# CSCE 970 Lecture 4: Nonlinear Classifiers

Stephen D. Scott

January 29, 2001

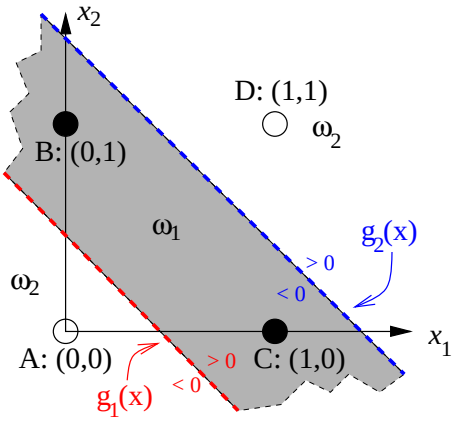


## Introduction

- For non-linearly separable classes, performance of even the best linear classifier might not be good
- Thus we will remap feature vectors to new space where they are (almost) linearly separable
- Outline:
  - Multiple layers of neurons
    - Backpropagation
    - Sizing the network
  - Polynomial remapping
  - Gaussian remapping (radial basis functions)
  - Efficiency issues (support vector machines)
  - Other nonlinear classifiers (decision trees)



## Getting Started: The XOR Problem




- Can't represent with a single linear separator, but can with intersection of two:

$$g_1(\mathbf{x}) = 1 \cdot x_1 + 1 \cdot x_2 - 1/2$$
$$g_2(\mathbf{x}) = 1 \cdot x_1 + 1 \cdot x_2 - 3/2$$

- $\omega_1 = \left\{\mathbf{x} \in \Re^\ell : g_1(\mathbf{x}) > 0 \text{ AND } g_2(\mathbf{x}) < 0\right\}$
- $\omega_2 = \left\{\mathbf{x} \in \Re^\ell : g_1(\mathbf{x}), g_2(\mathbf{x}) < 0 \text{ OR } g_1(\mathbf{x}), g_2(\mathbf{x}) > 0\right\}$



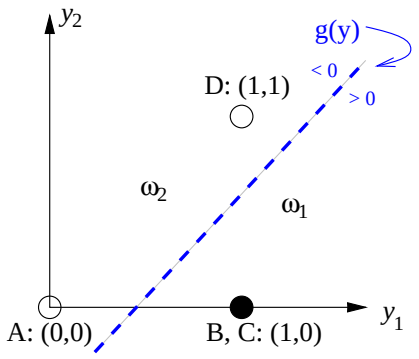
## Getting Started: The XOR Problem (cont'd)

- Let $y_i = \begin{cases} 0 & \text{if } g_i(\mathbf{x}) < 0 \\ 1 & \text{otherwise} \end{cases}$

| Class | $(x_1, x_2)$ | $g_1(\mathbf{x})$ | $y_1$ | $g_2(\mathbf{x})$ | $y_2$ |
|---|---|---|---|---|---|
| $\omega_1$ | B: $(0,1)$ | $1/2$ | 1 | $-1/2$ | 0 |
| $\omega_1$ | C: $(1,0)$ | $1/2$ | 1 | $-1/2$ | 0 |
| $\omega_2$ | A: $(0,0)$ | $-1/2$ | 0 | $-3/2$ | 0 |
| $\omega_2$ | D: $(1,1)$ | $3/2$ | 1 | $1/2$ | 1 |

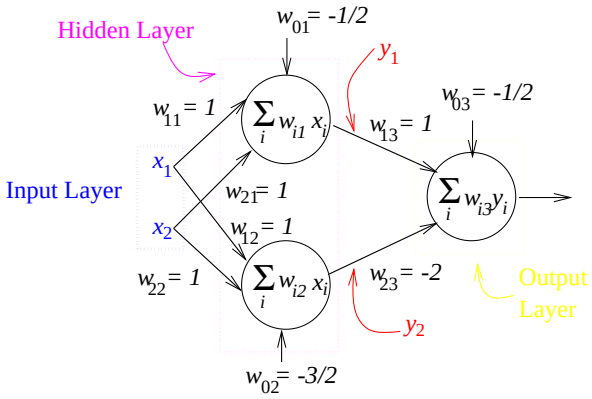
- Now feed $y_1$, $y_2$ into:

$$g(\mathbf{y}) = 1 \cdot y_1 - 2 \cdot y_2 - 1/2$$



## Getting Started: The XOR Problem (cont'd)

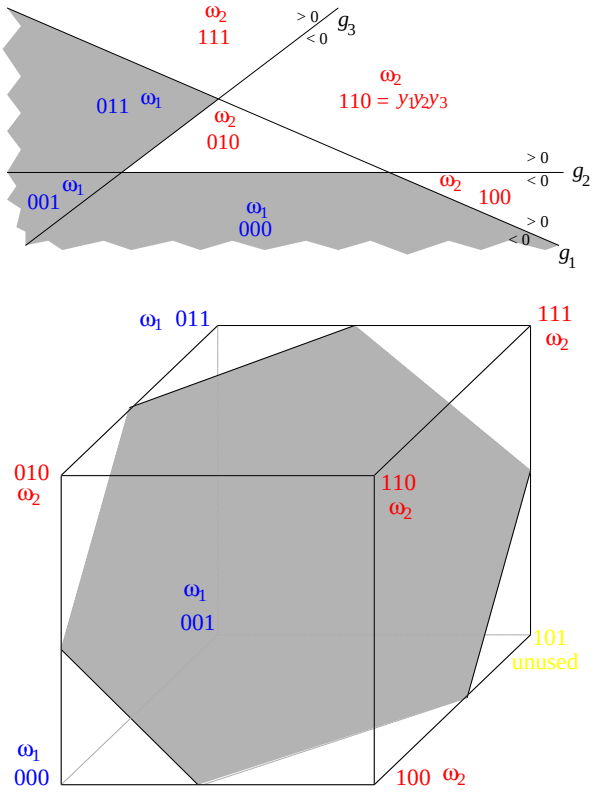
- In other words, we remapped all vectors $\mathbf{x}$ to $\mathbf{y}$ such that the classes are linearly separable in the new vector space




