**Connectionist Models** 

# Consider humans:

- Total number of neurons  $\approx 10^{10}$
- Neuron switching time  $\approx 10^{-3}$  second (vs.  $10^{-10}$ )
- Connections per neuron  $\approx 10^4 10^5$
- Scene recognition time  $\approx 0.1$  second
- 100 inference steps doesn't seem like enough
- $\Rightarrow$  much parallel computation

Properties of artificial neural nets (ANNs):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

Strong differences between ANNs for ML and ANNs for biological modeling

3

6

**Decision Surface of Perceptron/Winnow** 



Represents some useful functions

• What weights represent  $g(x_1, x_2) = AND(x_1, x_2)$ ?

But some functions not representable

- I.e. those not linearly separable
- Therefore, we'll want networks of neurons

CSCE 478/878 Lecture 4: Artificial Neural Networks

Stephen D. Scott (Adapted from Tom Mitchell's slides)

When to Consider Neural Networks

1

4

- Input is high-dimensional discrete- or real-valued (e.g. raw sensor input)
- · Output is discrete- or real-valued
- · Output is a vector of values
- Possibly noisy data
- · Form of target function is unknown
- · Human readability of result is unimportant
- · Long training times acceptable

# Examples:

- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction

# The Perceptron & Winnow



$o(x_1,\ldots,x_n)=$	$\left\{ \begin{array}{c} +1\\ -1 \end{array} \right.$	$\begin{array}{l} \text{if } w_0+w_1x_1+\cdots+w_nx_n > 0\\ \text{otherwise} \end{array}$
----------------------	--	---

 $o(\vec{x}) = \begin{cases} +1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise} \end{cases}$ 

2



- - (sometimes use 0 instead of -1)

# Sometimes we'll use simpler vector notation:

Outline

• Threshold units: Perceptron, Winnow

Multilayer networks

Backpropagation

• Support Vector Machines

· Gradient descent/exponentiated gradient

# Perceptron vs. Winnow

# Perceptron Training Rule

 $w_i \leftarrow w_i + \Delta w_i^{add}$ , where  $\Delta w_i^{add} = \eta (t - o) x_i$ 

# and

- $t = c(\vec{x})$  is target value
- o is perceptron output
- $\eta$  is small constant (e.g. 0.1) called learning rate

I.e. if (t - o) > 0 then increase  $w_i$  w.r.t.  $x_i$ , else decrease

Can prove rule will converge if training data is linearly separable and  $\eta$  sufficiently small

## Winnow Training Rule

 $w_i \gets w_i \cdot \Delta w_i^{mult}, \text{ where } \Delta w_i^{mult} = \alpha^{(t-o)x_i}$  and  $\alpha > 1$ 

I.e. use multiplicative updates vs. additive updates

Problem: Sometimes negative weights are required

- Maintain two weight vectors  $\vec{w}^+$  and  $\vec{w}^-$  and replace  $\vec{w} \cdot \vec{x}$  with  $(\vec{w}^+ \vec{w}^-) \cdot \vec{x}$
- Update  $\vec{w}^+$  and  $\vec{w}^-$  independently as above, using  $\Delta w_i^+ = \alpha^{(t-o)x_i}$  and  $\Delta w_i^- = 1/\Delta w_i^+$

Can also guarantee convergence

# Winnow works well when most attributes irrelevant, i.e. when optimal weight vector $\vec{w}^*$ is sparse (many 0 entries)

E.g. let examples  $\vec{x} \in \{0, 1\}^n$  be labeled by a *k*-disjunction over *n* attributes,  $k \ll n$ 

- Remaining n k are irrelevant
- E.g.  $c(x_1, \dots, x_{150}) = x_5 \lor x_9 \lor \neg x_{12}, n = 150, k = 3$
- For disjunctions, number of prediction mistakes (in online model) is O (k log n) for Winnow and (in worst case) Ω (kn) for Perceptron
- So in worst case, need exponentially fewer for learning with Winnow than Perceptron

Bound is only for disjunctions, but improvement for learning with irrelevant attributes is often true

