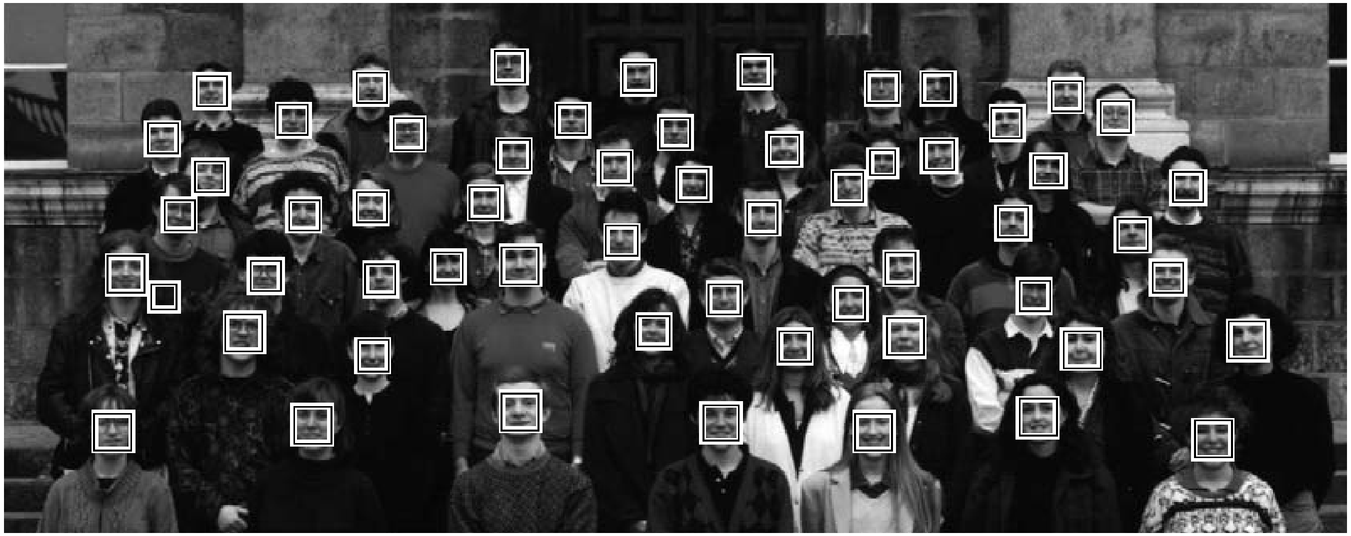
CSEP 573

Machine Learning I: Supervised Learning



(Rowley, Baluja & Kanade, 1998)

What's on our menu today?

Supervised Learning

- Classification
  - Decision trees
  - Cross validation
  - K-nearest neighbor
  - Neural networks
    - Perceptrons
  - Support Vector Machines (SVMs)
- Regression
  - Backpropagation networks

Why Learning?

Learning is essential for unknown environments

- e.g., when designer lacks omniscience

Learning is necessary in dynamic environments

- Agent can adapt to changes in environment not foreseen at design time

Learning is useful as a system construction method

- Expose the agent to reality rather than trying to approximate it through equations etc.

Learning modifies the agent's decision mechanisms to improve performance

Types of Learning

Supervised learning: correct answers for each input is provided

- E.g., decision trees, backpropagation neural networks

Unsupervised learning: correct answers not given, must discover patterns in input data

- E.g., clustering, principal component analysis

Reinforcement learning: occasional rewards (or punishments) given to guide behavior

Inductive learning

We will focus on one form of supervised learning called Inductive Learning:

Learn a function from examples

f is the target function. Examples are pairs (x, $f(x)$)

Problem: learn a function ("hypothesis") h

such that $h \approx f$ (h approximates f as best as possible)

given a training set of examples

(This is a highly simplified model of real learning:

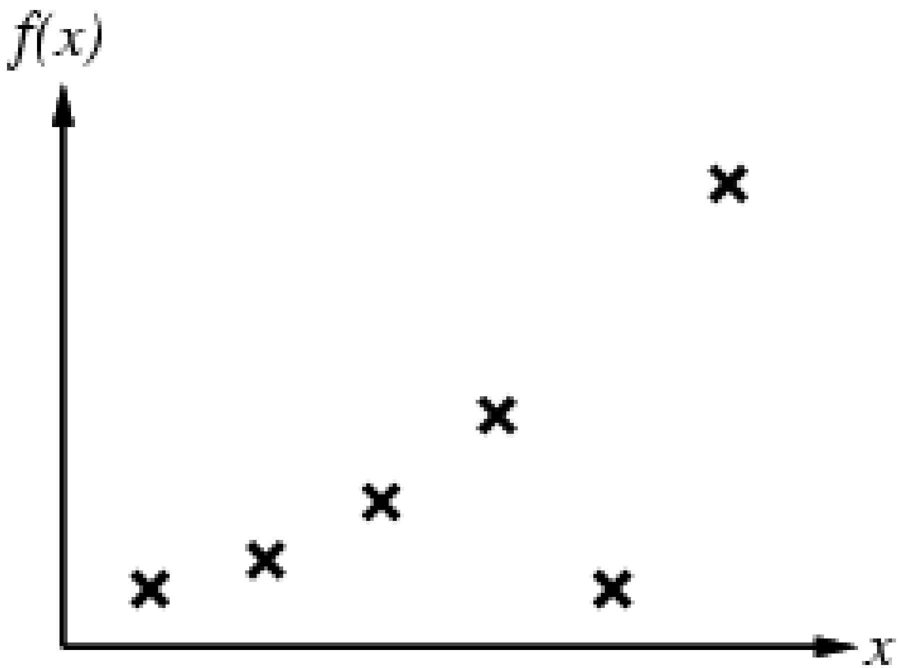
- Ignores prior knowledge
- Assumes examples are given)

Inductive learning example

Construct *h* to agree with *f* on training set

- *h* is consistent if it agrees with *f* on all training examples

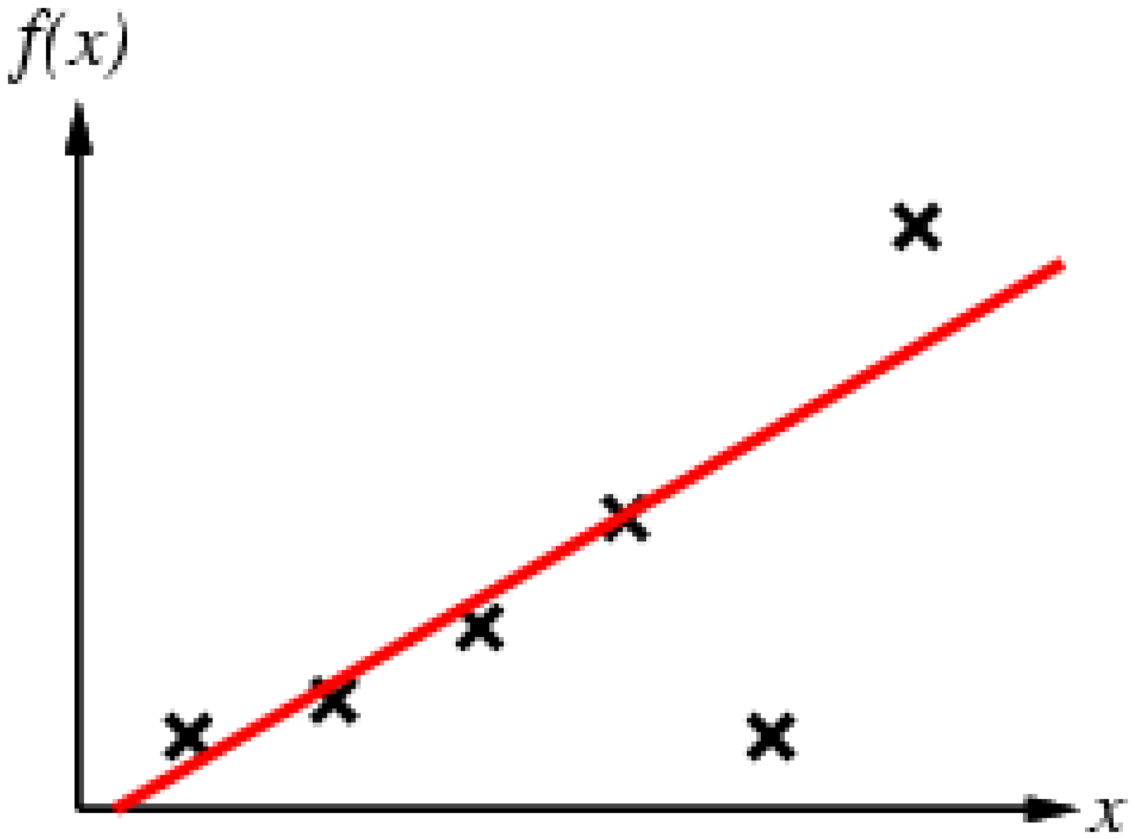
E.g., curve fitting (regression):



