Linear discriminant functions

The Perceptron Algorithm

Least squares Algorithm

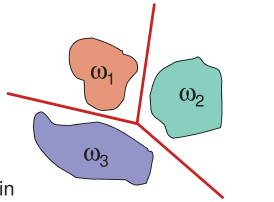
Logistic Regression

Summary of previous episodes

- **In the previous chapters, we have seen how we can**
  - Estimate parametric or non-parametric probability density functions in a supervised framework;
  - *Select* or *extract* a subset of pertinent features for classification;
  - Perform an unsupervised classification, i.e. from unlabeled data.
- **Most of the classification methods we studied so far where introduced in the Bayesian framework,**
  - i.e. based on the knowledge of underlying probabilities.
- **What can be done were these probabilities are too complex or/and difficult to estimate?**
- **Is it possible to directly set up decision surfaces by doing some hypotheses on their shape (linear or not)?**

Introduction

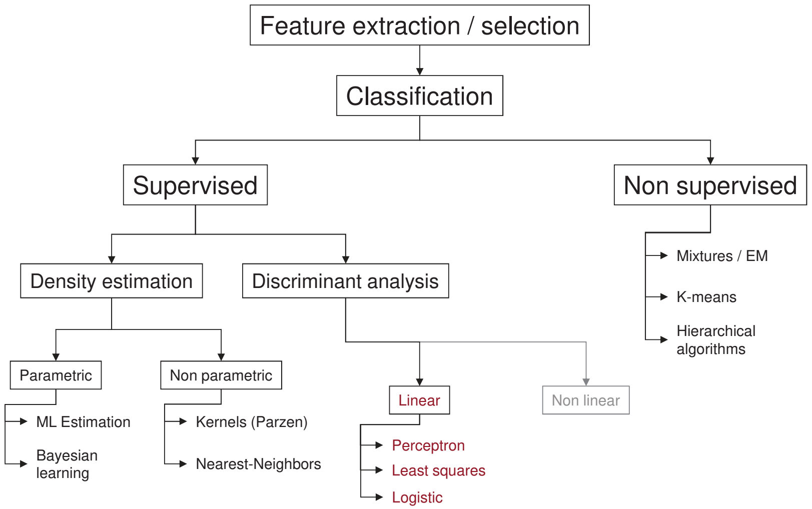
- **In the present chapter :**
  - We do not assume anything about the underlying probabilities.
  - But we suppose that the classes are <u>linearly separable</u>
  - We consider supervised learning, i.e. we have in hand a set of labeled samples, $\mathcal{D} = \{\mathbf{x}_k, t_k\}_{k=1 \ldots N}$



Following [Guttierez]

- **<u>Directly</u> estimate the parameters of the discriminant function?**
  - Once again, the problem is set up as an optimization problem
    - ✓ Numerous criteria were proposed, several algorithms exist
  - We will study three main approaches :
    - ✓ the Perceptron, least squares and logistic regression algorithms.
  - We will first consider the 2-class case (dichotomy), then we will study the generalization to C classes.

A hierarchy of methods



Linear discriminant: classification rule (C=2)

- **Linear discriminants implement a classification rule of the following kind**

$$g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 \underset{\omega_2}{\overset{\omega_1}{\gtrless}} 0 \quad \text{or} \quad \mathbf{a}^T\mathbf{y} \underset{\omega_2}{\overset{\omega_1}{\gtrless}} 0 \quad \text{with:} \quad \mathbf{y} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}$$

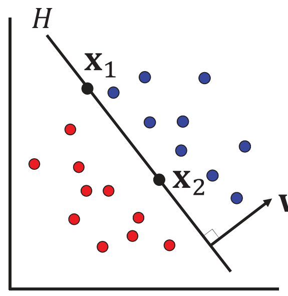
  - In this expression, $\mathbf{w}$ is called the *weight vector* and w_0 is the *bias* or threshold weight. These are the parameters we have to optimize
  - We may gather them in a vector, $\mathbf{a}$ and consider $\mathbf{y}$, an *augmented* feature vector
  - NB: we might also consider $y = [1, \phi(\mathbf{x})]^T$ and apply the following methodologies in a transformed space... We will go back to this later on!

Graphical representation *[Duda, p. 217]*

- **If $\mathbf{x}_1$ and $\mathbf{x}_2$ both lie on the separating hyper-plane, $g(\mathbf{x}) = 0$, then:**

$$\mathbf{w}^T\mathbf{x}_1 + w_0 = 0 = \mathbf{w}^T\mathbf{x}_2 + w_0 \quad \Rightarrow \quad \mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2) = 0$$

➔ The separating hyper-plane H is perpendicular to $\mathbf{w}$



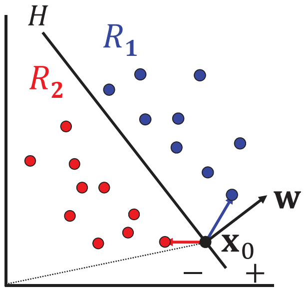
Graphical representation *[Duda, p. 217]*

- **The separating hyper-plane H is perpendicular to $\mathbf{w}$**
- **Take $\mathbf{x}_0$ on H**

$$g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 = \mathbf{w}^T(\mathbf{x} - \mathbf{x}_0) + \underbrace{\mathbf{w}^T\mathbf{x}_0 + w_0}_{0,\, since\ \mathbf{x}_0 \in H} = \mathbf{w}^T(\mathbf{x} - \mathbf{x}_0)$$

➔ **H splits feature space in two decision regions, R_1 for ω_1 and R_2 for ω_2**

- Since g is positive for $\mathbf{x} \in R_1$, the vector $\mathbf{w}$ points towards R_1 : $\mathbf{x}$ lies on the positive side of H.



Graphical representation *[Duda, p. 217]*

- **The separating hyper-plane H is perpendicular to $\mathbf{w}$**
- **H splits feature space in two decision regions, R_1 for ω_1 and R_2 for ω_2, $\mathbf{w}$ points towards $R1$**
- **Now, consider the orthogonal projection $\mathbf{x}_p$ of $\mathbf{x}$ on H**
  - We may write

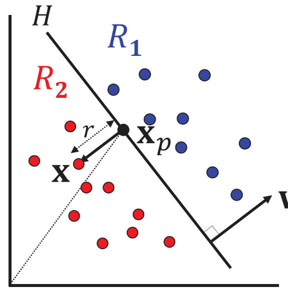
$$\mathbf{x} = \mathbf{x}_p + r\frac{\mathbf{w}}{\|\mathbf{w}\|}$$

where r is the signed distance to H. We have that

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

$$g(\mathbf{x}) = \underbrace{\mathbf{w}^T\mathbf{x}_p + w_0}_{0,\, since\ x_p \in H} + r\frac{\mathbf{w}^T\mathbf{w}}{\|\mathbf{w}\|} = r\|\mathbf{w}\|$$

  - Discriminant function ∝ signed distance r from $\mathbf{x}$ to H



Graphical representation *[Duda, p. 217]*

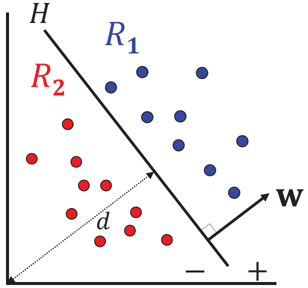
- **The separating hyper-plane H is perpendicular to w**
- **H splits feature space in two decision regions, R_1 for ω_1 and R_2 for ω_2, w points towards $R1$**
- **The discriminant function measures the signed distance r from x to H**

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

- **The position of H is determined by the threshold (or bias) w_0.**

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

  - $w_0 > 0$ if the origin is on the positive side of H.
- **Classification according to $sign(g(\mathbf{x}))$**
- **Learning : estimate w and w_0**

