CSCE 478/878 Lecture 5: Artificial Neural Networks and Support Vector Machines

Stephen Scott

(Adapted from Ethem Alpaydin and Tom Mitchell)

sscott@cse.unl.edu

Introduction

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 ≈ 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

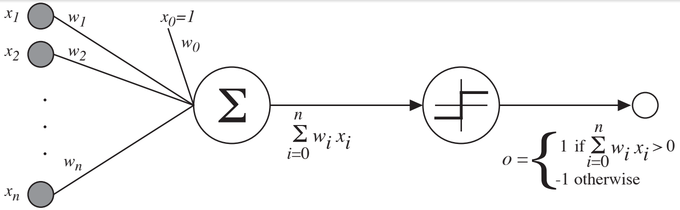
When to Consider ANNs

- 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

Outline

- Linear threshold units: Perceptron
- Gradient descent
- Multilayer networks
- Backpropagation
- Support Vector Machines

The Perceptron



$$y = o(x_1, \ldots, x_n) = \begin{cases} +1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

(sometimes use 0 instead of -1)

Sometimes we'll use simpler vector notation:

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

Decision Surface

(a) (b)

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

Perceptron Training Rule

$$w_j^{t+1} \leftarrow w_j^t + \Delta w_j^t \text{ , where } \Delta w_j^t = \eta \left(r^t - y^t \right) x_j^t$$

and

- r^t is label of training instance t
- y^t is perceptron output on training instance t
- η is small constant (e.g., 0.1) called *learning rate*

I.e., if $(r^t - y^t) > 0$ then increase w_j^t w.r.t. x_j^t, else decrease

Can prove rule will converge if training data is linearly separable and η sufficiently small

Where Does the Training Rule Come From?

- Consider simpler *linear unit*, where output

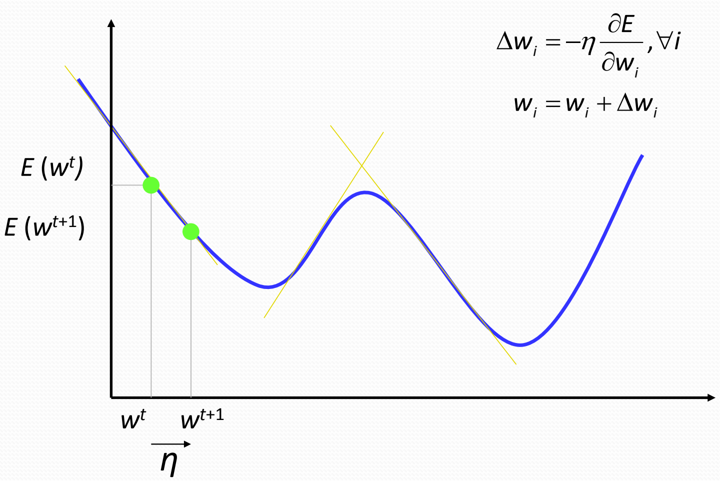
$$y^t = w_0^t + w_1^t x_1^t + \cdots + w_n^t x_n^t$$

(i.e., no threshold)

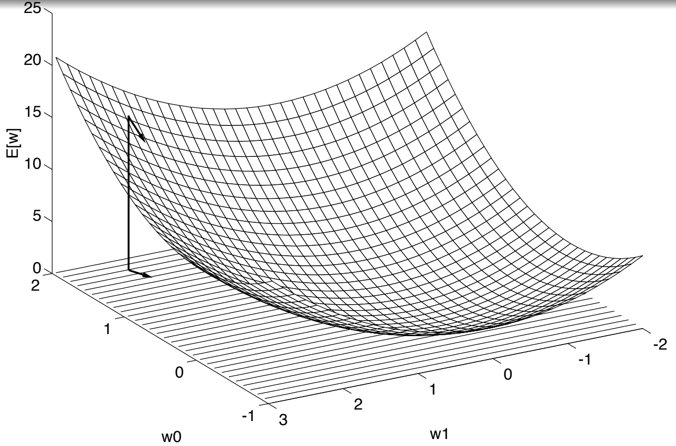
- For each example, want to compromise between *correctiveness* and *conservativeness*
  - Correctiveness: Tendency to improve on $\mathbf{x}^t$ (reduce error)
  - Conservativeness: Tendency to keep $\mathbf{w}^{t+1}$ close to $\mathbf{w}^t$ (minimize distance)
- Use *cost function* that measures both:

$$U(\mathbf{w}) = dist\left(\mathbf{w}^{t+1}, \mathbf{w}^t\right) + \eta\, error\left(r^t, \overbrace{\mathbf{w}^{t+1} \cdot \mathbf{x}^t}^{\text{curr ex, new wts}}\right)$$