x = Input data point (training example)

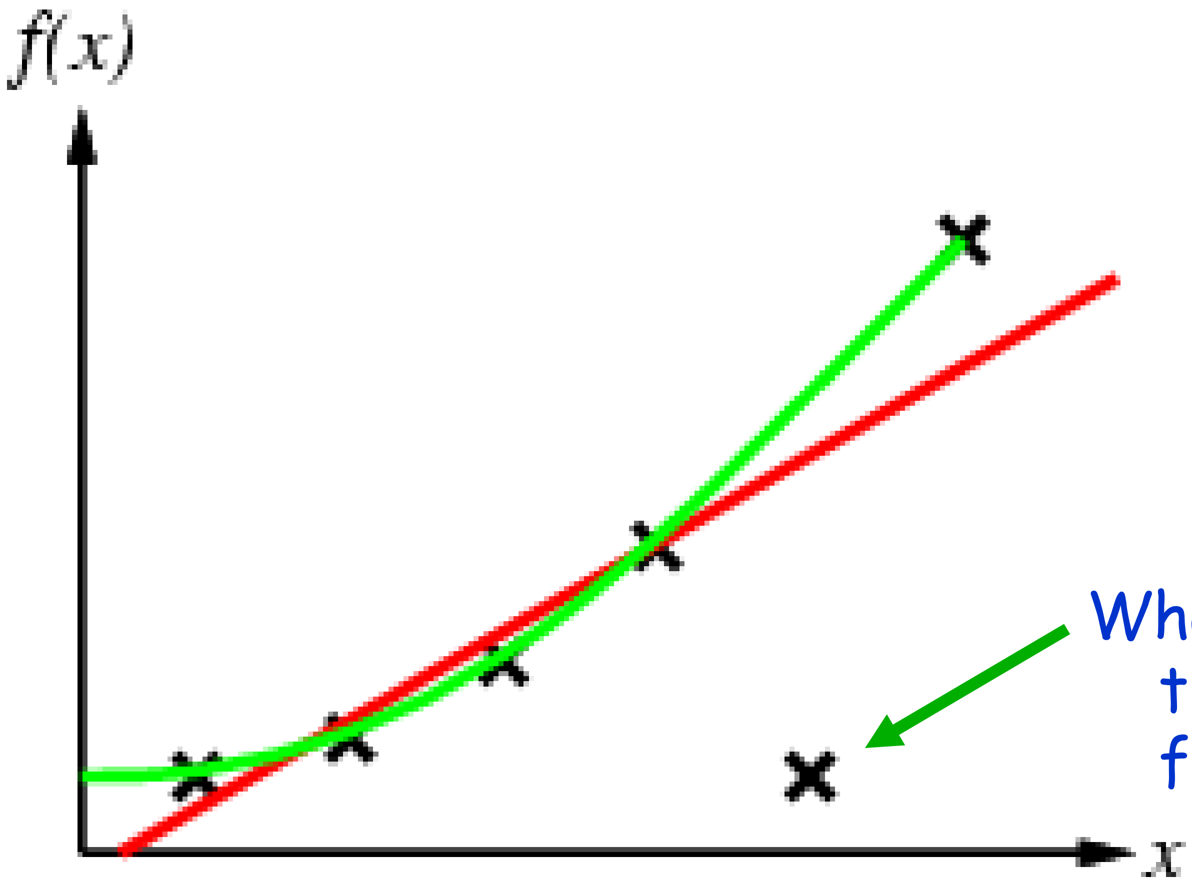
Inductive learning example

h = Straight line?