- This is a two-layer perceptron or two-layer feedforward neural network
- Each neuron outputs 1 if its weighted sum exceeds its threshold, 0 otherwise



## What Else Can We Do with Two Layers?






## What Else Can We Do with Two Layers? (cont'd)

- Define the $p$-dimensional unit hypercube as

$$H_p = \left\{ [y_1, \ldots, y_p]^T \in \Re^p, y_i \in [0,1] \, \forall i \right\}$$

- A hidden layer with $p$ neurons maps an $\ell$-dim vector $\mathbf{x}$ to a $p$-dim vector $\mathbf{y}$ whose elements are corners of $H_p$, i.e. $y_i \in \{0,1\} \, \forall i$
- Each of the $p$ neurons corresponds to an $\ell$-dim hyperplane
- The intersection* of the (pos. or neg.) half-spaces from these $p$ hyperplanes maps to a vertex of $H_p$
- If the classes of $H_p$'s vertices are linearly separable, then a perfect two-layer network exists
- I.e. a 2-layer network can separate classes consisting of unions of adjacent polyhedra

*Also known as polyhedra.



## Three-Layer Networks

- With two-layer networks, there exist unions of polyhedra not linearly separable on $H_p$
- I.e. there exist assignments of classes to points on $H_p$ that are not linearly separable
- Solution: Add a second hidden layer of $q$ neurons to partition $H_p$ into regions based on class
- Output layer combines appropriate regions
- E.g. including 110 from Slide 6 in $\omega_1$ is possible using procedure similar to XOR solution
- In general, can always use simple procedure of isolating each $\omega_1$ node in $H_p$ with its own second-layer hyperplane and taking disjunction
- Thus, can use 3-layer network to perfectly classify any union polyhedral regions

## The Backpropagation Algorithm

- A popular way to train a neural network
- Assume the architecture is fixed and complete
  - $k_r$ = number of nodes in layer $r$ (could have $k_L > 1$)
  - $w_{ji}^r$ = weight from neuron $i$ in layer $r-1$ to neuron $j$ in layer $r$
  - $v_j^r = \sum_{k=1}^{k_{r-1}} w_{jk}^r \, y_k^{r-1} + w_{j0}^r$
  - $y_j^r = f\left(v_j^r\right)$ = output of neuron $j$ in layer $r$
- During training we'll attempt to minimize a cost function, so use differentiable activation func. $f$, e.g.:

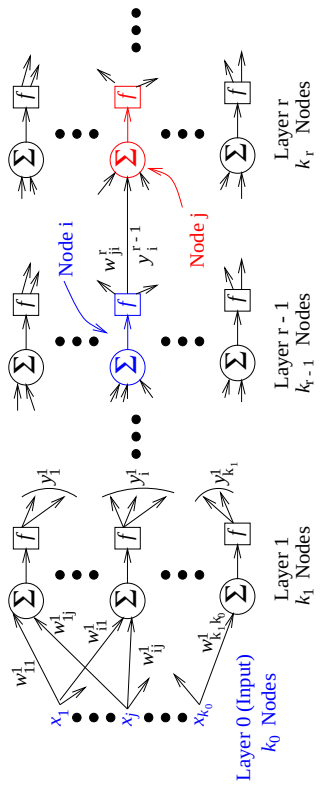
$$f(v) = \frac{1}{1+e^{-av}} \in [0,1]$$

OR

$$f(v) = c\tanh(av) \in [-c,c]$$



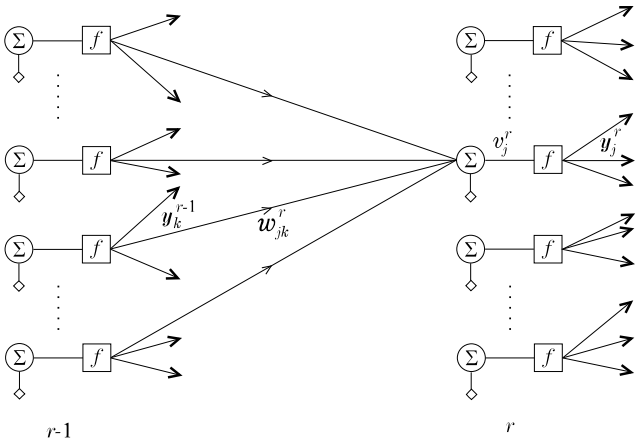
## The Backpropagation Algorithm






## The Backpropagation Algorithm

Another Picture






## The Backpropagation Algorithm

Intuition

- Recall derivation of Perceptron update rule:
  - Cost function:

$$U(\mathbf{w}) = \sum_{i=1}^{\ell} (w_i(t+1) - w_i(t))^2 + \eta \left( y(t) - \sum_{i=1}^{\ell} w_i(t+1) x_i(t) \right)^2$$