Gradient Descent



Gradient Descent (cont'd)



$$\frac{\partial U}{\partial \mathbf{w}} = \left[\frac{\partial U}{\partial w_0}, \frac{\partial U}{\partial w_1}, \cdots, \frac{\partial U}{\partial w_n} \right]$$

Gradient Descent (cont'd)

$$
\begin{aligned}
U(\mathbf{w}) &= \overbrace{\|\mathbf{w}^{t+1} - \mathbf{w}^{t}\|_2^2}^{conserv.} + \overbrace{\eta}^{coef.} \overbrace{(r^t - \mathbf{w}^{t+1} \cdot \mathbf{x}^t)^2}^{corrective} \\
&= \sum_{j=1}^{n} \left(w_j^{t+1} - w_j^t \right)^2 + \eta \left(r^t - \sum_{j=1}^{n} w_j^{t+1} x_j^t \right)^2
\end{aligned}
$$

Take gradient w.r.t. $\mathbf{w}^{t+1}$ and set to $\mathbf{0}$:

$$
0 = 2\left(w_i^{t+1} - w_i^t \right) - 2\eta \left(r^t - \sum_{j=1}^{n} w_j^{t+1} x_j^t \right) x_i^t
$$

Gradient Descent (cont'd)

Approximate with

$$0 = 2\left(w_i^{t+1} - w_i^t\right) - 2\eta\left(r^t - \sum_{j=1}^{n} w_j^t x_j^t\right) x_i^t \ ,$$

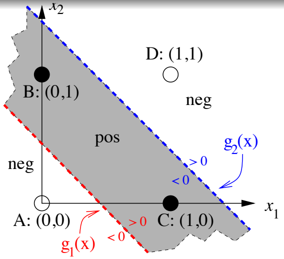
which yields

$$w_i^{t+1} = w_i^t + \overbrace{\eta\left(r^t - y^t\right) x_i^t}^{\Delta w_i^t}$$

Implementation Approaches

- Can use rules on previous slides on an example-by-example basis, sometimes called *incremental*, *stochastic*, or *on-line* GD
  - Has a tendency to “jump around” more in searching, which helps avoid getting trapped in local minima
- Alternatively, can use *standard* or *batch* GD, in which the classifier is evaluated over all training examples, summing the error, and then updates are made
  - I.e., sum up Δw_i for all examples, but don’t update w_i until summation complete
  - This is an inherent averaging process and tends to give better estimate of the gradient

Handling Nonlinearly Separable Problems
The XOR Problem



Represent with *intersection* of two linear separators

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

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

pos $= \left\{\mathbf{x} \in \mathbb{R}^2 : g_1(\mathbf{x}) > 0 \text{ AND } g_2(\mathbf{x}) < 0\right\}$

neg $= \left\{\mathbf{x} \in \mathbb{R}^2 : g_1(\mathbf{x}), g_2(\mathbf{x}) < 0 \text{ OR } g_1(\mathbf{x}), g_2(\mathbf{x}) > 0\right\}$

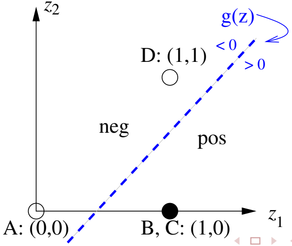
Handling Nonlinearly Separable Problems

The XOR Problem (cont'd)

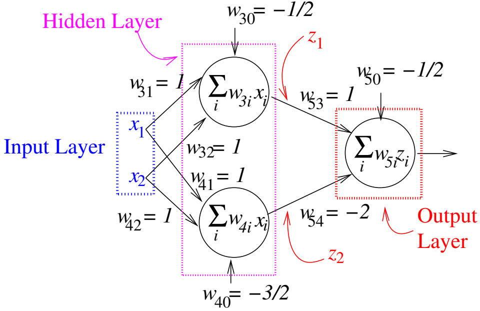
Let $z_i = \begin{cases} 0 & \text{if } g_i(\mathbf{x}) < 0 \\ 1 & \text{otherwise} \end{cases}$