Inductive learning example

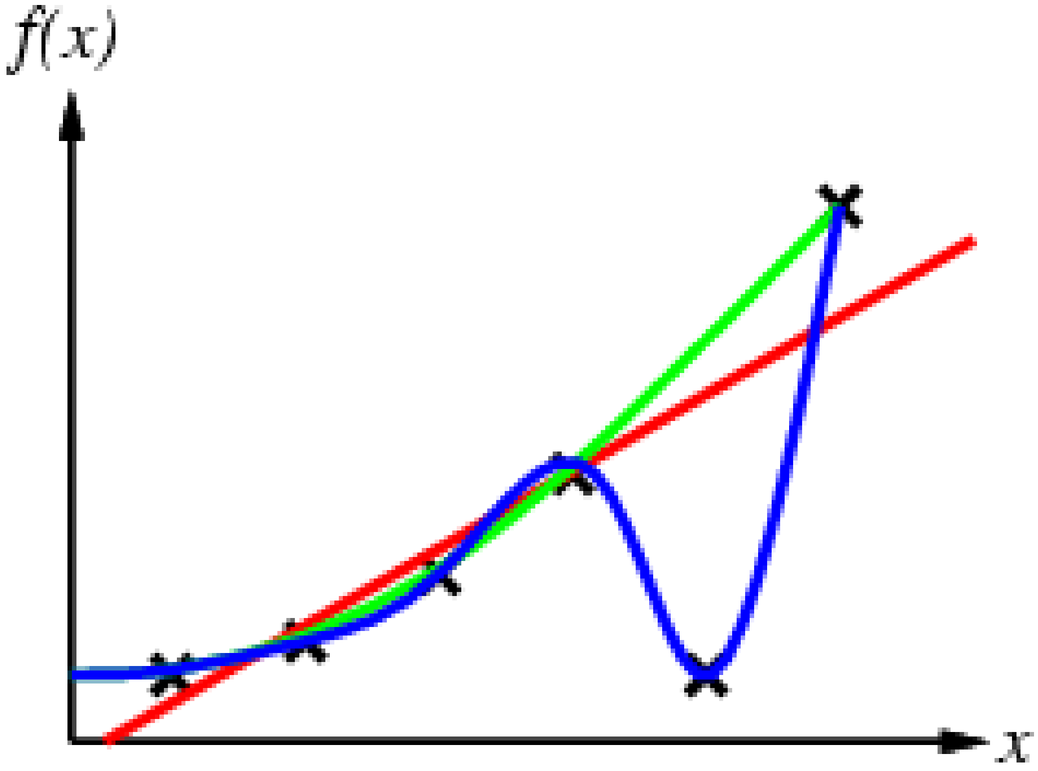
What about a quadratic function?



What about this little fella?