  - Take gradient w.r.t. $\mathbf{w}(t+1)$, set to $\mathbf{0}$, approximate, and solve:

$$w_i(t+1) = w_i(t) + \eta \left( y(t) - \sum_{i=1}^{\ell} w_i(t) x_i(t) \right) x_i(t)$$

## The Backpropagation Algorithm
Intuition: Output Layer

- Now use similar idea with $j$th node of output layer (layer $L$):
  - Cost function:

$$U\left(\mathbf{w}_j^L\right) = \sum_{k=1}^{k_{L-1}} \left(w_{jk}^L(t+1) - w_{jk}^L(t)\right)^2 + \eta \left[\overbrace{y_j(t)}^{correct} - \overbrace{f\left(\sum_{k=1}^{k_{L-1}} w_{jk}^L(t+1) y_k^{L-1}(t)\right)}^{pred = y_j^L(t) \text{ with } \mathbf{w}(t+1)}\right]^2$$

  - Take gradient w.r.t. $\mathbf{w}_j^L(t+1)$ and set to $\mathbf{0}$:

$$0 = 2\left(w_{jk}^L(t+1) - w_{jk}^L(t)\right) - 2\eta \left[y_j(t) - f\left(\sum_{k=1}^{k_{L-1}} w_{jk}^L(t+1) y_k^{L-1}(t)\right)\right] \cdot f'\left(\sum_{k=1}^{k_{L-1}} w_{jk}^L(t+1) y_k^{L-1}(t)\right) y_k^{L-1}(t)$$



## The Backpropagation Algorithm
Intuition: Output Layer
(cont'd)

- Again, approximate and solve for $w_{jk}^L(t+1)$:

$$w_{jk}^L(t+1) = w_{jk}^L(t) + \eta\, y_k^{L-1}(t) \cdot \left[y_j(t) - f\left(\sum_{k=1}^{k_{L-1}} w_{jk}^L(t) y_k^{L-1}(t)\right)\right] \cdot f'\left(\sum_{k=1}^{k_{L-1}} w_{jk}^L(t) y_k^{L-1}(t)\right)$$

- So:

$$w_{jk}^L(t+1) = w_{jk}^L(t) + \eta\, y_k^{L-1}(t) \underbrace{\left(y_j(t) - f\left(v_j^L(t)\right)\right) f'\left(v_j^L(t)\right)}_{\delta_j^L(t) = \text{“error term”}}$$

- For $f(v) = 1/(1+\exp(-av))$:

$$\delta_j^L(t) = a \cdot y_j^L(t) \cdot \left(y_j(t) - y_j^L(t)\right)\left(1 - y_j^L(t)\right)$$

where $y_j(t) =$ target and $y_j^L(t) =$ output



## The Backpropagation Algorithm
Intuition: The Other Layers

- How can we compute the “error term” for the hidden layers $r < L$ when there is no “target vector” $\mathbf{y}$ for these layers?
- Instead, propagate back error values from output layer toward input layers, scaling with the weights
- Scaling with the weights characterizes how much of the error term each hidden unit is “responsible for”:

$$w_{jk}^r(t+1) = w_{jk}^r(t) + \eta\, y_k^{r-1}(t)\, \delta_j^r(t)$$

where

$$\delta_j^r(t) = f'\left(v_j^r(t)\right) \sum_{k=1}^{k_{r+1}} \delta_k^{r+1}(t)\, w_{kj}^{r+1}(t)$$

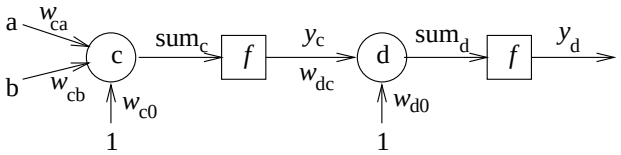
- Derivation comes from computing gradient of cost function w.r.t. $\mathbf{w}_j^r(t+1)$ via chain rule



## The Backpropagation Algorithm
Example

target = $y$
$f(x) = 1 / (1 + \exp(-x))$

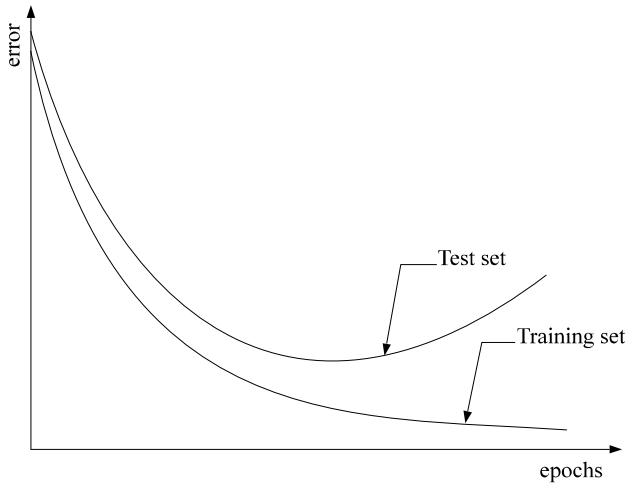
trial 1: a = 1, b = 0, y = 1
trial 2: a = 0, b = 1, y = 0