When  $\vec{w}^*$  not sparse, sometimes Perceptron better

Also, have proofs for <u>agnostic</u> error bounds for both algorithms

## Gradient Descent and Exponentiated Gradient

7

10

- Useful when linear separability impossible but still want to minimize training error
- Consider simpler linear unit, where

 $o = w_0 + w_1 x_1 + \dots + w_n x_n$ 

(i.e. no threshold)

- For moment, assume that we update weights after seeing each example  $\vec{x}_d$
- For each example, want to compromise between <u>correctiveness</u> and <u>conservativeness</u>
  - Correctiveness: Tendency to improve on  $\vec{x}_d$  (reduce error)
  - Conservativeness: Tendency to keep  $\vec{w}_{d+1}$  close to  $\vec{w}_d$  (minimize distance)
- Use cost function that measures both:

$$U(\vec{w}) = dist\left(\vec{w}_{d+1}, \vec{w}_{d}\right) + \eta \operatorname{error}\left(t_{d}^{\text{currex, new wis}}_{d+1} \cdot \vec{x}_{d}\right)$$

# **Gradient Descent**

$$U(\vec{w}) = \underbrace{\|\vec{w}_{d+1} - \vec{w}_d\|_2^2}_{i=1} + \underbrace{\stackrel{coef.}{\hat{\eta}} \underbrace{\stackrel{coef.}{(t_d - \vec{w}_{d+1} \cdot \vec{x}_d)^2}}_{i=1} = \sum_{i=1}^n \left( w_{i,d+1} - w_{i,d} \right)^2 + \eta \left( t_d - \sum_{i=1}^n w_{i,d+1} x_{i,d} \right)^2$$

Take gradient w.r.t.  $\vec{w}_{d+1}$  and set to  $\vec{0}$ :

$$0 = 2\left(w_{i,d+1} - w_{i,d}\right) - 2\eta\left(t_d - \sum_{i=1}^n \frac{w_{i,d+1}}{x_{i,d}}\right) x_{i,d}$$

Approximate with

$$0 = 2\left(w_{i,d+1} - w_{i,d}\right) - 2\eta\left(t_d - \sum_{i=1}^n w_{i,d} x_{i,d}\right) x_{i,d} ,$$

which yields

$$w_{i,d+1} = w_{i,d} + \overbrace{\eta (t_d - o_d) x_{i,d}}^{\Delta w_{i,d}^{aaa}}$$

11

8

# Exponentiated Gradient

# Conserv. portion uses unnormalized relative entropy:

$$U(\vec{w}) = \overbrace{\sum_{i=1}^{n} \left( w_{i,d} - w_{i,d+1} + w_{i,d+1} \ln \frac{w_{i,d+1}}{w_{i,d}} \right)}^{conserv.} + \overbrace{\eta}^{conserv.} (\underbrace{t_d - \vec{w}_{d+1} \cdot \vec{x}_d}^{conserv.})^2$$

Take gradient w.r.t.  $\vec{w}_{d+1}$  and set to  $\vec{0}$ :

$$0 = \ln \frac{w_{i,d+1}}{w_{i,d}} - 2\eta \left( t_d - \sum_{i=1}^n \frac{w_{i,d+1}}{x_{i,d}} \right) x_{i,d}$$

# Approximate with

$$0 = \ln \frac{w_{i,d+1}}{w_{i,d}} - 2\eta \left( t_d - \sum_{i=1}^n \frac{w_{i,d} x_{i,d}}{w_{i,d}} \right) x_{i,d} ,$$

$$\begin{array}{l} \mbox{which yields (for $\eta = \ln \alpha/2$)} \\ w_{i,d+1} &= w_{i,d} \exp\left(2\eta \left(t_d - o_d\right) x_{i,d}\right) = w_{i,d} \overbrace{\alpha^{(t_d - o_d) x_{i,d}}}^{\Delta w_{i,d}^{mult}} \end{array}$$

13



 Can't represent with a single linear separator, but can with <u>intersection of two</u>:

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

$$\begin{split} g_2(\vec{x}) &= 1 \cdot x_1 + 1 \cdot x_2 - 3/2 \\ \mathrm{pos} &= \left\{ \vec{x} \in \Re^\ell : g_1(\vec{x}) > 0 \text{ } \underbrace{\mathrm{AND}} g_2(\vec{x}) < 0 \right\} \end{split}$$

$$\operatorname{neg} = \left\{ \vec{x} \in \Re^{\ell} : g_1(\vec{x}), g_2(\vec{x}) < 0 \text{ } \underbrace{\operatorname{OR}}_{16} g_1(\vec{x}), g_2(\vec{x}) > 0 \right\}$$

### Implementation Approaches

- Can use rules on previous slides on an example-byexample basis, sometimes called incremental, stochastic, or <u>on-line</u> GD/EG
- Has a tendency to "jump around" more in searching, which helps avoid getting trapped in local minima
- Alternatively, can use <u>standard</u> or <u>batch</u> GD/EG, in which the classifier is evaluated over all training examples, summing the error, and then updates are made
  - I.e. sum up  $\Delta w_i$  for all examples, but don't update  $w_i$  until summation complete (p. 93, Table 4.1)
  - This is an inherent averaging process and tends to give better estimate of the gradient

The XOR Problem (cont'd)

	0	if $g_i(\vec{x}) < 0$
Let $g_i = $	1	otherwise

(	Class	$(x_1, x_2)$	$g_1(\vec{x})$	$y_1$	$g_2(\vec{x})$	$y_2$	
_	pos	B: (0,1)	1/2	1	-1/2	0	
	pos	C:(1,0)	1/2	1	-1/2	0	
	neg	A: (0,0)	-1/2	0	-3/2	0	
	neg	D:(1,1)	3/2	1	1/2	1	

• Now feed  $y_1, y_2$  into:



### Remarks

- Perceptron and Winnow update weights based on thresholded output, while GD and EG use unthresholded outputs
- P/W converge in finite number of steps to perfect hyp if data linearly separable; GD/EG work on non-linearly separable data, but only converge asymptotically (to wts with minimum squared error)
- As with P vs. W, EG tends to work better than GD when many attributes are irrelevant
  - Allows the addition of attributes that are nonlinear combinations of original ones, to work around linear sep. problem (perhaps get linear separability in new, higher-dimensional space)
  - E.g. if two attributes are  $\boldsymbol{x}_1$  and  $\boldsymbol{x}_2,$  use as EG inputs

 $\vec{x} = \begin{bmatrix} x_1, x_2, x_1 x_2, x_1^2, x_2^2 \end{bmatrix}$ 

Also, both have provable agnostic results

The XOR Problem (cont'd)

 In other words, we remapped all vectors x to 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

14

# Generally Handling Nonlinearly Separable Data

• By adding up to 2 <u>hidden layers</u> of perceptrons, can represent any <u>union</u> of intersection of halfspaces



• Problem: The above is still defined linearly





# $\sigma(net)$ is the logistic function

 $\frac{1}{1+e^{-net}}$ 

(a type of sigmoid function)

# Squashes *net* into [0, 1] range

### Nice property:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

# We can derive GD/EG rules to train

- One sigmoid unit
- Multilayer networks of sigmoid units ⇒ Backpropagation

# **GD/EG for Sigmoid Unit**

- First note that conservativeness and correctiveness are only additively related ⇒ derivatives always independent
- · Thus in general get

$$\begin{split} w_{i,d+1} &= w_{i,d} - \frac{\eta}{2} \, \frac{\partial \, correc}{\partial w_{i,d}} \, \text{ for GD} \\ \\ w_{i,d+1} &= w_{i,d} \exp\left(-\eta \, \frac{\partial \, correc}{\partial w_{i,d}}\right) \, \text{ for EG} \end{split}$$

0

• So all we have to do is define an error function, take its gradient, and substitute into the equations

21

# GD/EG for Sigmoid Unit (cont'd)

Return to book notation, where correctiveness is:

$$E(\vec{w}_d) = \frac{1}{2} (t_d - o_d)^2$$

(folding 1/2 of correctiveness into error func)

Thus 
$$\frac{\partial E}{\partial w_{i,d}} = \frac{\partial}{\partial w_{i,d}} \frac{1}{2} (t_d - o_d)^2$$
  