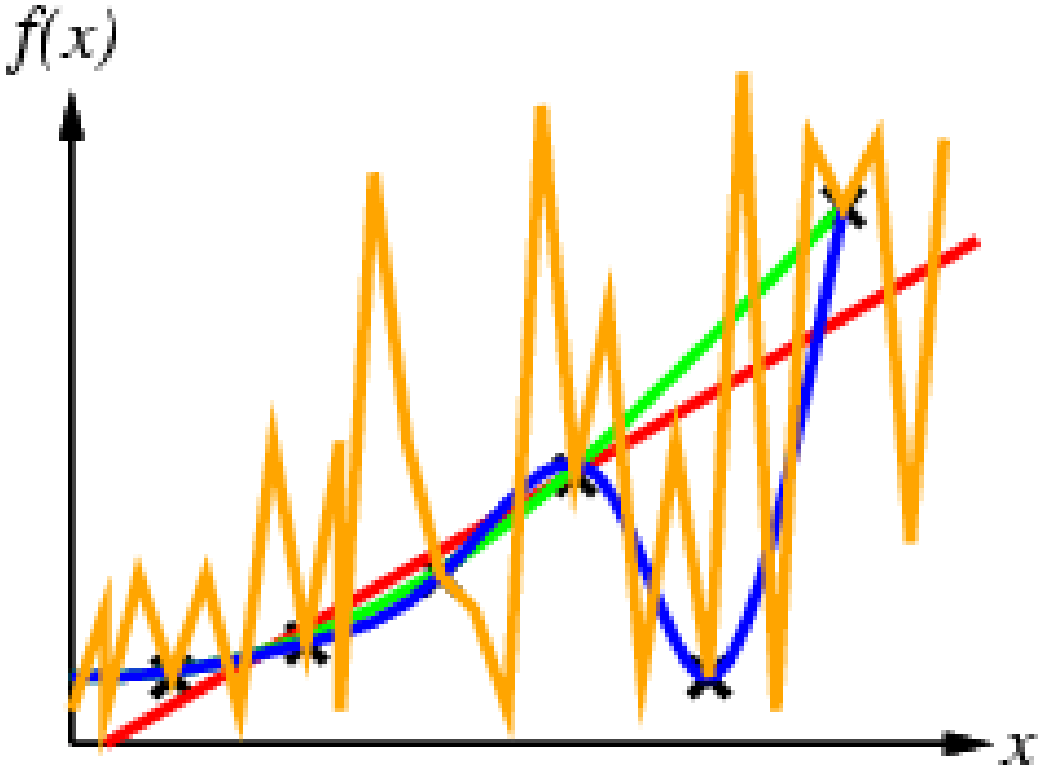
Inductive learning example

Finally, a function that satisfies all!

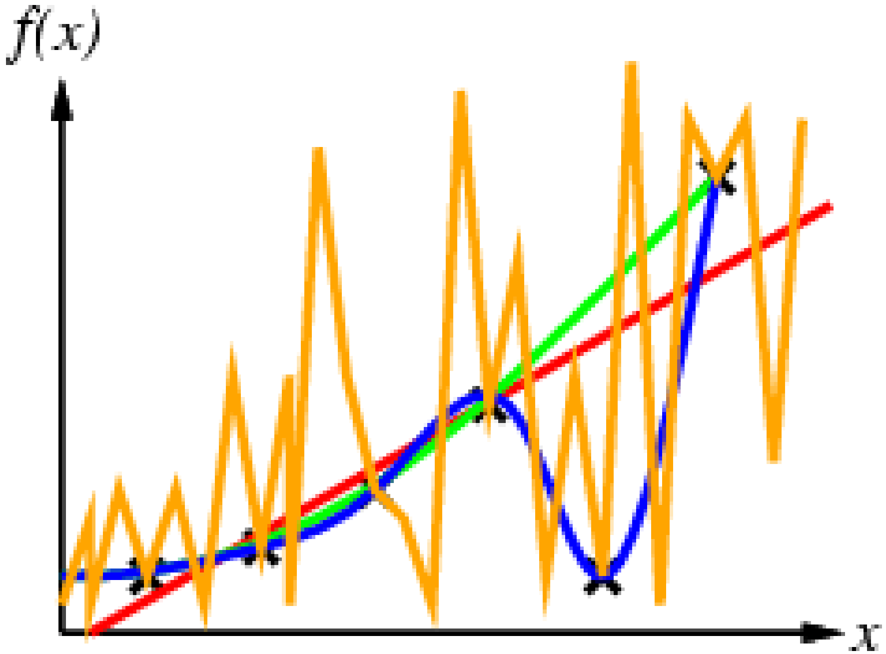


Inductive learning example

But so does this one…



Ockham's Razor Principle



Ockham's razor: prefer the simplest hypothesis consistent with data

- Related to KISS principle ("keep it simple stupid")
- Smooth blue function preferable over wiggly yellow one
- If noise known to exist in this data, even linear might be better (the lowest x might be due to noise)