| | | | |
|---|---|---|---|
| eta | 0.3 | | |
| | | | |
| | trial 1 | trial 2 | |
| w_ca | 0.1 | 0.1008513 | 0.1008513 |
| w_cb | 0.1 | 0.1 | 0.0987985 |
| w_c0 | 0.1 | 0.1008513 | 0.0996498 |
| a | 1 | 0 | |
| b | 0 | 1 | |
| const | 1 | 1 | |
| sum_c | 0.2 | 0.2008513 | |
| y_c | 0.5498340 | 0.5500447 | |
| | | | |
| w_dc | 0.1 | 0.1189104 | 0.0964548 |
| w_d0 | 0.1 | 0.1343929 | 0.0935679 |
| sum_d | 0.1549834 | 0.1997990 | |
| y_d | 0.5386685 | 0.5497842 | |
| | | | |
| target | 1 | 0 | |
| delta_d | 0.1146431 | -0.136083 | |
| delta_c | 0.0028376 | -0.004005 | |
| | | | |
| | | | |
| delta_d(t) = y_d(t) * (y(t) - y_d(t)) * (1 - y_d(t)) | | | |
| delta_c(t) = y_c(t) * (1 - y_c(t)) * delta_d(t) * w_dc(t) | | | |
| w_dc(t+1) = w_dc(t) + eta * y_c(t) * delta_d(t) | | | |
| w_ca(t+1) = w_ca(t) + eta * a * delta_c(t) | | | |

## The Backpropagation Algorithm
Issues

- When to stop iterating through training set?
  - When weights don't change much
  - When value of cost function is small enough
  - Must also avoid overtraining






## The Backpropagation Algorithm
Issues
(cont'd)

- How to set learning rate $\eta$ ($\mu$ in text)?
  - Small values slow convergence
  - Large values might overshoot minimum
  - Can adapt it over time
- Might hit local minima that aren't very good; try re-running with new random weights



## Variations

- Can smooth oscillations of weight vector with momentum term $\alpha < 1$ that tends to keep it moving in the same direction as previous trials:

$$\Delta \mathbf{w}_j^r(t+1) = \alpha \Delta \mathbf{w}_j^r(t) + \eta\, y_k^{r-1}(t)\, \delta_j^r(t)$$
$$\mathbf{w}_j^r(t+1) = \mathbf{w}_j^r(t) + \Delta \mathbf{w}_j^r(t+1)$$

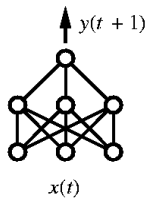
- Different training modes:
  - On-line (what we presented) has more randomness during training (might avoid local minima)
  - Batch mode (in text) averages gradients, giving better estimates and smoother convergence:
    - Before updating, first compute $\delta_j^r(t)$ for each vector $\mathbf{x}_t$, $t = 1, \ldots, N$

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) + \eta \sum_{t=1}^{N} \delta_j^r(t)\, \mathbf{y}^{r-1}(t)$$

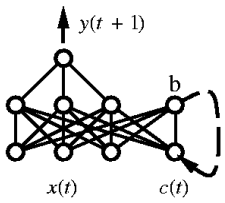


## Variations
(cont'd)

- A Recurrent network feeds output of e.g. layer $r$ to the input of some earlier layer $r' < r$
  - Allows predictions to be influenced by past predictions (for e.g. sequence data)




(a) Feedforward network




(b) Recurrent network

## Variations
(cont'd)

- Can implement a "backprop" scheme with EG
- Other nonlinear optimization schemes:
  - Conjugate gradient
  - Newton's method
  - Genetic algorithms
  - Simulated annealing
- Other cost functions, e.g. cross-entropy:

$$-\sum_{k=1}^{k_L}\left(\overbrace{y_k(t)}^{label}\ln\overbrace{\left(y_k^L(t)\right)}^{pred}+(1-y_k(t))\ln\left(1-y_k^L(t)\right)\right)$$

"blows up" if $y_k(t)\approx 1$ and $y_k^L(t)\approx 0$ or vice-versa (Section 4.8)



## Sizing the Network

- Before training, need to choose appropriate number of layers and size of each layer
  - Too small: Cannot learn what features make same classes similar and separate classes different
  - Too large: Adapts to details of the particular training set and cannot generalize well (called overfitting)
  - Also, increasing size increases complexity
- Approaches:
  - Analytical methods: Use knowledge of data to est. number of needed layers and neurons
  - Pruning techniques: Start with a large network and periodically remove weights and neurons that don't affect output much
  - Constructive techniques: Start with small netw. and periodically add neurons and wts

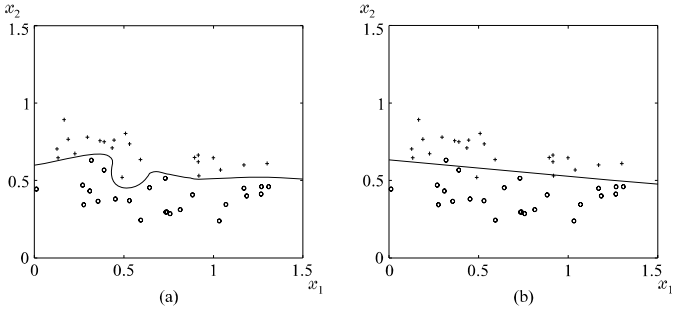


## Sizing the Network
Pruning Techniques [Also see Bishop, Sec. 9.5]

- Approach 1: Train with backprop, periodically computing effect of varying $w_i$ on cost func:
  - From Taylor series expansion (p. 109),

$$\overbrace{\delta J}^{cost\ change}\approx\frac{1}{2}\sum_i h_{ii}\,\delta w_i^2 \quad \text{where} \quad h_{ii}=\frac{\partial^2 J}{\partial^2 w_i}$$