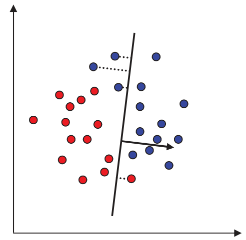


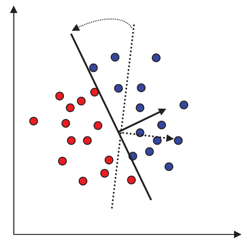
Linear discriminant: learning

- **Estimate direction, w and bias, w_0**

➔ **Design a criterion and optimize it**

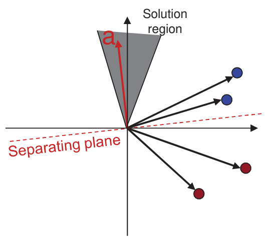
- **Some existing criteria**
  - *Fisher LDA* (already studied)
  - Perceptron (related to Neural Nets)
  - Least Squares
  - Logistic regression
  - SVM (see chapter later)
- **Optimization**
  - Most of the time, iterative and deterministic





Solution region

- **In the space of weights: $\mathbf{a}^T\mathbf{y}_k = 0$**
  - Every training sample $\mathbf{y}_k$ defines a hyper-plane passing through the origin and orthogonal to $\mathbf{y}_k$.
  - The solution lies on the positive side of all hyper-planes for $\mathbf{y}_k \in \omega_1$ and on the negative side for $\mathbf{y}_k \in \omega_2$: their intersection defines a region in which any vector is solution.
  - The boundaries of the solution region depend on the closest samples to the separating hyperplane



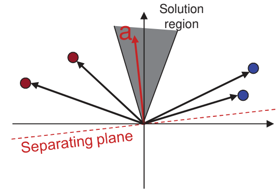
Following [Duda]

➔ **The solution may not be unique**

« Normalization »

- **« Normalization »:**
  - Correct classification iff $\begin{cases} \mathbf{a}^T\mathbf{y}_k > 0 \text{ and } \mathbf{x}_k \in \omega_1 \\ \mathbf{a}^T\mathbf{y}_k < 0 \text{ and } \mathbf{x}_k \in \omega_2 \end{cases}$
  - Simplification: replace $\mathbf{y}_k \in \omega_2$ by its opposite.
  - The problem is then to find $\mathbf{a}$ such that:

$$\boxed{\mathbf{a}^T\mathbf{y}_k > 0, \forall \mathbf{x}_k}$$



Following [Duda]

- **NB: Normalization amounts to multiplying each (augmented) sample by its label t_k, provided that $t_k \in \{-1,1\}$**

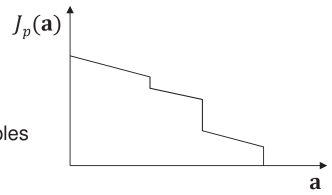
$$\boxed{t_k(\mathbf{w}^T\mathbf{x}_k + w_0) > 0} \quad \text{or} \quad \boxed{t_k(\mathbf{a}^T\mathbf{y}_k) > 0}$$

The « Perceptron » criterion

- **To estimate a, we first have to define an objective function**
  - It penalize solutions that misclassify data
  - Let us denote the set of misclassified samples by $\mathcal{Y}$
  - The Perceptron criterion is (proportional to) the sum of their absolute distance to the decision boundary:

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

  - J_p is non-negative
  - J_p is zero iff $\mathcal{Y} = \{\emptyset\}$
  - J_p is decreasing and piecewise linear (discontinuities correspond to points where the number of misclassified samples changes)



The Perceptron algorithm

- **J_p is "well suited" to gradient descent**

$$\mathbf{a} \leftarrow \mathbf{a} - \delta \frac{\partial J_p(\mathbf{a})}{\partial \mathbf{a}}$$

where δ may be constant, or not

- **Differentiating J_p is straightforward**

$$\frac{\partial J_p(\mathbf{a})}{\partial \mathbf{a}} = \sum_{\mathbf{y}\in\mathcal{Y}} -t\mathbf{y}$$

  - Note : the gradient is not defined at discontinuities of J_p (hence the ".")

- **Iteration**

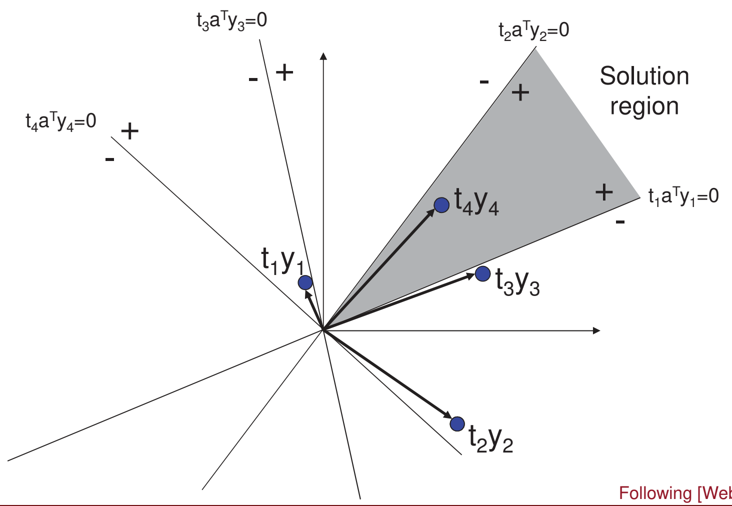
$$\mathbf{a} \leftarrow \mathbf{a} + \delta \sum_{\mathbf{y}\in\mathcal{Y}} t\mathbf{y}$$

The Perceptron algorithm

- **Also called « batch » Perceptron because all the samples are processed at the same time:**
  - Initialize $\mathbf{a}$ and choose δ
  - Iterate
    - ✓ Classify samples according to $t(\mathbf{a}^T y) > 0$ and detect errors ⇨ $\mathcal{Y}$
    - ✓ Form correction term and update $\mathbf{a}$
  - Until convergence

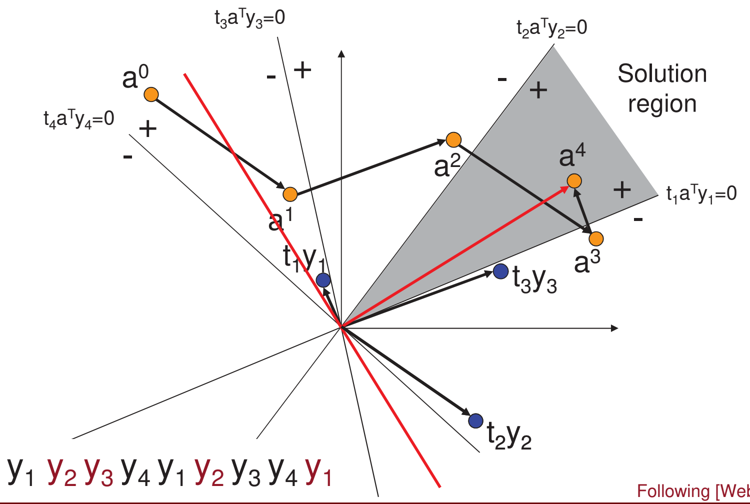
- **Sequential algorithm (variant where samples are processed one by one) with fixed increment**
  - Initialize $\mathbf{a}$ and choose δ
  - Cycle through the list of $\mathbf{y}$'s
    - ✓ If $\mathbf{y}$ is misclassified then $\mathbf{a} \leftarrow \mathbf{a} + \delta\, t\mathbf{y}$
    - ✓ Otherwise leave $\mathbf{a}$ unchanged
  - Until all samples are correctly classified

Graphical interpretation (4 samples)



Following [Webb]

Graphical interpretation (4 samples)



Following [Webb]

The Perceptron in 5 lines of code, $t_k \in \{-1,1\}$

1. **Initialize parameter vector:** $\mathbf{a} = 0$
2. **For** $k = 1 \ldots N$
3. $\quad l_k = sign(\mathbf{a}^T \mathbf{y}_k)$
4. $\quad$ **if** $l_k \neq t_k$ **then**
5. $\qquad \mathbf{a} = \mathbf{a} + \boldsymbol{\delta}\, t_k\, \mathbf{y}_k$