Class	(x_1, x_2)	$g_1(\mathbf{x})$	z_1	$g_2(\mathbf{x})$	z_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 z_1, z_2 into $g(\mathbf{z}) = 1 \cdot z_1 - 2 \cdot z_2 - 1/2$



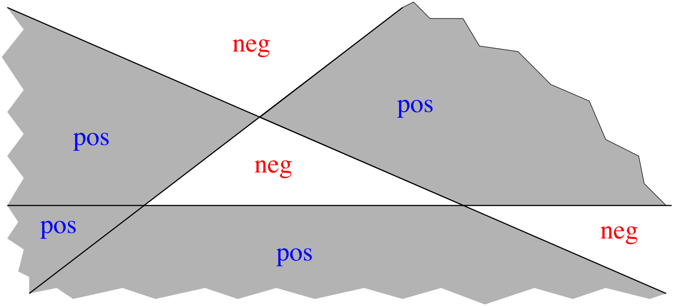
In other words, we *remapped* all vectors $\mathbf{x}$ to $\mathbf{z}$ such that the classes are linearly separable in the new vector space



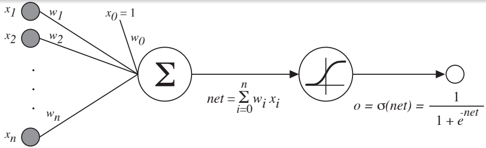
This is a *two-layer perceptron* or *two-layer feedforward neural network*

Each neuron outputs 1 if its weighted sum exceeds its threshold, 0 otherwise

By adding up to 2 *hidden layers* of perceptrons, can represent any *union* of *intersection of halfspaces*



The Sigmoid Unit



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

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

Squashes net into $[0, 1]$ range

Nice property:

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

Continuous, differentiable approximation to threshold

Sigmoid Unit
Gradient Descent

Again, use squared error for correctiveness:

$$E(\mathbf{w}^t) = \frac{1}{2}\left(r^t - y^t\right)^2$$

(folding $1/2$ of correctiveness into error func)

Thus $\displaystyle \frac{\partial E}{\partial w_j^t} = \frac{\partial}{\partial w_j^t}\frac{1}{2}\left(r^t - y^t\right)^2$

$$= \frac{1}{2}\,2\left(r^t - y^t\right)\frac{\partial}{\partial w_j^t}\left(r^t - y^t\right) = \left(r^t - y^t\right)\left(-\frac{\partial y^t}{\partial w_j^t}\right)$$

Sigmoid Unit
Gradient Descent (cont'd)

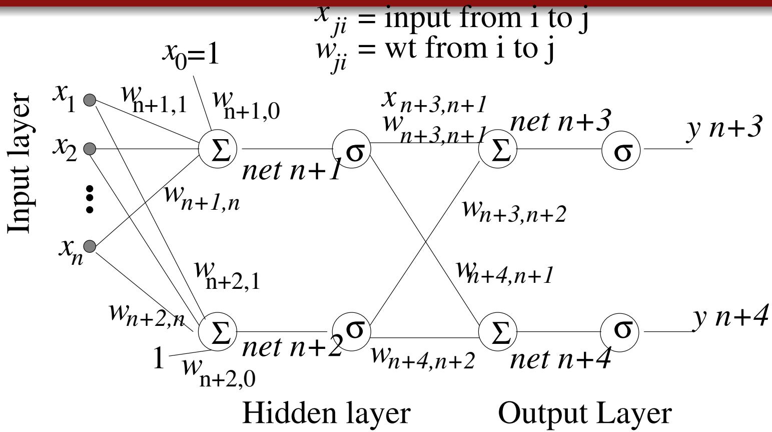
Since y^t is a function of $net^t = \mathbf{w}^t \cdot \mathbf{x}^t$,

$$
\begin{aligned}
\frac{\partial E}{\partial w_j^t} &= -\left(r^t - y^t\right) \frac{\partial y^t}{\partial net^t} \frac{\partial net^t}{\partial w_j^t} \\
&= -\left(r^t - y^t\right) \frac{\partial \sigma\left(net^t\right)}{\partial net^t} \frac{\partial net^t}{\partial w_j^t} \\
&= -\left(r^t - y^t\right) y^t \left(1 - y^t\right) x_j^t
\end{aligned}
$$

