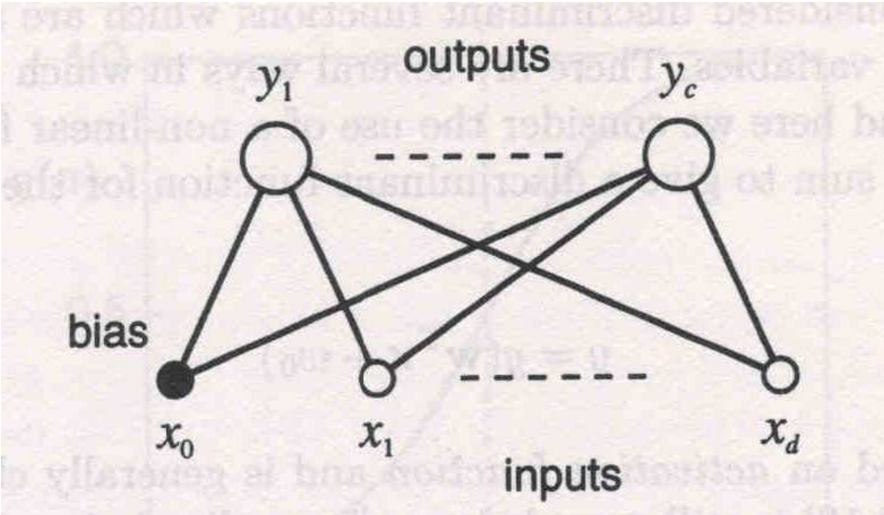
Linear Separability for multi-class problems:

There exist ***c linear discriminant functions*** $y_1(x), \ldots, y_c(x)$ such that each $\mathbf{x}$ is assigned to class C_k if and only if $y_k(x) > y_j(x)$ for all $j \neq k$

All algorithms discussed so far can be generalized to handle multi-class classification problems. In some cases the generalization is easy, in others not.

Generalized Perceptron convergence theorem:
If the c sets of points are linearly separable then the generalized perceptron algorithm terminates after a finite number of iterations, separating all classes.

Generalized Perceptron Algorithm (Duda et al.)



initialize weights **w** at random

while (there are misclassified training examples)

Select a misclassified example **(x, c_i)**

Then *some nodes* are activated more than *the node c_i*

1) update weights of *these nodes* by **-x: w = w - x;**

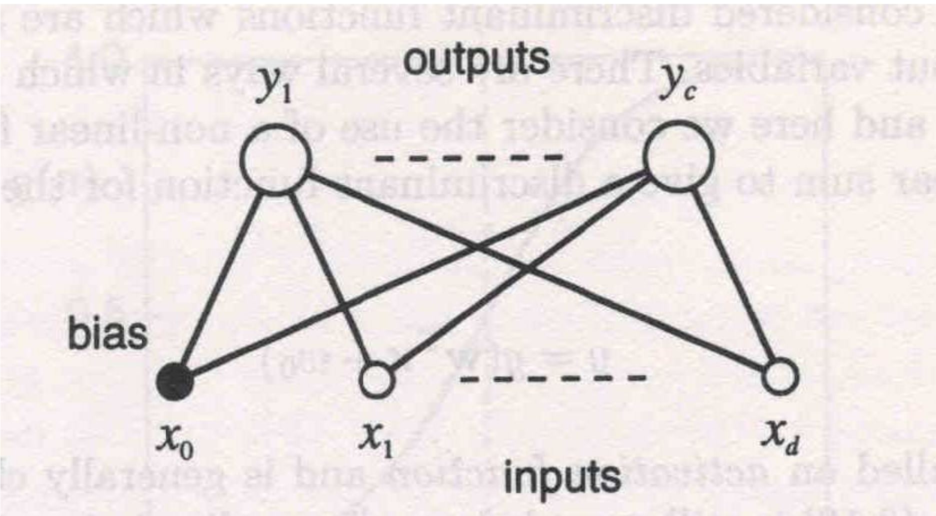
2) update weights of the node **c_i** by **x: w = w + x;**

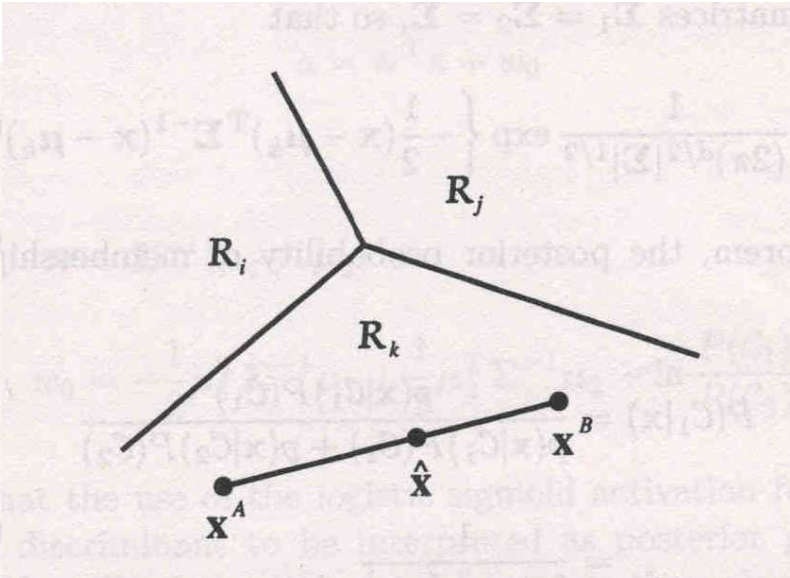
3) leave weights of all other nodes unchanged

end-while;

Perceptron for multi-class problems

A network of c perceptrons that share the same input vector represent c linear discriminant functions and can be used for solving multi-class classification problems.





Note that $y_i - y_j$ is a linear function which separates class *i* from *j, for all i, j*

So decision regions are intersections of half-spaces!

Decision regions are always ***convex***: for any two points x^A and x^B from the same region, the whole line interval between x^A and x^B is also in this region.

To Remember

- Discriminant functions, linear discriminants, linear separability, Cover’s theorem (the plot and the interpretation!)
- The perceptron learning algorithm and key properties: convergence and bad behaviour on non-separable data sets
- The pocket algorithm (also with ratchet) of Gallant
- Adaline and (logistic) Perceptron; Incremental vs Batch Learning
- Derivation of learning rules for Adaline and Perceptron
- The concept of SVM; quadratic optimization criterion
- Generalized Linear Discriminant
- Multi-class linear separability
- The generalized perceptron algorithm