  - If $h_{ii}w_i^2/2$ (saliency factor) small, then $w_i$ doesn't have much impact and is removed
  - Now continue training with backprop
- Example (Sec 4.10): 480 wts pruned to 25






## Sizing the Network
Pruning Techniques
(cont'd)
[Also see Bishop, Sec. 9.5]

- Approach 2: Train with backprop, but add to the cost function $J$ a term that penalizes large weights:

$$J' = J + \text{penalty}$$

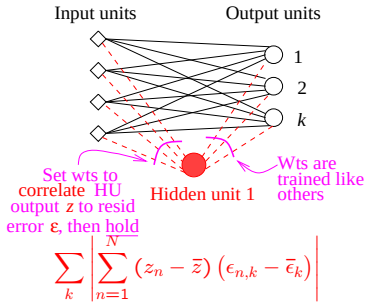
  - If $w_i$'s contribution to network output is small, then its share of $J$ is small
  - So penalty term dominates $w_i$'s share of $J'$, driving it down
  - Periodically prune weights that get too low

## Sizing the Network

Constructive Techniques

Cascade Correlation [Also Bishop, Sec. 9.5]

- Start with no hidden units and train
- After training, if residual error too high, then add a hidden unit:



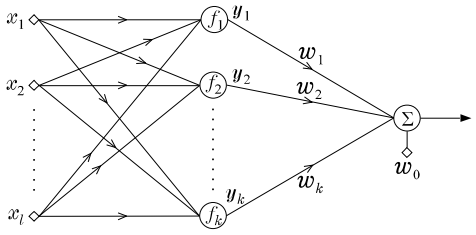

- Then continue training; if residual error still to high, add another hidden unit:
  - Same as HU1, but connect input units and HU1's output to inputs of HU2
- Limit the number added to avoid overfitting



## Generalized Linear Classifiers

Section 4.12

- In XOR problem, used linear threshold funcs. in hidden layer to map non-lin. sep. classes to new space where they were lin. sep.
- Output layer gave sep. hyperplane in new space
- Replace hidden-layer lin. thresh. funcs. with family of nonlinear functions $f_i : \Re^\ell \to \Re,\ i = 1, \ldots, k$
- Hidden layer maps $\mathbf{x} \in \Re^\ell$ to $\mathbf{y} = [f_1(\mathbf{x}), \ldots, f_k(\mathbf{x})]^T$ and output layer finds separating hyperplane:




- I.e. approximating separating surface as linear combination of interpolation functions:

$$g(\mathbf{x}) = w_0 + \sum_{i=1}^{k} w_i f_i(\mathbf{x})$$



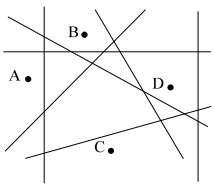
## Generalized Linear Classifiers

Cover's Theorem

- For arbitrary set of $N$ points, there are $2^N$ ways to classify them into $\omega_1$ and $\omega_2$ (i.e. $2^N$ dichotomies)
- If classification done by a single hyperplane, then the number of linear dichotomies is

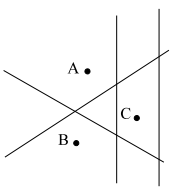
$$O(N,\ell) = 2\sum_{i=0}^{\ell} \binom{N-1}{i}$$
$$= 2^N \text{ if } N \le \ell + 1, \text{ else } < 2^N$$

14 linear dichotomies




(a)

8 linear dichotomies




(b)



## Generalized Linear Classifiers

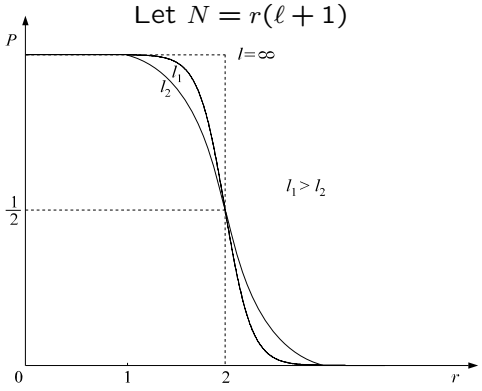
Cover's Theorem

(cont'd)

- Thus if dimensionality $\ell \ge N - 1$ then a perfect separating hyperplane is guaranteed to exist
- Otherwise ($N > \ell + 1$) the fraction of dichotomies that are linear dichotomies is

$$P = \frac{1}{2^{N-1}} \sum_{i=0}^{\ell} \binom{N-1}{i}$$

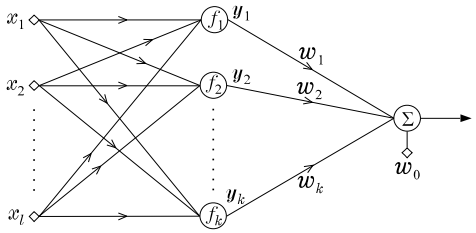
Let $N = r(\ell + 1)$




- For fixed $N$, mapping to higher dimensional space increases likelihood of $\exists$ of sep. hyperplane!