Update rule:

$$
\boxed{w_j^{t+1} = w_j^t + \eta\, y^t \left(1 - y^t\right) \left(r^t - y^t\right) x_j^t}
$$

Multilayer Networks



Use sigmoid units since continuous and differentiable

$$E^t = E(\mathbf{w}^t) = \frac{1}{2} \sum_{k \in outputs} \left(r_k^t - y_k^t\right)^2$$

Training Multilayer Networks

Output Units

Adjust weight w_{ji}^t according to E^t as before

For output units, this is easy since contribution of w_{ji}^t to E^t when j is an output unit is the same as for single neuron case[1], i.e.,

$$\frac{\partial E^t}{\partial w_{ji}^t} = -\left(r_j^t - y_j^t\right) y_j^t \left(1 - y_j^t\right) x_{ji}^t = -\delta_j^t \, x_{ji}^t$$

where $\delta_j^t = -\frac{\partial E^t}{\partial net_j^t} =$ *error term* of unit j

[1] This is because all other outputs are constants w.r.t. w_{ji}^t

- How can we compute the error term for hidden layers when there is no target output $\mathbf{r}^t$ 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"

Training Multilayer Networks

Hidden Units (cont'd)

The impact that w_{ji}^t has on E^t is only through net_j^t and units immediately "downstream" of j:

$$\frac{\partial E^t}{\partial w_{ji}^t} = \frac{\partial E^t}{\partial net_j^t}\frac{\partial net_j^t}{\partial w_{ji}^t} = x_{ji}^t \sum_{k \in down(j)} \frac{\partial E^t}{\partial net_k^t}\frac{\partial net_k^t}{\partial net_j^t}$$

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

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

Works for arbitrary number of hidden layers

Backpropagation Algorithm

Initialize all weights to small random numbers

Until termination condition satisfied do

- For each training example $(r^t, \mathbf{x}^t)$ do
  1. Input $\mathbf{x}^t$ to the network and compute the outputs $\mathbf{y}^t$
  2. For each output unit k
  $$\delta_k^t \leftarrow y_k^t\,(1 - y_k^t)\,(r_k^t - y_k^t)$$
  3. For each hidden unit h
  $$\delta_h^t \leftarrow y_h^t\,(1 - y_h^t) \sum_{k \in down(h)} w_{k,h}^t\,\delta_k^t$$
  4. Update each network weight $w_{j,i}^t$
  $$w_{j,i}^t \leftarrow w_{j,i}^t + \Delta w_{j,i}^t$$
  where
  $$\Delta w_{j,i}^t = \eta\,\delta_j^t\,x_{j,i}^t$$

Backpropagation Algorithm

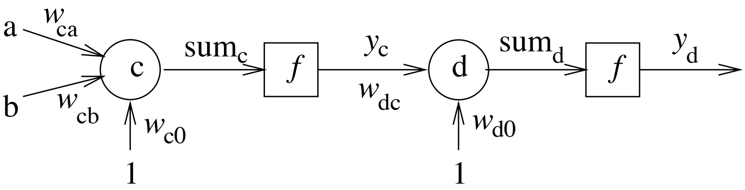
Example

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))

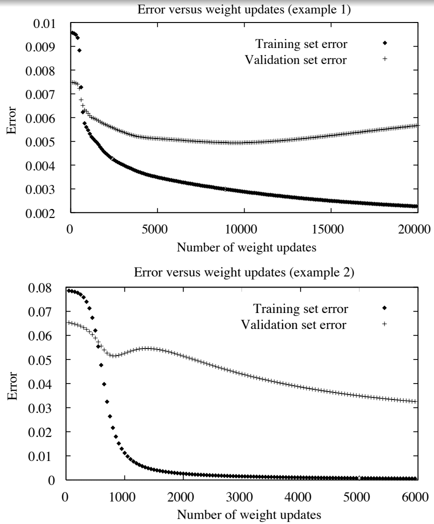
delta_c(t) = y_c(t) * (1 - y_c(t)) * delta_d(t) * w_dc(t)

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)

- 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 η: Small values slow convergence, large values might overshoot minimum, can adapt it over time

Backpropagation Algorithm
Overfitting



Danger of stopping too soon!