Repeat:
- **Until convergence**
- **For a number of *epochs***

- **Loop invariant: a is a weighted sum of training samples**
  - So, $\mathbf{a} = \sum_k \alpha_k\ t_k\ \mathbf{y}_k$, where α_k counts the number of times $\mathbf{y}_k$ was misclassified from the beginning of the algorithm (note that $t_k\ \mathbf{y}_k$ is the *normalized* vector)

    and the discriminant is $\mathbf{a}^{\mathrm{T}}\mathbf{y} = \sum_k \alpha_k\, t_k \mathbf{y}_k^T \mathbf{y} = \sum_k \alpha'_k\, \mathbf{y}_k^T \mathbf{y}$, where $\alpha'_k = \alpha_k t_k$
  - This defines the *dual form* of the Perceptron
- **The algorithm may be *kernelized* to deal with nonlinear classif.**
  - See later (chapter on SVM's) and Computer Exercise

Remarks

- **The Perceptron is not an ordinary gradient descent algorithm**
  - The gradient is discontinuous. A special convergence proof is required.
  - If classes are linearly separable, the Perceptron algorithm converges in a finite number of iterations

    else, it may not converge
- **The Perceptron generated many variants**
  - Variable increment, by decreasing δ,
    - ✓ Helps convergence in the non-lineraly separable case
  - Fractional correction
    - ✓ If $\mathbf{y}$ is misclassified, fix the step δ according to the distance between $\mathbf{a}$ and the separating plane:

$$\mathbf{a} \leftarrow \mathbf{a} + \delta t\mathbf{y} = \mathbf{a} + \lambda \frac{(\mathbf{a}^T\mathbf{y})}{\|\mathbf{y}\|^2} t\mathbf{y} = \mathbf{a} + \lambda . r . \frac{\mathbf{y}}{\|\mathbf{y}\|}$$

    - ✓ Taking $\lambda > 1$ ensures that $\mathbf{a}$ moves beyond the separation plane, to its positive side.

The Least Squares (LS) criterion

- **The Perceptron only uses misclassified data.**
- **The LS criterion uses all samples.**
- **Difference between approaches**
  - The Perceptron tries making all scalar products $t_k(\mathbf{a}^{\mathrm{T}}\mathbf{y}_k)$ positive
  - LS try to have $t_k(\mathbf{a}^{\mathrm{T}}\mathbf{y}_k) = b_k$, $\forall k$, where the b_k's are arbitrary positive constants.
- **This leads to solving a set of linear equations**

$$\boxed{Ya = b}$$

  - Each line of Y contains one (augmented and normalized) sample, $t_k \mathbf{y}_k$.
  - Y is of size N x (d+1).

Direct Least Squares solution

- **The direct solution $a=Y^{-1}b$ is generally not available**
  - There are more samples than the dimension of feature space.
  - The system is over-determined
- **One shall rather minimize the error norm: $J_{LS}=\|\ Y\ a - b\ \|^2$**
- **As usual, we have to cancel the derivative of J_{LS}**

$$\frac{\partial J_{LS}}{\partial a} = 2Y^T(Ya-b)=0 \longrightarrow Y^TYa = Y^Tb$$

- **Y^TY is square-sized and most often non-singular, hence**

$$\boxed{a=(Y^TY)^{-1}Y^Tb = Y^\dagger b}$$

Regularized Least Squares solution

- **Note that if Y is non singular, the pseudo-inverse $Y^\dagger$ coincides with the usual inverse.**
- **In some cases, Y^TY may become almost singular.**
  - One must then regularize the solution
  - e.g. *ridge regression* [Guttierez]

$$\boxed{a=\left((1-\varepsilon)Y^TY+\varepsilon\frac{tr(Y^TY)}{d}.I\right)^{-1}Y^Tb}$$

- ε is a regularization parameter

Iterative Least Squares solution

- **One can also minimize J_{LS} iteratively**
  - This avoids singularity problems (stopping iterations = regularization)
  - This avoids inverting large matrices
- **The gradient descent iteration on J_{LS} is**

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

  - This sequence converges if δ_k decreases as 1/k. (regularization!)
- **To save memory, samples may be considered in a sequential fashion**

$$\boxed{a=a+\delta^i(b_i-a^Ty_i)y_i}$$

- **This is called LMS or Widrow-Hoff iteration**

Remarks

- **Relationship with Fisher's linear discriminant**
  - Using normalization: $y \leftarrow [-y]$, $\forall\ y \in \omega_2$
  - If n_1 samples belong to ω_1 and the other n_2, to ω_2
  - For a particular choice of b, the LS solution is Fisher's linear discriminant (see [Duda, pp.242-243], [Bishop p.190])

$$Y=\begin{bmatrix} 1_1 & X_1 \\ -1_2 & -X_2 \end{bmatrix}$$

$$a=\begin{bmatrix} w_0 \\ w \end{bmatrix} \qquad b=\begin{bmatrix} \frac{n}{n_1}1_1 \\ \frac{n}{n_2}1_2 \end{bmatrix}$$

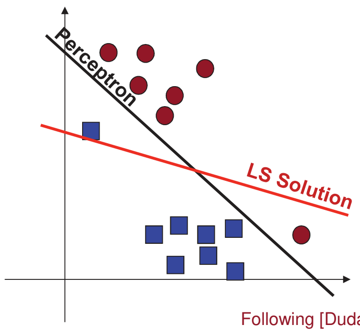
- **Bayesian interpretation**
  - Recall that the MAP Bayesian discriminant function is $g_0(x)=P(\omega_1|x)-P(\omega_2|x)$
  - The mean quadratic error due to the approximation of g_0 by a^Ty is

$$\varepsilon^2=\int\left(a^Ty-g_0(x)\right)^2p(x)dx$$

  - It may be shown that minimizing J_{LS} is equivalent to minimizing ε^2 when the number of samples becomes arbitrarily large [Duda, Webb].

Conclusion

- **Contrary to the Perceptron, LS always lead to a solution, even if the classes are not separable…**
- **However, there is no warranty that the solution corresponds to a separating hyper-plane, even in the separable case.**



Following [Duda]

- **It depends on the choice of b...**
- **If the classes are separable, there should exist a* and b* such that Ya* = b* > 0 (i.e. with a certain *margin*)**
  - Note: this is in the *normalized* framework
  - Implementation: Ho-Kashyap procedure

Ho-Kashyap Procedure (1965)

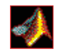
- **This time, we have 2 unknowns:**
  - The weight vector, which defines the separating hyper-plane, a*
  - The vector b*, which defines the separation margin and must be positive.
- **Algorithm: choose b^{*0} positive**
  - Repeat
    - ✓ Find the values of b* using gradient descent with positivity constraint

$$\boxed{b^{*k+1} = b^{*k} - \delta \frac{e-|e|}{2}} \qquad e = b^{*k} - Ya^{*k}$$

    - ✓ Estimate the weight vector (ML solution)

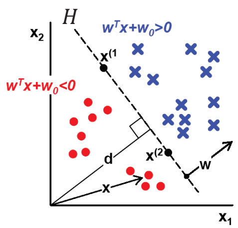
$$\boxed{a^{*k+1} = Y^{\dagger} b^{*k+1}}$$

  - until (b* does not evolve) or (maximal number of iterations reached)
  - If large residuals $|e|$ remain, we know that the samples are not separable !