=  $\frac{1}{2} 2 (t_d - o_d) \frac{\partial}{\partial w_{i,d}} (t_d - o_d) = (t_d - o_d) \left( -\frac{\partial o_d}{\partial w_{i,d}} \right)$ 

Since  $o_d$  is a function of  $net_d = \vec{w}_d \cdot \vec{x}_d$ ,

$$\begin{array}{lll} \frac{\partial E}{\partial w_{i,d}} &=& -(t_d - o_d) \; \frac{\partial o_d}{\partial net_d} \; \frac{\partial net_d}{\partial w_{i,d}} \\ &=& -(t_d - o_d) \; \frac{\partial \sigma \left(net_d\right)}{\partial net_d} \; \frac{\partial net_d}{\partial w_{i,d}} \\ &=& -(t_d - o_d) \; o_d \left(1 - o_d\right) x_{i,d} \end{array}$$

$$w_{i,d+1} = w_{i,d} + \eta o_d (1 - o_d) (t_d - o_d) x_{i,d}$$
 for GD

$$w_{i,d+1} = w_{i,d} \exp \left( 2\eta \, o_d \, (1 - o_d) \, (t_d - o_d) \, x_{i,d} \right)$$
 for EG

Multilayer Networks



Use sigmoid units since continuous and differentiable

$$E_d = E(\vec{w}_d) = \frac{1}{2} \sum_{k \in outputs} (t_{k,d} - o_{k,d})^2$$



- Adjust wt  $w_{ii,d}$  according to  $E_d$  as before
- For output units, this is easy since contribution of w<sub>ji,d</sub> to E<sub>d</sub> when j is an output unit is the same as for single neuron case\*, i.e.

$$\begin{split} & \frac{\partial E_d}{\partial w_{ji,d}} = -\left(t_{j,d} - o_{j,d}\right) o_{j,d} \left(1 - o_{j,d}\right) x_{ji,d} = -\delta_j x_{ji,d} \\ & \text{where } \delta_j = -\frac{\partial E_d}{\partial net_j} = \frac{\text{error term}}{\text{of unit } j} \end{split}$$

\*This is because all other outputs are constants w.r.t.  $w_{ji,d}$ 

22

### **Backpropagation Algorithm**

Initialize all weights to small random numbers.

Until termination condition satisfied, Do

- · For each training example, Do
  - 1. Input the training example to the network and compute the network outputs
  - 2. For each output unit k

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

3. For each hidden unit h

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in down(h)} w_{k,h} \delta_k$$

### 4. Update each network weight $w_{ii}$

$$w_{j,i} \leftarrow w_{j,i} + \Delta w_{j,i}$$

where

$$\Delta w_{j,i} = \eta \, \delta_j x_{j,i}$$

27

Overfitting Error versus weight updates (example 1) 0.01 0.009 Training set error Validation set error 0.008 0.007 0.006 0.005 0.004 0.003 0.002 5000 10000 15000 20000 0 Number of weight updates Error versus weight updates (example 2) 0.08 0.07 Training set error Validation set error 0.06 0.05 E 0.04 0.03 0.02 0.01 0 1000 2000 4000 5000 0 3000 6000 Number of weight updates

## Danger of stopping too soon!

### Works for arbitrary number of hidden layers

25

# The Backpropagation Algorithm Example

	targ f( x	get = y y = 1/(1 + 1)	exp(-x)	trial 1 trial 2	a = 1, b =	= 0, y =
a√ b∕	v v ca w cb	$c \xrightarrow{sum_c} w$	$f \frac{y_c}{w_c}$	$d = \frac{1}{1c} \frac{s_1}{s_1}$	$m_{d} f$	y <sub>d</sub> >
		1		1		
		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			
		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		
		у_d	0.5386685	0.5497842		
		target	1	0		
		deita_d	0.1146431	-0.136083		
		delta_c	0.0028376	-0.004005		
		delta d(t) =	v d(t) * (v(t) -	v d(t)) * (1 -	v d(t))	
		delta c(t) = 1	c(t)) * delta	f(t) * w dc(t)		
		$d_{c(t+1)} = $	$w dc(t) \perp ota$	* v c(t) * delta_t	ta d(t)	
		$w_{ca(t+1)} = 0$	$W_{ca(t)} + eta$	* a * delta ci	(t)	
		[oa((+1)) =	+ Old	a aona_o		