Backpropagation Algorithm

Remarks

- Alternative error function: *cross entropy*

$$E^t = \sum_{k \in outputs} \left(r_k^t \ln y_k^t + \left(1 - r_k^t\right) \ln\left(1 - y_k^t\right)\right)$$

  "blows up" if $r_k^t \approx 1$ and $y_k^t \approx 0$ or vice-versa (vs. squared error, which is always in $[0, 1]$)

- *Regularization:* penalize large weights to make space more linear and reduce risk of overfitting:

$$E^t = \frac{1}{2} \sum_{k \in outputs} \left(r_k^t - y_k^t\right)^2 + \gamma \sum_{i,j} (w_{ji}^t)^2$$

Backpropagation Algorithm

Remarks (cont'd)

Representational power:

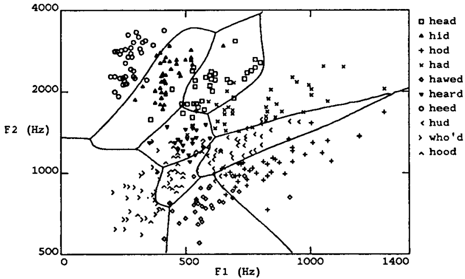
- Any boolean function can be represented with 2 layers
- Any bounded, continuous function can be represented with arbitrarily small error with 2 layers
- Any function can be represented with arbitrarily small error with 3 layers

Number of required units may be large

May not be able to find the right weights

Hypothesis Space

- Hyp. space $\mathcal{H}$ is set of all weight vectors (continuous vs. discrete of decision trees)
- Search via Backprop: Possible because error function and output functions are continuous & differentiable
- Inductive bias: (Roughly) smooth interpolation between data points

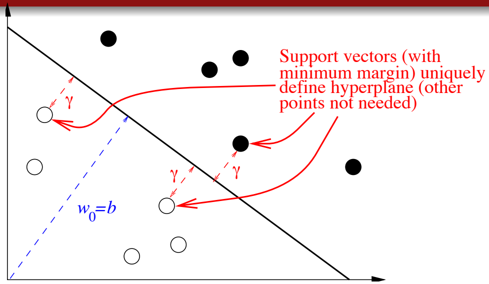


Similar to ANNs, polynomial classifiers, and RBF networks in that it remaps inputs and then finds a hyperplane

- Main difference is how it works

Features of SVMs:

- Maximization of *margin*
- *Duality*
- Use of *kernels*
- Use of problem *convexity* to find classifier (often without local minima)



- A hyperplane's *margin* γ 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 $1/\gamma^2$ (under appropriate assumptions)
- Definition assumes linear separability (more general definitions exist that do not)

Support Vector Machines

The Perceptron Algorithm Revisited

$\mathbf{w}_0 \leftarrow \mathbf{0}$, $b_0 \leftarrow 0$, $m \leftarrow 0$, $r^t \in \{-1, +1\} \; \forall t$

While mistakes are made on training set

- For $t = 1$ to N ($=$ # training vectors)
  - If $r^t \left(\mathbf{w}_m \cdot \mathbf{x}^t + b_m \right) \leq 0$
    - $\mathbf{w}_{m+1} \leftarrow \mathbf{w}_m + \eta \, r^t \, \mathbf{x}^t$
    - $b_{m+1} \leftarrow b_m + \eta \, r^t$
    - $m \leftarrow m + 1$

Final predictor: $h(\mathbf{x}) = \operatorname{sgn}\left(\mathbf{w}_m \cdot \mathbf{x} + b_m \right)$