Supervised Learning Technique I: Decision Trees

http://www.sfgate.com/blogs/images/sfgate/sgreen/2007/09/05/2240773250x321.jpg

Example data for learning the concept "Good day for tennis"

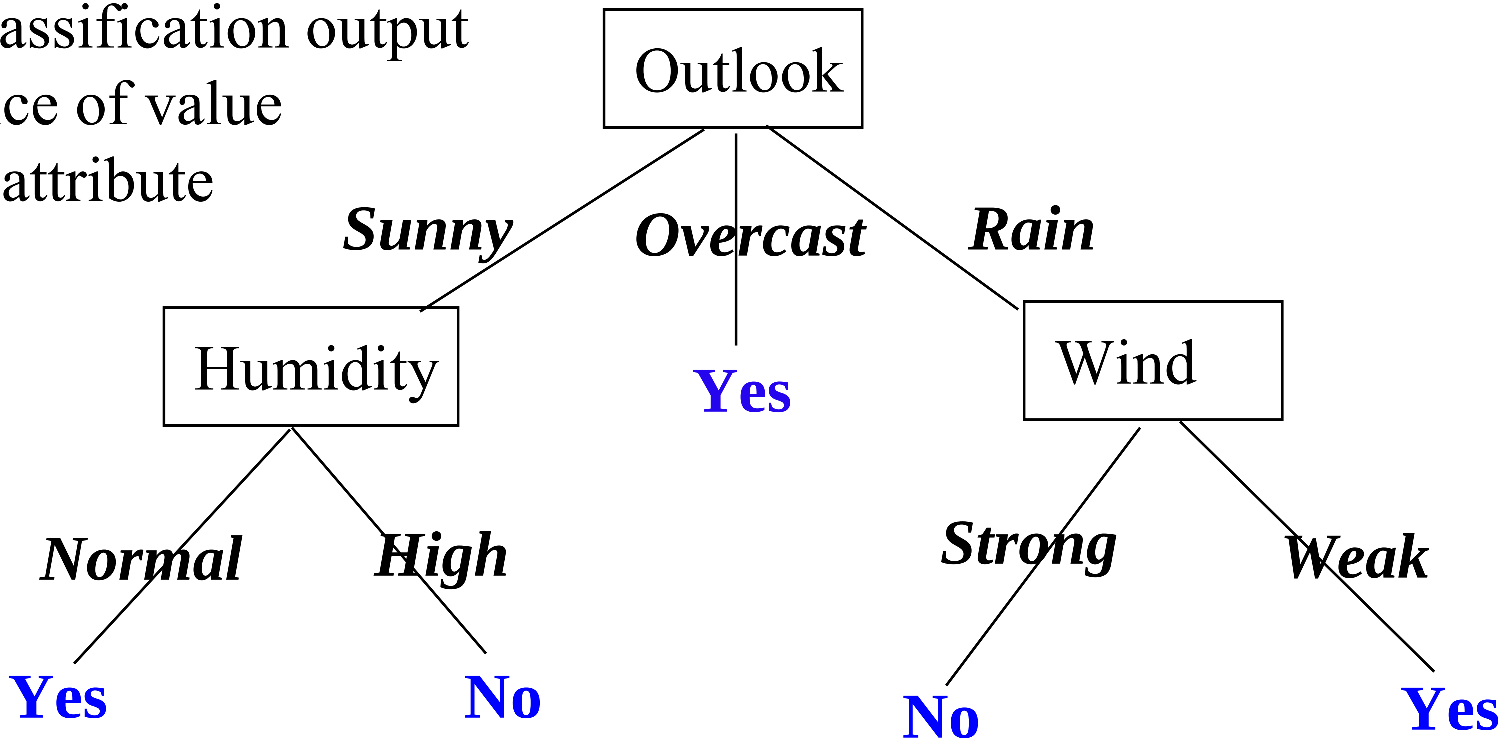
Day	Outlook	Humid	Wind	**PlayTennis?**
d1	s	h	w	n
d2	s	h	s	n
d3	o	h	w	y
d4	r	h	w	y
d5	r	n	w	y
d6	r	n	s	y
d7	o	n	s	y
d8	s	h	w	n
d9	s	n	w	y
d10	r	n	w	y
d11	s	n	s	y
d12	o	h	s	y
d13	o	n	w	y
d14	r	h	s	n