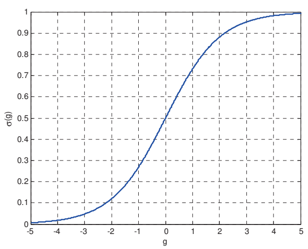
Logistic regression, NB: $t_k \in \{0,1\}$

- **Despite its name, classification algorithm**
  - But learning = regression
- **Intuition: Bayesian decision**
  - The farthest $\mathbf{x}$ from H, the closest:
    $P(\omega_1|\mathbf{x})$ to 1, so $\mathbf{x} \to \omega_1$
    or $P(\omega_1|\mathbf{x})$ to 0, so $\mathbf{x} \to \omega_2$
  - If $\mathbf{x}$ lies on H, $P(\omega_1|\mathbf{x}) = P(\omega_2|\mathbf{x}) = \frac{1}{2}$



➔ $P(\omega_1|\mathbf{x}) = \sigma$**, function of signed distance to** H**:** $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 = \mathbf{a}^T\mathbf{y}$

$$\boxed{\sigma(g) = \frac{1}{1 + exp(-g)}}$$



- **S-shaped function = logistic sigmoid**

Logistic regression: decision rule

$$\boxed{P(\omega_1|x) = \sigma(g) = \frac{1}{1+\exp(-g)}} \qquad \boxed{P(\omega_2|x) = 1-\sigma(g) = \frac{\exp(-g)}{1+\exp(-g)}}$$

$$\boxed{ln\left(\frac{P(\omega_1|\mathbf{x})}{P(\omega_2|\mathbf{x})}\right) = ln\left(\frac{\sigma}{1-\sigma}\right) = g(\mathbf{x})}$$

- **Corresponds to usual Bayesian decision (MAP) rule**

$$\boxed{\frac{P(\omega_1|x)}{P(\omega_2|x)} \underset{\omega_2}{\overset{\omega_1}{\gtrless}} 1} \Longleftrightarrow \boxed{g(x) \underset{\omega_2}{\overset{\omega_1}{\gtrless}} 0} \Longleftrightarrow \boxed{\sigma[g(x)] \underset{\omega_2}{\overset{\omega_1}{\gtrless}} \frac{1}{2}}$$

Logistic regression: learning

- **Estimate $\mathbf{a} = (\mathbf{w}, w_0)^T$ from a set of labeled data $\{\mathbf{x}_k, t_k\}_{k=1\dots N}$**
  - We suppose that $t_k \in \{0,1\}$. Recall that $\mathbf{y}_k = (1, \mathbf{x}_k)^T$.
- **Maximum Likelihood estimation**
  - Each t_k is a Bernoulli variable of parameter $P(\omega_1|\mathbf{x}_k) = \sigma(g(\mathbf{x}_k)) \stackrel{\text{def}}{=} \sigma_k$
  - The likelihood is

$$p(\mathbf{t}|\mathbf{a}) = \prod_{k=1}^{k=N} \sigma_k^{t_k}(1-\sigma_k)^{1-t_k}$$

  - The negative log-likelihood is the *cross-entropy* error function

$$E(\mathbf{a}) = -ln[p(\mathbf{t}|\mathbf{a})] = -\sum_{k=1}^{k=N} \{t_k ln(\sigma_k) + (1-t_k) ln(1-\sigma_k)\}$$

  - Whose gradient is

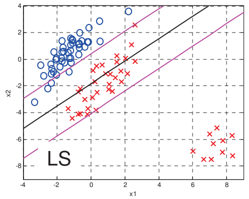
$$\nabla_{\mathbf{a}} E = \sum_{k=1}^{k=N} (\sigma_k - t_k)\mathbf{y}_k$$

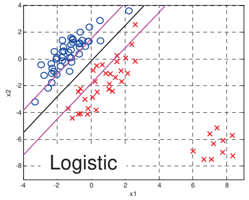
Logistic regression: learning

- **Logistic Regression is nonlinear**
  - No closed-form solution (contrary to LS)
  - …but simple expression for the gradient
- **Iterative descent algorithms**
  - Gradient descent: $\mathbf{a} \leftarrow \mathbf{a} - \delta \nabla_{\mathbf{a}} E$
    - ✓ e.g. sequential

$$\mathbf{a} \leftarrow \mathbf{a} + \delta(t_k - \sigma(\mathbf{a}^T\mathbf{y}_k))\mathbf{y}_k$$

    - ✓ Compare to Widrow-Hoff iteration !
  - Newton-Raphson ($\mathbf{a} \leftarrow \mathbf{a} - H^{-1}\nabla_{\mathbf{a}} E$), IRLS

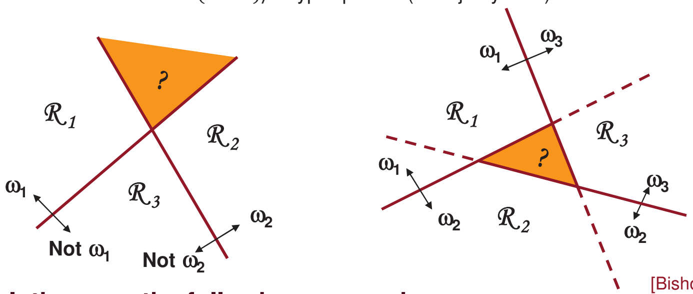




- **Remark (cf. [Bishop, PRML])**
  - More robust than LS (see figures & Computer Exercise)

Generalization to C classes

- **Attempting to construct a C class discriminant from a set of two-class discriminants may lead to ambiguities**
  - One-versus-the-rest : $C-1$ hyper-planes
  - One-versus-one : $C(C-1)/2$ hyper-planes (+ majority vote)



[Bishop]

- **Solution: use the following approach**
  - Evaluate C **discriminant functions,** $g_i(\mathbf{x}) = \mathbf{a}_i^T\mathbf{y}$ and choose the maximum
  - Decision regions are singly connected and convex

Generalization of the Perceptron to C classes

- **Generalization of the fixed increment algorithm**
  - Choose arbitrary initial values for $\mathbf{a}_i$
  - Consider each training sample in turn
  - If the sample $\mathbf{y}^k$ belongs to ω_i while the maximal discriminant function is g_j, then both vectors $\mathbf{a}_i$ and $\mathbf{a}_j$ are updated

$$\begin{cases} \mathbf{a}_i \leftarrow \mathbf{a}_i + \boldsymbol{\delta}.\mathbf{y}^k \\ \mathbf{a}_j \leftarrow \mathbf{a}_j - \boldsymbol{\delta}.\mathbf{y}^k \end{cases}$$

- **In other words:**
  - The weight of the true (desired) category is increased
  - The weight of the wrong category is decreased
  - Other weights do not change
- **Convergence results (convergence in a finite number of iterations) generalize: Kesler's construction**

Generalization of LS to C classes *[Duda pp. 268-269]*

- **A is the $(d+1)\times C$ matrix of weighting coefficients (one vector a_1 per class)** $\quad A = [\boldsymbol{a}_1 \quad \boldsymbol{a}_2 \quad \cdots \quad \boldsymbol{a}_C]$
- **Y is a $N\times(d+1)$ matrix, where Y_i gathers all *augmented* samples from class ω_i.** $\quad Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_C \end{bmatrix}$
- **T is a $N\times C$ matrix, whose columns T_i are zero, except the i-th one, which is 1 (1-of-C or one-hot encoding)** $\quad T = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_C \end{bmatrix}$
- **LS minimize $\mathrm{tr}\{(YA-T)^T(YA-T)\}$, leading to $\hat{A} = Y^{\dagger}T$**

Generalization of logistic regression to C classes

- **Use Bayes theorem:** $P(\omega_k|\mathbf{x}) = \frac{p(\mathbf{x}|\omega_k)P(\omega_k)}{\sum_j p(\mathbf{x}|\omega_j)P(\omega_j)}$

  and consider $g_k(\mathbf{x}) = \ln p(\mathbf{x}|\omega_k)P(\omega_k)$
- **The posterior probabilities are given by the normalized exponential of $g_k(\mathbf{x}) = \mathbf{a}_k^T\mathbf{y}$, a.k.a. the *softmax* function**

$$\boxed{P(\omega_k|\mathbf{x}) = \frac{\exp\left(g_k(\mathbf{x})\right)}{\sum_j \exp\left(g_j(\mathbf{x})\right)}}$$

  - When $g_k(\mathbf{x}) \gg g_j(\mathbf{x}), \forall j \neq k$, then $P(\omega_k|\mathbf{x}) \cong 1$ and $P(\omega_j|\mathbf{x}) \cong 0\ \forall j \neq k$
  - When $C=2$, $P(\omega_1|\mathbf{x}) = 1/(1+\exp -\mathbf{a}^T\mathbf{y})$ with $\mathbf{a} = \mathbf{a}_1 - \mathbf{a}_0$ (check it!)
- **Learning $E(\mathbf{a}) = -ln[p(\mathbf{T}|\mathbf{a})] = -\sum_{n=1}^{n=N}\sum_{k=1}^{k=C} t_{nk} ln\left(P(\omega_k|\mathbf{x}_n)\right)$**
  - ML estimation: minimize cross-entropy by iterative (descent) algorithms