# Training

# Hidden Units (cont'd)

The impact that  $w_{ji,d}$  has on  $E_d$  is only through  $net_j$  and units immediately "downstream" of *j*:

$$\frac{\partial E_d}{\partial w_{ji,d}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji,d}} = x_{ji} \sum_{k \in down(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

 $= x_{ji} \sum_{k \in down(j)} -\delta_k \frac{\partial net_k}{\partial net_j} = x_{ji} \sum_{k \in down(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$ 

$$= x_{ji} \sum_{k \in down(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j} = x_{ji} \sum_{k \in down(j)} -\delta_k w_{kj} o_j \left(1 - o_j\right)$$

**Remarks on Backprop** 

- When to stop training? When weights don't change much, error rate sufficiently low, etc. (be aware of overfitting: use validation set)
- Cannot ensure convergence to global minimum due to myriad local minima, but tends to work well in practice (can re-run with new random weights)
- · Generally training very slow (thousands of iterations), use is very fast
- Setting  $\eta$ : Small values slow convergence, large values might overshoot minimum, can adapt it over time
- Can add momentum term  $\alpha < 1$  that tends to keep the updates moving in the same direction as previous trials:

 $\Delta w_{ji,d+1} = \eta \, \delta_{j,d+1} \, x_{ji,d+1} + \alpha \, \Delta w_{ji,d}$ Can help move through small local minima to better ones & move along flat surfaces

26

# Training Hidden Units

• How can we compute the error term for hidden layers when there is no target output  $\vec{t}$  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"

• Alternative error function: cross entropy

 $E_d = \sum_{k \in outputs} \left( t_{k,d} \ln o_{k,d} + \left( 1 - t_{k,d} \right) \ln \left( 1 - o_{k,d} \right) \right)$ 

"blows up" if  $t_{k,d}\approx 1$  and  $o_{k,d}\approx 0$  or vice-versa (vs. squared error, which is always in [0, 1])

• Can penalize large weights to make space more linear and reduce risk of overfitting:

 $E_d = \frac{1}{2} \sum_{k \in outputs} (t_{kd} - o_{ok})^2 + \gamma \sum_{i,j} w_{ji,d}^2$ 

- Representational power: Any boolean func. can be represented with 2 layers, any bounded, continuous func. can be rep. with arbitrarily small error with 2 layers, any func. can be rep. with arbitrarily small error with 3 layers
  - Number of required units may be large
  - GD/EG may not be able to find the right weights



- 1. Hyp. space is set of all weight vectors (continuous vs. discrete of decision trees)
- Search via GD/EG: Possible because error function and output functions are continuous & differentiable
- 3. Inductive bias: (Roughly) smooth interpolation between data points



Support Vector Machines

Perceptron Algorithm Revisited

•  $\vec{w}(0) \leftarrow \vec{0}, b(0) \leftarrow 0, k \leftarrow 0, y_i \in \{-1, +1\} \forall i$ 

· While mistakes are made on training set

\* If  $y_i (\vec{w}_k \cdot \vec{x}_i + b_k) \leq 0$ 

 $b_{k+1} \leftarrow b_k + \eta y_i$ 

 $\cdot k \leftarrow k + 1$ 

 $\cdot \vec{w}_{k+1} \leftarrow \vec{w}_k + \eta y_i \vec{x}_i$ 

• Final predictor:  $h(\vec{x}) = \operatorname{sgn}(\vec{w}_k \cdot \vec{x} + b_k)$ 

- For i = 1 to N (= # training vectors)

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 ANNs, 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 <u>convexity</u> to find classifier (often without local minima)



Duality

• Another way of representing predictor:

$$h(\vec{x}) = \operatorname{sgn}\left(\vec{w} \cdot \vec{x} + b\right) = \operatorname{sgn}\left(\eta \sum_{i=1}^{N} \left(\alpha_{i} y_{i} \vec{x}_{i}\right) \cdot \vec{x} + b\right)$$
$$= \operatorname{sgn}\left(\eta \sum_{i=1}^{N} \alpha_{i} y_{i} \left(\vec{x}_{i} \cdot \vec{x}\right) + b\right)$$