Support Vector Machines

The Perceptron Algorithm Revisited (partial example)

t	x_1^t	x_2^t	r^t	w_1	w_2	b	α	x_1^t	x_2^t	r^t	w_1	w_2	b	α
1	4	1	$+1$	0.4	0.1	0.1	1	4	1	$+1$	0.4	0.0	0.0	2
2	5	3	$+1$	0.4	0.1	0.1	0	5	3	$+1$	0.4	0.0	0.0	0
3	6	3	$+1$	0.4	0.1	0.1	0	6	3	$+1$	0.4	0.0	0.0	0
4	2	1	-1	0.4	0.1	0.1	0	2	1	-1	0.2	-0.1	-0.1	3
5	2	2	-1	0.4	0.1	0.1	0	2	2	-1	0.2	-0.1	-0.1	0
6	3	1	-1	0.4	0.1	0.1	0	3	1	-1	0.2	-0.1	-0.1	0
1	4	1	$+1$	0.4	0.1	0.1	1	4	1	$+1$	0.2	-0.1	-0.1	2
2	5	3	$+1$	0.4	0.1	0.1	0	5	3	$+1$	0.2	-0.1	-0.1	0
3	6	3	$+1$	0.4	0.1	0.1	0	6	3	$+1$	0.2	-0.1	-0.1	0
4	2	1	-1	0.2	0.0	0.0	1	2	1	-1	0.0	-0.2	-0.2	4
5	2	2	-1	0.2	0.0	0.0	0	2	2	-1	0.0	-0.2	-0.2	0
6	3	1	-1	0.2	0.0	0.0	0	3	1	-1	0.0	-0.2	-0.2	0
1	4	1	$+1$	0.2	0.0	0.0	1	4	1	$+1$	0.4	-0.1	-0.1	3
2	5	3	$+1$	0.2	0.0	0.0	0	5	3	$+1$	0.4	-0.1	-0.1	0
3	6	3	$+1$	0.2	0.0	0.0	0	6	3	$+1$	0.4	-0.1	-0.1	0
4	2	1	-1	0.0	-0.1	-0.1	2	2	1	-1	0.4	-0.1	-0.1	4
5	2	2	-1	0.0	-0.1	-0.1	0	2	2	-1	0.4	-0.1	-0.1	0
6	3	1	-1	0.0	-0.1	-0.1	0	3	1	-1	0.4	-0.1	-0.1	0
1	4	1	$+1$	0.4	0.0	0.0	2	4	1	$+1$	0.4	-0.1	-0.1	3
2	5	3	$+1$	0.4	0.0	0.0	0	5	3	$+1$	0.4	-0.1	-0.1	0
3	6	3	$+1$	0.4	0.0	0.0	0	6	3	$+1$	0.4	-0.1	-0.1	0
4	2	1	-1	0.4	0.0	0.0	2	2	1	-1	0.2	-0.2	-0.2	5
5	2	2	-1	0.4	0.0	0.0	0	2	2	-1	0.2	-0.2	-0.2	0
6	3	1	-1	0.4	0.0	0.0	0	3	1	-1	0.2	-0.2	-0.2	0

At this point, $\mathbf{w} = (0.2, -0.2)$, $b = -0.2$, $\alpha = (3, 0, 0, 5, 0, 0)$

Can compute

$w_1 = \eta(\alpha_1 r^1 x_1^1 + \alpha_4 r^4 x_1^4) = 0.1(3(1)4 + 5(-1)2) = 0.2$

$w_2 = \eta(\alpha_1 r^1 x_2^1 + \alpha_4 r^4 x_2^4) = 0.1(3(1)1 + 5(-1)1) = -0.2$

I.e., $\mathbf{w} = \eta \sum_{t=1}^{N} \alpha_t r^t \mathbf{x}^t$

Another way of representing predictor:

$$h(\mathbf{x}) = \mathrm{sgn}\left(\mathbf{w} \cdot \mathbf{x} + b\right) = \mathrm{sgn}\left(\eta \sum_{t=1}^{N} \left(\alpha_t \, r^t \, \mathbf{x}^t\right) \cdot \mathbf{x} + b\right)$$

$$= \mathrm{sgn}\left(\eta \sum_{t=1}^{N} \alpha_t \, r^t \, \left(\mathbf{x}^t \cdot \mathbf{x}\right) + b\right)$$

($\alpha_t =$ # prediction mistakes on $\mathbf{x}^t$)

So perceptron alg has equivalent *dual* form:

$\alpha \leftarrow \mathbf{0}$, $b \leftarrow 0$

While mistakes are made in For loop

- For $t = 1$ to N ($=$ # training vectors)
  - If $r^t \left(\eta \sum_{j=1}^{N} \alpha_j \, r^j \, \left(\mathbf{x}^j \cdot \mathbf{x}^t \right) + b \right) \leq 0$

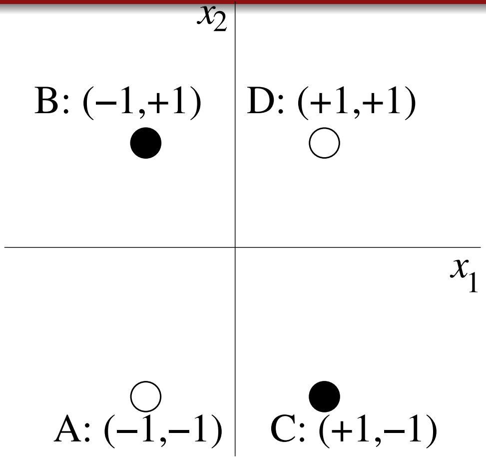
$$\alpha_t \leftarrow \alpha_t + 1$$

$$b \leftarrow b + \eta \, r^t$$

Replace weight vector with data in dot products

So what?

XOR Revisited

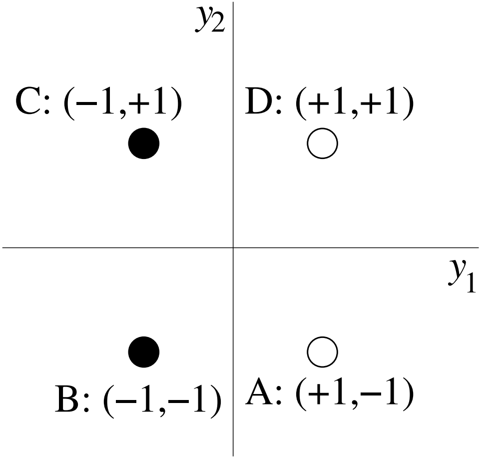


Remap to new space:

$$\phi(x_1, x_2) = \left[x_1^2, x_2^2, \sqrt{2}\,x_1x_2, \sqrt{2}\,x_1, \sqrt{2}\,x_2, 1\right]$$

XOR Revisited (cont'd)

Now consider the third and fourth dimensions of the remapped vector (scaling $\sqrt{2}$ to 1):



XOR Revisited (cont'd)

- Can easily compute the dot product $\phi(\mathbf{x}) \cdot \phi(\mathbf{z})$ (where $\mathbf{x} = [x_1, x_2]$) without first computing ϕ:

$$
\begin{aligned}
K(\mathbf{x}, \mathbf{z}) &= (\mathbf{x} \cdot \mathbf{z} + 1)^2 = (x_1 z_1 + x_2 z_2 + 1)^2 \\
&= (x_1 z_1)^2 + (x_2 z_2)^2 + 2x_1 z_1 x_2 z_2 + 2x_1 z_1 + 2x_2 z_2 + 1 \\
&= \underbrace{\left[x_1^2, x_2^2, \sqrt{2}\, x_1\, x_2, \sqrt{2}\, x_1, \sqrt{2}\, x_2, 1\right]}_{\phi(\mathbf{x})} \\
&\quad \cdot \underbrace{\left[z_1^2, z_2^2, \sqrt{2}\, z_1\, z_2, \sqrt{2}\, z_1, \sqrt{2}\, z_2, 1\right]}_{\phi(\mathbf{z})}
\end{aligned}
$$

- I.e., since we use dot products in new Perceptron algorithm, we can *implicitly* work in the remapped y space via k

Kernels

- A *kernel* is a function K such that $\forall\, \mathbf{x}, \mathbf{z}$, $K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{z})$
- E.g., previous slide (quadratic kernel)
- In general, for degree-q *polynomial kernel*, computing $(\mathbf{x} \cdot \mathbf{z} + 1)^q$ takes ℓ multiplications + 1 exponentiation for $\mathbf{x}, \mathbf{z} \in \mathbb{R}^\ell$
- In contrast, need over $\binom{\ell+q-1}{q} \geq \left(\frac{\ell+q-1}{q}\right)^q$ multiplications if compute ϕ first

Kernels (cont'd)

- 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:

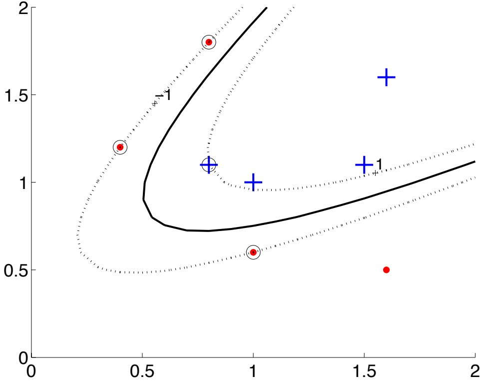
$$\{\phi_i(x)\}_{i=0}^{\infty} = [1, \sin(x), \cos(x), \sin(2x), \cos(2x), \ldots]$$

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

Types of Kernels

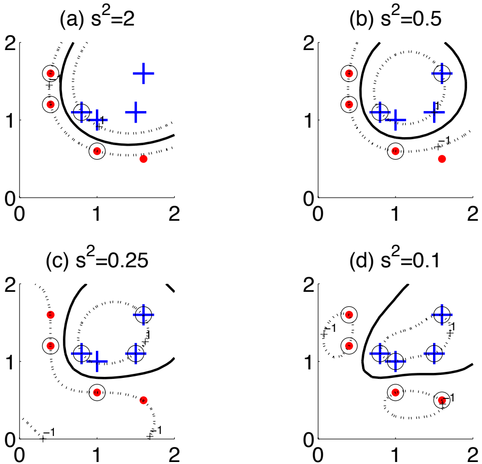
Polynomial

$$K(\mathbf{x}^t, \mathbf{x}) = \left(\mathbf{x}^t \cdot \mathbf{x} + 1\right)^q$$



Types of Kernels
Gaussian

$$K(\mathbf{x}^t, \mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}^t - \mathbf{x}\|^2}{2s^2}\right)$$



Hyperbolic tangent:

$$K(\mathbf{x}^t, \mathbf{x}) = \tanh\left(2\mathbf{x}^t \cdot \mathbf{x} + 1\right)$$

(not a true kernel)

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

In addition, the sum of two kernels is a kernel, the product of two kernels is a kernel

Finally, note that a kernel is a *similarity measure*, useful in clustering, nearest neighbor, etc.

Can show that if data linearly separable in remapped space, then get maximum margin classifier by minimizing $\mathbf{w} \cdot \mathbf{w}$ subject to $r^t\left(\mathbf{w} \cdot \mathbf{x}^t + b\right) \geq 1$

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

$$\begin{array}{ll} \underset{\boldsymbol{\alpha}}{\text{maximize}} & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \, \alpha_j \, r^i \, r^j \, K(\mathbf{x}^i, \mathbf{x}^j) \\ \text{s.t.} & \alpha_i \geq 0, i = 1, \ldots, m \\ & \sum_{i=1}^{N} \alpha_i \, r^i = 0 \end{array}$$

After optimization, label new vectors with decision function:

$$f(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^{N} \alpha_i \, r^t \, K(\mathbf{x}, \mathbf{x}^t) + b\right)$$

(Note only need to use $\mathbf{x}^t$ such that $\alpha_t > 0$, i.e., *support vectors*)

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

Support Vector Machines

Finding a Hyperplane (cont'd)

If kernel doesn't separate, can *soften* the margin with *slack variables* ξ^i:

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\xi}} \quad & \|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi^i \\ \text{s.t.} \quad & r^i((\mathbf{x}^i \cdot \mathbf{w}) + b) \geq 1 - \xi^i,\ \ i = 1, \ldots, N \\ & \xi^i \geq 0,\ \ i = 1, \ldots, N \end{aligned}$$

The dual is similar to that for hard margin:

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_{i=1}^{N} \alpha_i - \sum_{i,j} \alpha_i\, \alpha_j\, r^i\, r^j\, K(\mathbf{x}^i, \mathbf{x}^j) \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C,\ \ i = 1, \ldots, N \\ & \sum_{i=1}^{N} \alpha_i\, r^i = 0 \end{aligned}$$

Can still solve optimally

Support Vector Machines

Finding a Hyperplane (cont'd)

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)

When done, can throw out non-SVs