Summary

- **Finding a separating hyper-plane (learning)**
  - Is ill-posed (many possible solutions ∊ solution region), is solved by optimizing a criterion
- **The Perceptron algorithm**
  - Penalizes misclassified samples
  - Always finds a separating hyper-plane when the classes are separable
  - May not converge if the classes are not separable
- **Least-squares**
  - Minimize the quadratic error between $g(\mathbf{x})$ and a *margin*, b
  - Always converge
  - Find a solution which may not be a separating boundary, even if the samples are separable! The Ho-Kashyap procedure is a solution to this problem
- **Logistic regression**
  - Relates linear discriminants to Bayesian decision
  - Maximum likelihood estimation, descent algorithms
  - Is more robust to outliers
- **These algorithms generalize to C classes**
- **Extension to nonlinear discrimination: consider $\phi(\mathbf{x})$ instead of $\mathbf{x}$**

Kesler's construction *([Duda, p.266])*

- **Correct classification ($\mathbf{y} \in \omega_i$) iff**

$$\left(\mathbf{a}_i^T\mathbf{y} - \mathbf{a}_j^T\mathbf{y}\right) > 0 \Leftrightarrow \boldsymbol{\alpha}^T\boldsymbol{\eta}_{ij} > 0, \qquad \forall j \neq i, j \in [1 \dots C]$$

provided a $C(D+1)$ weight vector α and, for each sample y, $(C-1)$ extended observation vectors $\boldsymbol{\eta}_{ij}$

$$\boldsymbol{\alpha} = \begin{bmatrix} \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_C \end{bmatrix} \qquad \boldsymbol{\eta}_{ij} = \begin{array}{c} \begin{bmatrix} 0 \\ \vdots \\ \mathbf{y} \\ \vdots \\ -\mathbf{y} \\ \vdots \\ 0 \end{bmatrix} \end{array} \begin{array}{l} \\ \\ \leftarrow i \\ \\ \leftarrow j \\ \\ \\ \end{array}$$

- **Perceptron (fixed increment) rule**

$$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + \boldsymbol{\eta}_{ij} \Leftrightarrow \begin{cases} \mathbf{a}_i \leftarrow \mathbf{a}_i + \boldsymbol{\delta}.\mathbf{y} \\ \mathbf{a}_j \leftarrow \mathbf{a}_j - \boldsymbol{\delta}.\mathbf{y} \end{cases}$$

Neural approaches

Multilayer Perceptron

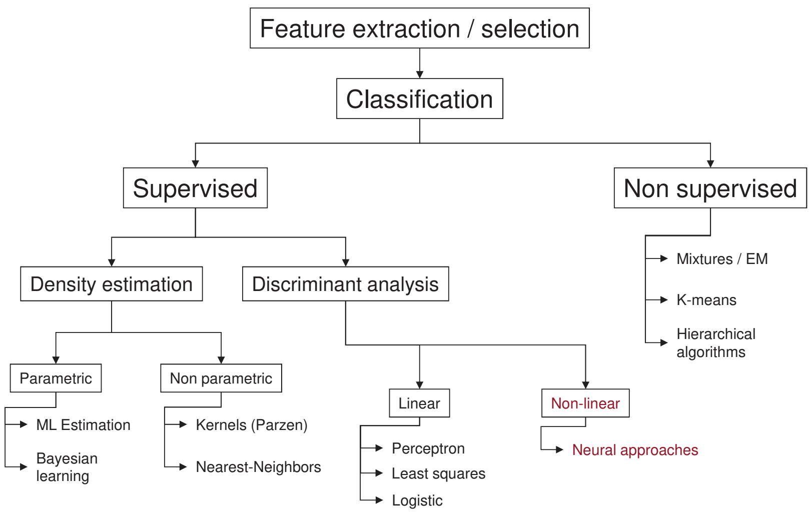
Back-Propagation Algorithm

Radial Basis Functions

Summary of previous episodes

- **In the first chapters, we studied**
  - The Bayesian classification framework,
  - Dimensionality reduction and learning methods,
- **…in the previous chapter, we saw how it is possible to design classifiers with linear decision boundaries, or linear classifiers.**
- **In the next two chapters, we will focus on ways to generalize these techniques to nonlinear discrimination.**
- **Neural approaches represent the first one…**

A hierarchy of methods



The Perceptron: a linear neuron model

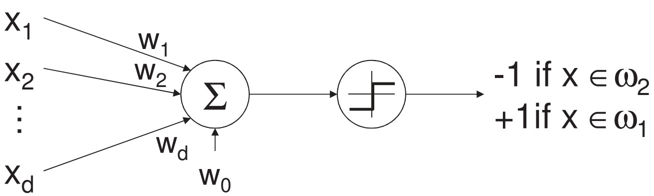
- **In the previous chapter,**
  - We have seen how to learn a linear machine by minimizing a criterion: Least Squares, Logistic or the *Perceptron*
  - Perceptron is the name of one of the earliest model of artificial neural networks that were proposed in the literature (Rosenblatt, 1957).
- **Once the parameters of the Perceptron, $a=(w,w_0)^T$ have been learned, the classification is performed using:**

$$g(x) = w^T x + w_0 \underset{\omega_2}{\overset{\omega_1}{\gtrless}} 0$$

  - It is possible to devise an automata that computes g(x) and thresholds the result, providing a binary answer: this is the artificial neuron or Perceptron.

Graphical representation

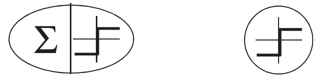
- **This machine may be represented in the following way**



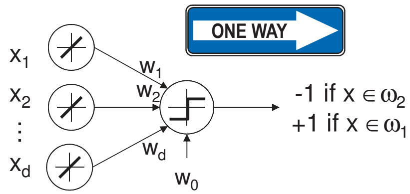
- **The neuron computes the weighted sum of its entries (synapses), then applies an activation function to it**
  - The weights are called synaptic weights or, simply, synapses
  - w_0 is the threshold (or bias)
  - The step function is only one of many possible choices for the activation function. Moreover, the outputs (class labels) are sometimes 0 and 1.

Other representations

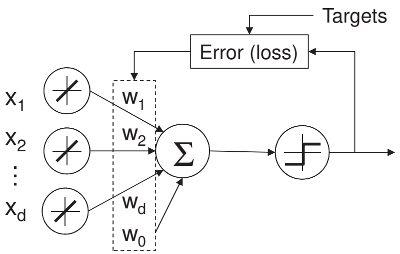
- **Other representations may be found in the literature**



- **The linear classifier may also be represented as a feed-forward network with 2 layers:**
  - The input layer: linear-response neurons (they only copy their entries)
  - The output layer has only one neuron in the 2-class case.

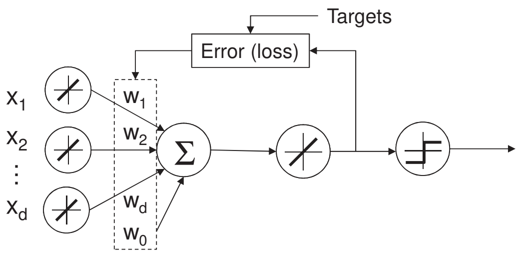


Linear discriminants as feed-forward Neural Net

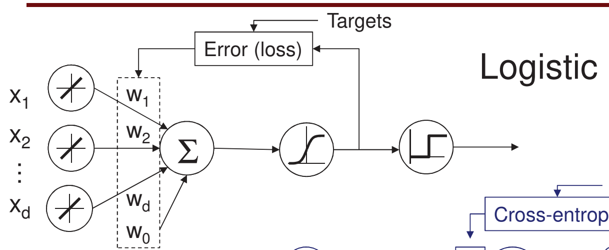


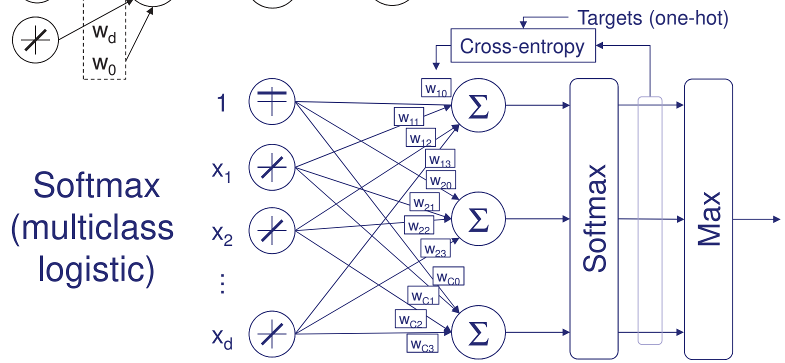
Perceptron

LMS



Linear discriminants as feed-forward Neural Net

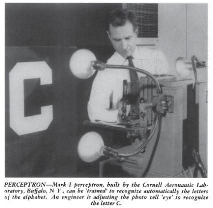




Historical notes (1)

following [Guttierez]