 $(\alpha_i = \# \text{ mistakes on } \vec{x_i})$ 

- So perceptron alg has equivalent <u>dual</u> form: –  $\vec{\alpha} \leftarrow \vec{0}, b \leftarrow 0$ ,
  - While mistakes are made in For loop

\* For 
$$i = 1$$
 to  $N$  (= # training vectors)

$$\text{If } y_i \left( \eta \sum_{j=1}^N \alpha_j \, y_j \left( \vec{x}_j \cdot \vec{x}_i \right) + b \right) \leq 0$$
$$\alpha_i \leftarrow \alpha_i + 1$$
$$b \leftarrow b + \eta \, y_i$$

Now data only in dot products



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

31

32

Kernels

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

$$h(\vec{x}) = \operatorname{sgn}\left(\sum_{i=1}^{N} \alpha_{i} y_{i} \left(\vec{\phi}\left(\vec{x}_{i}\right) \cdot \vec{\phi}\left(\vec{x}\right)\right) + b\right)$$

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

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

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

37

40

• Can show [Cristianini & Shawe-Taylor] that if data linearly separable in remapped space, then get maximum margin classifier by minimizing  $\vec{w} \cdot \vec{w}$  subject to  $y_i\left(\vec{w}\cdot\vec{x}_i+b\right)>1$ 

Support Vector Machines

Finding a Hyperplane

• Can reformulate this in dual form as a convex quadratic program that can be solved optimally, i.e. won't encounter local optima:

$$\begin{array}{ll} \underset{\alpha}{\text{maximize}} & \sum\limits_{i=1}^{m} \alpha_i - \frac{1}{2} \sum\limits_{i,j} \alpha_i \, \alpha_j \, y_i \, y_j \, k(\vec{x}_i, \vec{x}_j) \\ \text{s.t.} & \alpha_i \geq 0, i = 1, \dots, m \\ & \sum\limits_{i=1}^{m} \alpha_i \, y_i = 0 \end{array}$$

· After optimization, we can label new vectors with the decision function:

$$f(\vec{x}) = \operatorname{sgn}\left(\sum_{i=1}^{m} \alpha_i y_i k(\vec{x}, \vec{x}_i) + b\right)$$

· Can always find a kernel that will make training set linearly separable, but beware of choosing a kernel that is too powerful (overfitting)

# Kernels (cont'd)

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

si

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

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!

Kernels (cont'd)

- · Commonly-used kernels:
  - Polynomial:

$$K_{poly}(x, x') = (x \cdot x' + c)^d$$

- Gaussian Radial Basis Function (RBF):

$$K_{RBF}(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

- Hyperbolic tangent (sigmoid):

 $K_{sig}(x, x') = \tanh(\kappa(x \cdot x') + \theta)$ 

39

• Also have ones for structured data: e.g. graphs, trees, sequences, and sets of points

38

Support Vector Machines

# Finding a Hyperplane (cont'd)

- If kernel doesn't separate, can soften the margin with slack variables  $\xi_i$ :  $\begin{array}{l} \underset{\vec{w}, b, \boldsymbol{\xi}}{\text{minimize}} & \|\vec{w}\|^2 + C \sum_{i=1}^m \xi_i \\ \text{s.t.} & y_i((\vec{x}_i \cdot \vec{w}) + b) \geq 1 - \xi_i, \ i = 1, \dots, m \\ & \xi_i \geq 0, \ i = 1, \dots, m \end{array}$
- The dual is similar to that for hard margin:

$$\begin{array}{ll} \underset{\boldsymbol{\alpha}}{\text{maximize}} & \sum_{i=1}^{m} \alpha_i - \sum_{i,j} \alpha_i \, \alpha_j \, y_i \, y_j \, k(x_i, x_j) \\ \text{s.t.} & 0 \leq \alpha_i \leq C, \ i = 1, \dots, m \\ & \sum_{i=1}^{m} \alpha_i \, y_i = 0 \end{array}$$

- · 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]
- When done, can throw out non-SVs