## Generalized Linear Classifiers

Polynomial Classifiers



- Approximate $g(\mathbf{x})$ by linear combination of up to order $r$ polynomials over components of $\mathbf{x}$
- E.g. for $r = 2$

$$g(\mathbf{x}) = w_0 + \underbrace{\sum_{i=1}^{\ell} w_i x_i}_{w_1 f_1 + \cdots + w_\ell f_\ell} + \underbrace{\sum_{i=1}^{\ell-1} \sum_{m=i+1}^{\ell} w_{im} x_i x_m}_{w_{\ell+1} f_{\ell+1} + \cdots + w_{k-\ell} f_{k-\ell}} + \underbrace{\sum_{i=1}^{\ell} w_{ii} x_i^2}_{w_{k-\ell+1} f_{k-\ell+1} + \cdots w_k f_k}, \quad k = \ell(\ell+3)/2$$

- For $\ell = 2$, $\mathbf{x} = [x_1, x_2]^T$ and

$$\mathbf{y} = \left[x_1, x_2, x_1 x_2, x_1^2, x_2^2\right]^T$$
$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{y} + w_0$$
$$\mathbf{w}^T = [w_1, w_2, w_{12}, w_{11}, w_{22}]$$



## Generalized Linear Classifiers

Polynomial Classifiers
(cont'd)

- In general, will use all terms of form $x_1^{p_1} x_2^{p_2} \cdots x_\ell^{p_\ell}$ for all $p_1 + \cdots + p_\ell \leq r$
- This gives size of $\mathbf{y}$ to be

$$k = \frac{(\ell + r)!}{r!\,\ell!},$$

so time to classify and update exponential in $(\ell + r)$

- Fortunately, EG's loss bound logarithmic in $k$, though run time still (in general) linear in $k$
  - Special cases can be made efficient with exact or approximate output computation



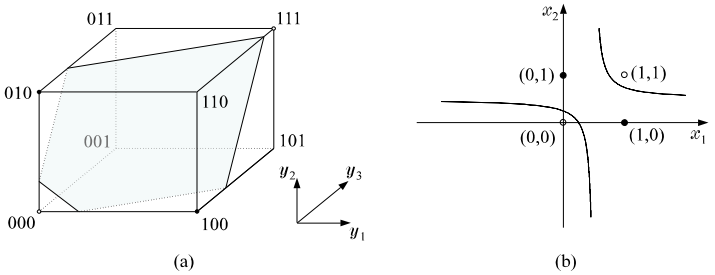
## Generalized Linear Classifiers

Polynomial Classifiers
Example: XOR

- Use $\mathbf{y} = [x_1, x_2, x_1 x_2]^T$

| Class | $[x_1, x_2]^T$ | $[y_1, y_2, y_3]^T$ |
|---|---|---|
| $\omega_1$ | $[0, 1]^T$ | $[0, 1, 0]^T$ |
| $\omega_1$ | $[1, 0]^T$ | $[1, 0, 0]^T$ |
| $\omega_2$ | $[0, 0]^T$ | $[0, 0, 0]^T$ |
| $\omega_2$ | $[1, 1]^T$ | $[1, 1, 1]^T$ |

$$g(\mathbf{y}) = y_1 + y_2 - 2y_3 - \frac{1}{4} \qquad g(\mathbf{x}) = -\frac{1}{4} + x_1 + x_2 - 2x_1 x_2$$



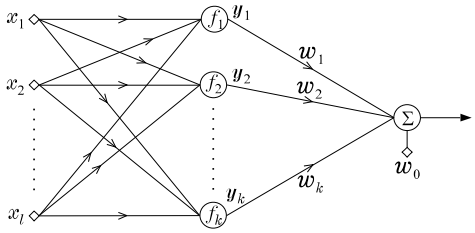

$> 0 \Rightarrow \mathbf{x} \in \omega_1$

$< 0 \Rightarrow \mathbf{x} \in \omega_2$



## Generalized Linear Classifiers

Radial Basis Function Networks



- Argument of func. $f_i$ is $\mathbf{x}$'s Euclidian distance from designated center $\mathbf{c}_i$, e.g.

$$f_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{c}_i\|_2^2}{2\sigma_i^2}\right)$$

- So

$$g(\mathbf{x}) = w_0 + \sum_{i=1}^{k} w_i \exp\left(-\frac{(\mathbf{x} - \mathbf{c}_i)^T (\mathbf{x} - \mathbf{c}_i)}{2\sigma_i^2}\right)$$

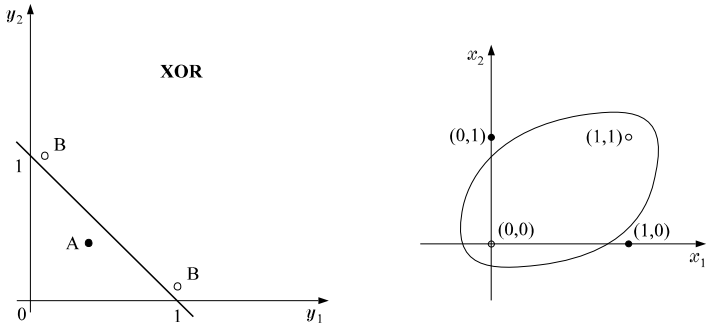
- Exponential decrease in increased distance gives a very localized activation response
- Related to nearest neighbor approaches since only $f_i$'s with centers near $\mathbf{x}$ will have significant output

## Generalized Linear Classifiers
Radial Basis Function Networks
Example: XOR

- $\mathbf{c}_1 = [1,1]^T$, $\mathbf{c}_2 = [0,0]^T$, $f_i(\mathbf{x}) = \exp\left(-\|\mathbf{x}-\mathbf{c}_i\|_2^2\right)$

| Class | $[x_1, x_2]^T$ | $[y_1, y_2]^T$ |
|---|---|---|
| $\omega_1$ (A) | $[0,1]^T$ | $[0.368, 0.368]^T$ |
| $\omega_1$ (A) | $[1,0]^T$ | $[0.368, 0.368]^T$ |
| $\omega_2$ (B) | $[0,0]^T$ | $[0.135, 1]^T$ |
| $\omega_2$ (B) | $[1,1]^T$ | $[1, 0.135]^T$ |