- **Earliest works trace back to the 1940's**
  - McCulloch and Pitts (a neuro-anatomist and a mathematician) explore artificial neural networks with binary activation functions (1943)
  - Hebb introduces learning: the efficiency of a synapse between two neurons is increased by repeated activations of one neuron by the other across the synapse (1949)
- **Rosenblatt (1957)**
  - Introduces the 2-layer Perceptron, its learning algorithm and convergence proof



- **Widrow and Hoff (1959)**
  - Propose the ADALINE (ADAptive Linear Neuron), similar to the Perceptron but with a linear activation function, as well as the LS learning algorithm.
- **The limitations of the Perceptron (linearity) were pointed out by (Minsky and Papert 1969): the famous "XOR case"…**

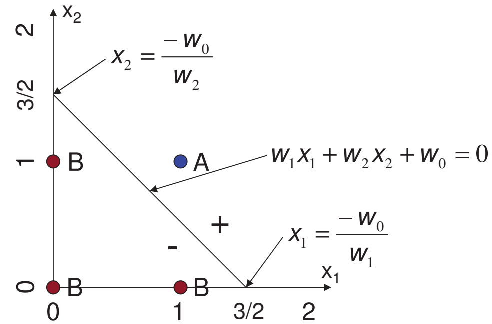
A linear classifier for the AND function

- **Truth table of the AND function**

x1	x2	x1 and x2	Class
0	0	0	B
0	1	0	B
1	0	0	B
1	1	1	A

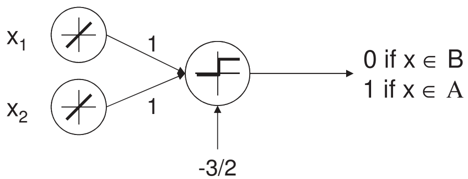
- **Graphical representation**
- **Coefficients**

$w_1 = 1$
$w_2 = 1$
$w_0 = -3/2$



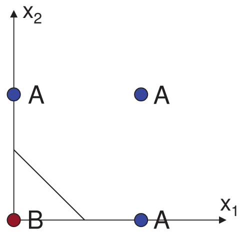
Neural linear classifier

- **Neural implementation**



- **The case of the OR function is straightforward,**

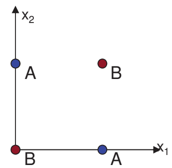
$w_1 = 1, w_2 = 1, w_0 = -1/2$



Example of the XOR function

- **Truth table of the XOR function:**

x1	x2	x1 XOR x2	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B



- **Linear discrimination using a single line is not achievable**
  - But we can make it using 2 lines…
  - This amounts to decomposing the problem into 2 successive phases: compute y1 and y2, then perform classification using these new features
  - Note: XOR = (OR) AND NOT (AND)

x1	x2	y1 = x1 OR x2	y2 = x1 AND x2	y = y1 AND NOT y2	Class
0	0	0	0	0	B
0	1	1	0	1	A
1	0	1	0	1	A
1	1	1	1	0	B

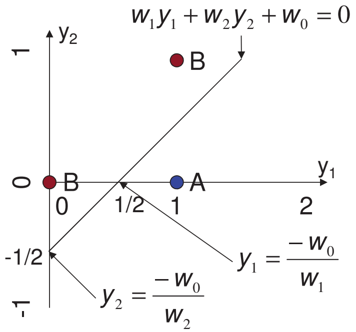
The XOR case (continued)

- **The first phase involves two discriminant functions: $g_1(x)$ and $g_2(x)$, with respective coefficients**

$$w_1 = 1, w_2 = 1, w_0 = -3/2$$

$$w_1 = 1, w_2 = 1, w_0 = -1/2$$

- **It yields 3 distinct samples: (0,0), (1,0), (1,1)**



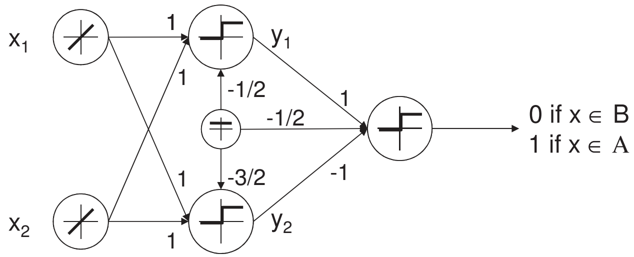
- **These samples are separable using g(x), with coefficients:**

$$w_1 = 1, w_2 = -1, w_0 = -1/2$$

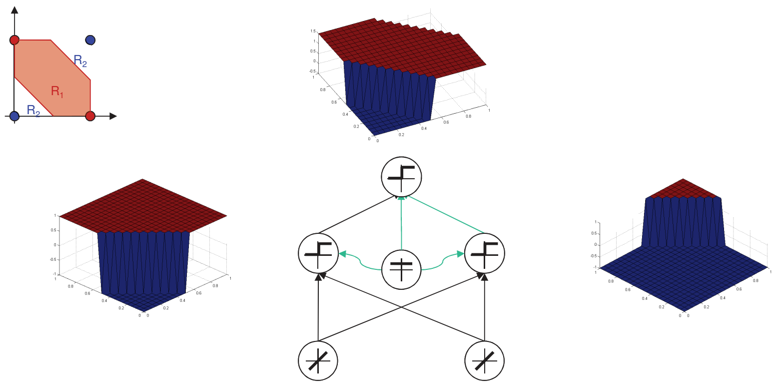
The XOR case: neural implementation

- **The first phase transformed the non-separable problem into a linearly separable one.**

- **A possible realization in the form of a neural network is:**



Graphical representation *(following [Duda])*



- **The XOR is implemented as a fully connected network with 2-2-1 topology.**
- **"Excited" synapses (positive weights) are shown in black, "inhibited" ones (negative weights), in cyan.**
- **The 3D plots show the discriminant functions implemented by the neurons**

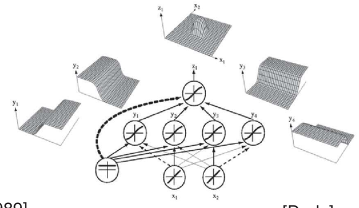
The multilayer Perceptron

- **We have just devised a simple multilayer Perceptron:**
  - 1 *input* layer
    - ✓ No calculations
    - ✓ As many nodes (neurons) as feature space dimensions
  - 1 *hidden* layer
    - ✓ here, neurons of phase 1
  - 1 *output* layer
    - ✓ here, the unique neuron which implements phase 2

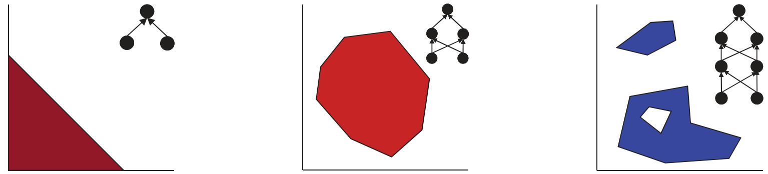
- **This architecture may be generalized**
  - Two or more hidden layers
  - More neurons in the hidden layer (best number of neurons?)
  - More neurons in the output layer (C>2): C neurons
  - Other kinds of activation functions

Representation capabilities

- **Multilayer Perceptrons**
  - with 1 hidden layer implement (as for the XOR) polyhedric decision regions
  - Using a sufficiently large number of neurons, they can approximate arbitrarily complex functions [Cybenko 1989] (beware of over-fitting, anyway!)
  - Networks with 2 hidden layers can discriminate classes stemming from the union of polyhedric regions



[Duda]

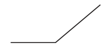


Following [Bishop95]

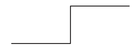
Activation functions

- **Linear: f(x) = x**
  - The neuron transmits the value of the weighted sum of its inputs

- **Rectifier: f(x)=max(x,0)**
  - Used for *Deep Learning* in *Rectified Linear Units* (ReLU)



- **Step function: f(x) = 1 if x>0, -1 (or 0) otherwise**
  - The neuron transmits the sign of the weighted sum of its inputs



- **Sigmoid (i.e. s-shaped functions)**
  - Logistic function: f(x) = 1/(1+exp(-αx))
  - Hyperbolic tangent : f(x) = c (1-exp(-αx))/(1+exp(-αx)) = c tanh(αx/2)
  - Take care to the effect of parameters on the shape of the function (saturation)

- **Softmax: for output units (C>2)**
  - $f(x_c) = \exp(x_c) / \sum_{c'=1}^{c'=C} \exp(x_{c'})$

Historical notes (2)

following [Guttierez]

- **The question of learning multilayer Perceptron…**
  - Remained unsolved for many years, despite the efforts of connectionists
  - Neural networks remained limited to linear discrimination!
  - This is probably the reason why neural networks received less attention during the 1970…
- **1986 is a key date:**
  - Announcement of the discovery of an algorithm that allowed a network to learn to discriminate nonlinearly separable classes, namely the error Back-Propagation algorithm…
  - This algorithm was indeed first proposed in 1974 (Werbos PhD), but became popular since 1986! (Rumelhart, Hinton, Williams, Le Cun)
- **We now study the derivation of this well-known algorithm**

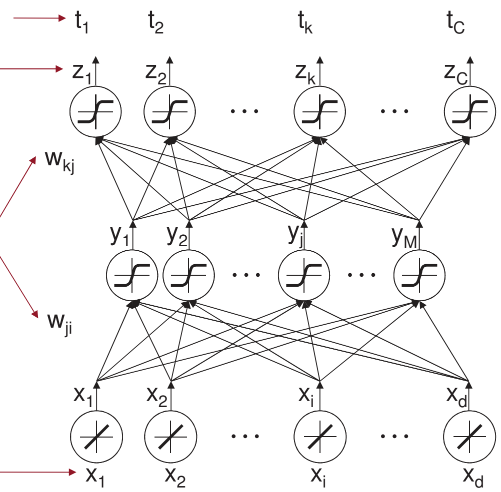
Multilayer Perceptron: notations

Target values t

Output z

Weights W

Inputs x (sample)



Back-propagation

- **The learning problem**
  - Find the weights W that best model the input/output correspondence from a set of training samples, $\{\mathbf{x}_n, \mathbf{t}_n\}_{n=1\ldots N}$
  - As usual, we set this as an optimization problem, e.g. minimize the quadratic error between expected outputs, t, and measured outputs, z

    (other *loss functions* might be used)
  - For the sake of simplicity, we first consider the case of a single sample (otherwise, the criterion must be averaged over the samples)

$$J(W) = \frac{1}{2}\sum_{k=1}^{C}(t_k - z_k)^2 = \frac{1}{2}\|t - z\|^2$$

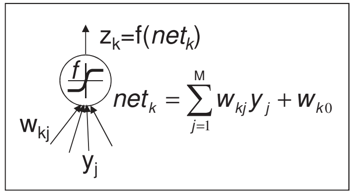
- **Back-propagation is essentially a gradient descent algorithm. For a weight w, the update rule is (where η = *learning rate*)**

$$w = w - \eta \frac{\partial J(W)}{\partial w}$$

Computing the updates (output layer)

- **Easy using the *chain rule***

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}}$$



- **One obtains**

$$\boxed{\frac{\partial J}{\partial w_{kj}} = -(t_k - z_k)f'(net_k)y_j \equiv -\delta_k^O y_j}$$

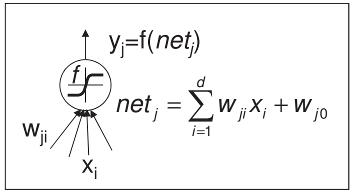
- **If f(x) = x, one obtains the LS (Widrow-Hoff) solution**
- **The update (sensitivity, δ_k^O) is proportional to the error, t_k-z_k**

Computing the updates (hidden layer)

- **Using the *chain rule***

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

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}f'(net_j)x_i$$

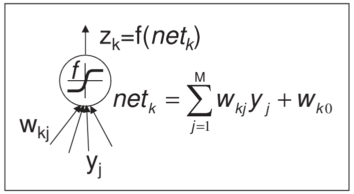


- **Computing $\partial J/\partial y_j$ seems more difficult than for the output layer because we have no idea of target values**
- **This is the *credit assignment problem* that puzzled connectionists for many years**

Computing the updates (hidden layer)

- **The trick is to note that all z_k's depend on y_j**

$$J = J(z_1(y_j),\ldots,z_k(y_j),\ldots,z_C(y_j))$$



- **Hence,**

$$\frac{\partial J}{\partial y_j} = \sum_{k=1}^{C}\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial y_j}$$

$$\frac{\partial J}{\partial y_j} = -\sum_{k=1}^{C}\underbrace{(t_k - z_k)\,f'(net_k)}_{\delta_k^O}w_{kj}$$

- **Finally,**

$$\boxed{\frac{\partial J}{\partial w_{ji}} = -\left[\sum_{k=1}^{C} w_{kj}\delta_k^O\right]f'(net_j)x_i \equiv -\delta_j^H x_i}$$

- **The error is *back-propagated* from output neurons, via δ_k**

Back-propagation

- **Then, finally, we obtain similar expressions:**

$$\boxed{\frac{\partial J}{\partial w_{kj}} = -(t_k - z_k) f'(net_k) y_j \equiv -\delta_k^O y_j}$$

For weights in the <u>output layer</u>

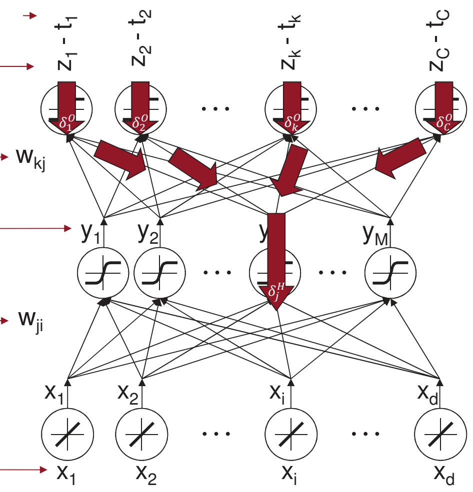
$$\boxed{\frac{\partial J}{\partial w_{ji}} = -\left[\sum_{k=1}^{C} w_{kj}\delta_k^O\right] f'(net_j) x_i \equiv -\delta_j^H x_i}$$

For weights in the <u>hidden layer</u>

- **Remark: the same procedure may be recursively applied to learn the weights of other (deeper) hidden layers, if needed**

Back-propagation and multilayer learning

4. Compute errors
3. Compute z_k
5. Update weights
2. Compute y_j
6. Update weights
1. Sample



Remarks

- **The name of the algorithm stems from the fact that, as just explained:**
  - During the learning stage, the error must be propagated from the output layer to the hidden layer, hence backward!
  - However, back-propagation is essentially a gradient descent algorithm dedicated to a stratified structure. Using differentiation composition rules allows differentiating the LMS criterion with respect to all the weights of the model…
- **The behavior of the algorithm depends on its starting point.**

  It is important to avoid setting all initial weights to zero!
  - If $w_{kj} = 0$, the back-propagated error is zero and weights in hidden layers never change!
- **This derivation may be generalized to more complex cases**
  - e.g. Convolutional Neural Networks (for image or speech recognition)