$g(\mathbf{y}) = y_1 + y_2 - 1$ $\quad$ $g(\mathbf{x}) = -1 + e^{-\|\mathbf{x}-\mathbf{c}_1\|_2^2} + e^{-\|\mathbf{x}-\mathbf{c}_2\|_2^2}$




$< 0 \Rightarrow \mathbf{x} \in \omega_1$
$> 0 \Rightarrow \mathbf{x} \in \omega_2$



## Generalized Linear Classifiers
Radial Basis Function Networks
Choosing the Centers

- Randomly select from the training set
  - Might work well if training set representative of probability distribution over data
- Learn the $\mathbf{c}_i$'s and $\sigma_i^2$'s via gradient descent
  - Frequently computationally complex
- First cluster the data (Chapters 11–16) and use results to find centers
- Use methods similar to constructive and pruning techniques when sizing neural network
  - Add new center when perceived as needed, delete unnecessary centers
  - E.g. if new input vector $\mathbf{x}$ far from all current centers and error high, then new center necessary, so add $\mathbf{x}$ as new center

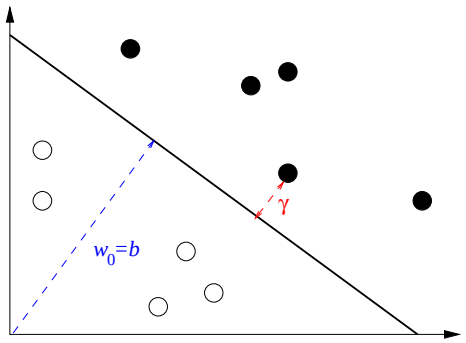


## Support Vector Machines
[See refs. on slides page]

- Introduced in 1992
- State-of-the-art technique for classification and regression
- Techniques can also be applied to e.g. clustering and principal components analysis
- Similar to polynomial classifiers and RBF networks in that it remaps inputs and then finds a hyperplane
  - Main difference is how it works
- Features of SVMs:
  - Maximization of margin
  - Duality
  - Use of kernels
  - Use of problem convexity to find classifier



## Support Vector Machines
Margins




- A hyperplane's margin $\gamma$ is the shortest distance from it to any training vector
- Intuition: larger margin $\Rightarrow$ higher confidence in classifier's ability to generalize
  - Guaranteed generalization error bound in terms of $1/\gamma^2$
- Definition assumes linear separability (more general definitions exist that do not)

## Support Vector Machines
Maximum-Margin Perceptron Algorithm

- $\mathbf{w}(0) \leftarrow \mathbf{0}$, $b(0) \leftarrow 0$, $k \leftarrow 0$, $R \leftarrow \max_{1 \le i \le N} \|\mathbf{x}_i\|_2$ ($R =$ radius of ball centered at origin containing training vectors), $y_i \in \{-1, +1\}\ \forall i$
- Update slope same as before, update offset differently
- While mistakes are made on training set
  - For $i = 1$ to $N$ ($=$ # training vectors)
    - If $y_i\,(\mathbf{w}_k \cdot \mathbf{x}_i + b_k) \le 0$
      - $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \eta\, y_i\, \mathbf{x}_i$
      - $b_{k+1} \leftarrow b_k + \eta\, y_i\, R^2$
      - $k \leftarrow k + 1$
- Final predictor: $h(\mathbf{x}) = \text{sgn}\,(\mathbf{w}_k \cdot \mathbf{x} + b_k)$



## Support Vector Machines
Duality

- Another way of representing predictor:

$$h(\mathbf{x}) = \text{sgn}\,(\mathbf{w} \cdot \mathbf{x} + b) = \text{sgn}\left(\sum_{i=1}^{N} (\alpha_i\, y_i\, \mathbf{x}_i) \cdot \mathbf{x} + b\right)$$
$$= \text{sgn}\left(\sum_{i=1}^{N} \alpha_i\, y_i\, (\mathbf{x}_i \cdot \mathbf{x}) + b\right)$$

($\alpha_i =$ # mistakes on $\mathbf{x}_i$, $\eta > 0$ ignored)

- So perceptron alg has equivalent dual form:
- $\alpha \leftarrow \mathbf{0}$, $b \leftarrow 0$, $R \leftarrow \max_{1 \le i \le N} \|\mathbf{x}_i\|_2$
- While mistakes are made in For loop
  - For $i = 1$ to $N$ ($=$ # training vectors)
    - If $y_i \left(\sum_{j=1}^{N} \alpha_j\, y_j\, \left(\mathbf{x}_j \cdot \mathbf{x}_i\right) + b\right) \le 0$
      - $\alpha_i \leftarrow \alpha_i + 1$
      - $b \leftarrow b + y_i\, R^2$
- Now data only in dot products



## Kernels

- Duality lets us remap to many more features!
- Let $\phi : \Re^\ell \rightarrow F$ be nonlinear map of f.v.s, so

$$h(\mathbf{x}) = \text{sgn}\left(\sum_{i=1}^{N} \alpha_i\, y_i\, (\phi\,(\mathbf{x}_i) \cdot \phi\,(\mathbf{x})) + b\right)$$