Implementation details

- **Sequential algorithm or batch algorithm ?**
  - Batch gradient descent uses the whole data set at once
  - Seems more reasonable but has poor performance!
  - One may use conjugate gradient or quasi-Newton methods
  - Other possibility: multiple random initializations
- **Sequential algorithms: stochastic gradient descent** (Le Cun *et al*, 1989)
  - Uses repeated updates by cycling through the data either in sequence or at random
  - May also use small sets of data points or “mini-batches” (intermediate scenario)
  - Better convergence properties
  - Used in *Deep Learning*
- **Avoiding over-fitting ?**
  - Learn with noisy data
  - Stop learning sufficiently early (according to the value of J(w), for example)

A Bayesian flavor

- **Each neuron k in the output layer computes a discriminant for class ω_k : $g_k(x;w)$**
  - It can be shown [Duda pp. 303-304] that minimizing J(W) becomes equivalent to minimizing:

$$\boxed{\varepsilon^2 = \sum_{k=1}^{C} \int \left(P(\omega_k|x) - g_k(x;w)\right)^2 p(x)\,dx}$$

  as the number of samples goes to the infinity

  ⇨ The multilayer Perceptron becomes then optimal (equivalent to MAP classifier)

- **This explains the success of neural networks in pattern recognition problems**

Radial basis functions (RBF's)

Radial Basis Functions (RBF)

- **RBF's are an alternative way of introducing non-linearity in linear discriminant models**

$$g(x) = w^T x + w_0 \quad \Rightarrow \quad g(x) = \sum_{k=1}^{M} w_k \varphi(x, c_k) + w_0$$

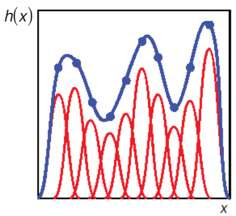
- **RBF's resemble exact interpolation techniques (Powell 1987)**
  - A basis function is placed on each center, x_k

$$h(x) = \sum_{k=1}^{M} w_k \varphi(x, x_k) = \Phi w$$

  - Then, the weights are optimized to minimize the Mean Square error at these points (LS solution)

$$h(x_k) = t_k \longrightarrow \mathbf{w} = \mathbf{\Phi}^{\dagger}\,\mathbf{t}$$

- **RBF's are, also, related to kernel-based PDF estimation methods (but M<<N!)**



[Gutierrez]

Basis functions

- **As their name suggests, the basis functions are radial, hence functions of || x - c_k ||.**

- **Examples:**

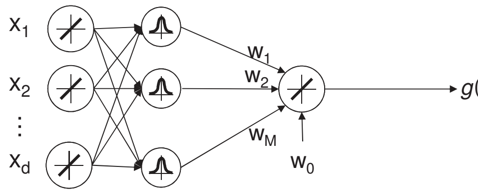
$$\varphi(x, c_k) = \exp\left(-\frac{1}{2\sigma_k^2}\|x - c_k\|^2\right)$$

$$\varphi(x, c_k) = \frac{\sigma_k^2}{\sigma_k^2 + \|x - c_k\|^2}$$

- **The spread of the function is a parameter (related to σ)**
- **Gaussian kernels are the more widely used functions**

Neural interpretation (2-classes case)

- **Nonlinear hidden layer and linear output layer**



$$g(x)=\sum_{k=1}^{M} w_k \varphi(x,c_k)+w_0$$

- **Remark: here, 2 classes ⇨ 1 output neuron**
  - Contrary to the Perceptron, RBF neurons in the hidden layer do not sum their entries, but compute φ
  - In the Perceptron, the hidden layer realizes a projection, whose value is identical on a hyper-plane, hence global. In RBF's, iso-surfaces are hyper-ellipsoids, hence with local range
  - RBF's generally need more centers to attain the same level of performance as Perceptrons. However, they "learn" faster.

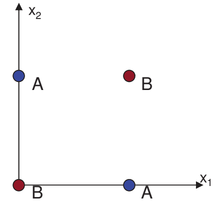
Example (1/2)

- **Back to the XOR problem**
- **Choose 2 centers:**

$$c_1=\begin{bmatrix}1\\1\end{bmatrix} \qquad c_2=\begin{bmatrix}0\\0\end{bmatrix}$$

- **and consider**

$$\varphi(x,c_k)=\exp\left(-\|x-c_k\|^2\right)$$



- **The hidden layer implements the following transformation:**

$$x=\begin{bmatrix}x_1\\x_2\end{bmatrix}\rightarrow y=\begin{bmatrix}y_1\\y_2\end{bmatrix}=\begin{bmatrix}\exp\left(-\|x-c_1\|^2\right)\\\exp\left(-\|x-c_2\|^2\right)\end{bmatrix}$$

Example (2/2)

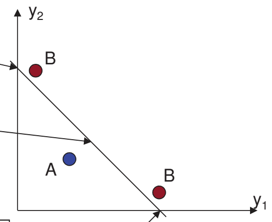
- **Hence:**

$$x=\begin{bmatrix}0\\0\end{bmatrix}\rightarrow y=\begin{bmatrix}\exp(-2)\\1\end{bmatrix} \qquad x=\begin{bmatrix}1\\1\end{bmatrix}\rightarrow y=\begin{bmatrix}1\\\exp(-2)\end{bmatrix}$$

$$x=\begin{bmatrix}0\\1\end{bmatrix}\rightarrow y=\begin{bmatrix}\exp(-1)\\\exp(-1)\end{bmatrix} \qquad x=\begin{bmatrix}1\\0\end{bmatrix}\rightarrow y=\begin{bmatrix}\exp(-1)\\\exp(-1)\end{bmatrix}$$

$$y_2=\frac{-w_0}{w_2}=1$$

$$g(y)=y_1+y_2-1=0$$

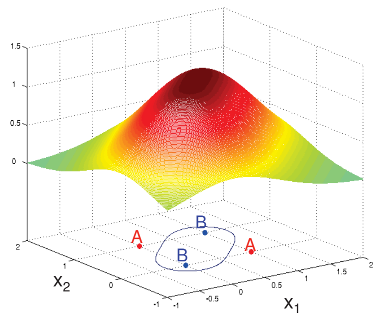


$$\boxed{g(x)=\exp\left(-\|x-c_1\|^2\right)+\exp\left(-\|x-c_2\|^2\right)-1}$$

$$y_1=\frac{-w_0}{w_1}=1$$

Interpretation

- We computed a function which sums 2 Gaussians, with variance ½, centered on the samples from class B.
- Then, we put a threshold at height 1.



- **The C-class case: C neurons in the output layer**
  - Compute C discriminant functions, g_k, and affect sample to the class that realizes the maximum

$$g_k(x)=\sum_{j=1}^{M} w_{kj}\varphi(x,c_j)+w_{0k}$$

Learning RBF’s

- **As in the XOR case, where it was arbitrary, the choice of centers is an important question.**
  - The problem is to approximate as well as possible the distribution of sample. One may use:
    - ✓ Random sampling, but this requires a large number of centers
    - ✓ A clustering algorithm, such as k-means or EM
      spreading parameters are either estimated separately (k-means) or given by the method (EM)

- **Learning the weights is performed using a linear technique, e.g. LS.**

- **Some methods perform these steps simultaneously, to minimize a unique criterion**
  - Orthogonal Least Squares (OLS) iterate
    - ✓ Center selection
    - ✓ Computation of weights using the pseudo-inverse

Take-away remarks

- **Neural Networks**
  - Efficient and flexible, allow generating arbitrarily complex decision boundaries (beware of the generalization issue !)
  - Require some “know-how”.
  - We studied the simplest architecture. Many others exist, for solving different problems:
    - ✓ Dimensionality reduction: non-linear PCA
    - ✓ Non-supervised learning and classification: ART,
    - ✓ “Mix” of both: Kohonen networks, Self-Organizing Maps…
  - *Deep learning* methods (i.e. large number of hidden layers) are beating records in many applications

- **Radial Basis Function (RBF) are one alternative way of accounting for non-linearity.**