- Can we compute $\phi\,(\mathbf{x}_i) \cdot \phi\,(\mathbf{x})$ without evaluating $\phi\,(\mathbf{x}_i)$ and $\phi\,(\mathbf{x})$? YES!
- $\mathbf{x} = [x_1, x_2]$, $\mathbf{z} = [z_1, z_2]$:

$$(\mathbf{x} \cdot \mathbf{z})^2 = (x_1\, z_1 + x_2\, z_2)^2$$
$$= x_1^2\, z_1^2 + x_2^2\, z_2^2 + 2\, x_1\, x_2\, z_1\, z_2$$
$$= \underbrace{\left[x_1^2, x_2^2, \sqrt{2}\, x_1\, x_2\right]}_{\phi(\mathbf{x})} \cdot \left[z_1^2, z_2^2, \sqrt{2}\, z_1\, z_2\right]$$

- LHS requires 2 mults + 1 squaring to compute, RHS takes 3 mults
- In general, $(\mathbf{x} \cdot \mathbf{z})^d$ takes $\ell$ mults + 1 expon., vs. $\binom{\ell+d-1}{d} \ge \left(\frac{\ell+d-1}{d}\right)^d$ mults if compute $\phi$ first



## Kernels
(cont'd)

- In general, a kernel is a function $K$ such that $\forall\, \mathbf{x}, \mathbf{z},\ K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{z})$
- Typically start with kernel and take the feature mapping that it yields
- E.g. Let $\ell = 1, \mathbf{x} = x, \mathbf{z} = z$, $K(x, z) = \sin(x - z)$
- By Fourier expansion,

$$\sin(x - z) = a_0 + \sum_{n=1}^{\infty} a_n \sin(n\, x) \sin(n\, z) + \sum_{n=1}^{\infty} a_n \cos(n\, x) \cos(n\, z)$$

for Fourier coeficients $a_0, a_1, \ldots$

- This is the dot product of two infinite sequences of nonlinear functions:

$$\{\phi_i(x)\}_{i=0}^{\infty} = [1, \sin(x), \cos(x), \sin(2x), \cos(2x), \ldots]$$

- I.e. there are an infinite number of features in this remapped space!

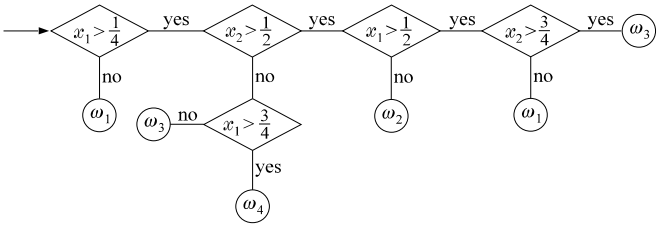
## Support Vector Machines

Finding a Hyperplane

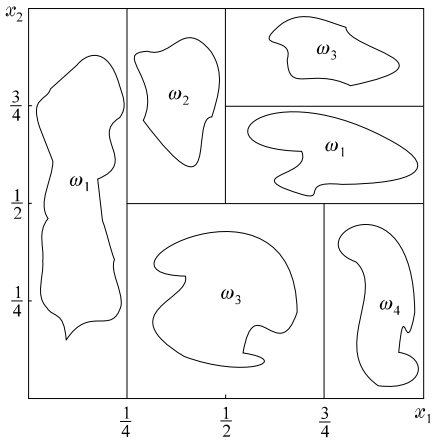
- Can show [Cristianini & Shawe-Taylor] that if data linearly separable in remapped space, then get maximum margin classifier by minimizing $\mathbf{w} \cdot \mathbf{w}$ subject to $y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$
- Can reformulate this into a convex quadratic program, which can be solved optimally, i.e. won't encounter local optima
- Can always find a kernel that will make training set linearly separable, but beware of choosing a kernel that is too powerful (overfitting)
- If kernel doesn't separate, can optimize subject to $y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i$, where $\xi_i$ are slack variables that soften the margin (can still solve optimally)
- If number of training vectors is very large, may opt to approximately solve these problems to save time and space
- Use e.g. gradient ascent and sequential minimal optimization (SMO) [Cristianini & Shawe-Taylor]



## Decision Trees [Also Mitchell, ch. 3]




- Start at root and work down tree until leaf reached; output that classification
- E.g. $\mathbf{x} = [1/2, 1/4]^T$ classified as $\omega_3$






## Decision Trees

Learning Good Trees [Also Mitchell, ch. 3]

- Feature at root is one that yields highest information gain, equivalent to max. reduction of entropy (class impurity) in training data:

$S$ = set of $N$ feature vectors $\quad N_i$ = number in $\omega_i$

$$p_i = N_i/N \qquad Ent(S) = \sum_{i=1}^{M} -p_i \log_2 (p_i)$$

- First partition along dimensions into set $A$ of features and places where classes change, e.g.

$$A = \{(x_1, 0), (x_1, 1/4), (x_1, 1/2), (x_1, 3/4), (x_2, 0), (x_2, 1/2), (x_2, 3/4)\}$$

- For $a = (x_i, b) \in A$, define

$$S_a = \{\mathbf{x} \in S : x_i > b\} \qquad S'_a = \{\mathbf{x} \in S : x_i \leq b\}$$

$$Gain(S, a) = Ent(S) - \underbrace{\left( \frac{|S_a|}{|S|} Ent(S_a) + \frac{|S'_a|}{|S|} Ent(S'_a) \right)}_{=0 \text{ for } (x_1, 1/4)}$$

- Choose $a$ from $A$ that maximizes $Gain$, place it at root, then recursively call on $S_a$ and $S'_a$
- Forms basis of algorithms ID3 and C4.5
- Can avoid overfitting by pruning