- Outlook = sunny, overcast, rain
- Humidity = high, normal
- Wind = weak, strong

A Decision Tree for the Same Data

Decision Tree for "PlayTennis?"

Leaves = classification output
Arcs = choice of value
for parent attribute



Decision tree is equivalent to logic in disjunctive normal form

PlayTennis $\Leftrightarrow$ (Sunny $\wedge$ Normal) $\vee$ Overcast $\vee$ (Rain $\wedge$ Weak)

Decision Trees

Input: Description of an object or a situation through a set of **attributes**

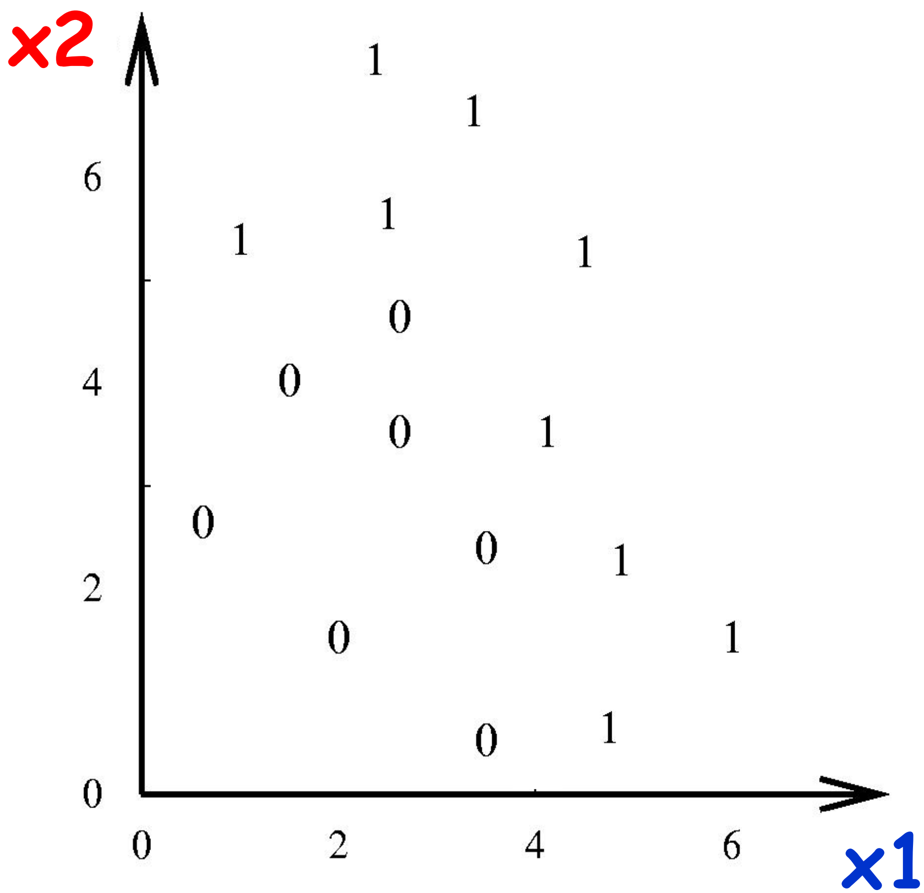
Output: a **decision** that is the predicted output value for the input

Both **input and output can be discrete or continuous**

Discrete-valued functions lead to **classification problems**

Example: Decision Tree for Continuous Valued Features and Discrete Output

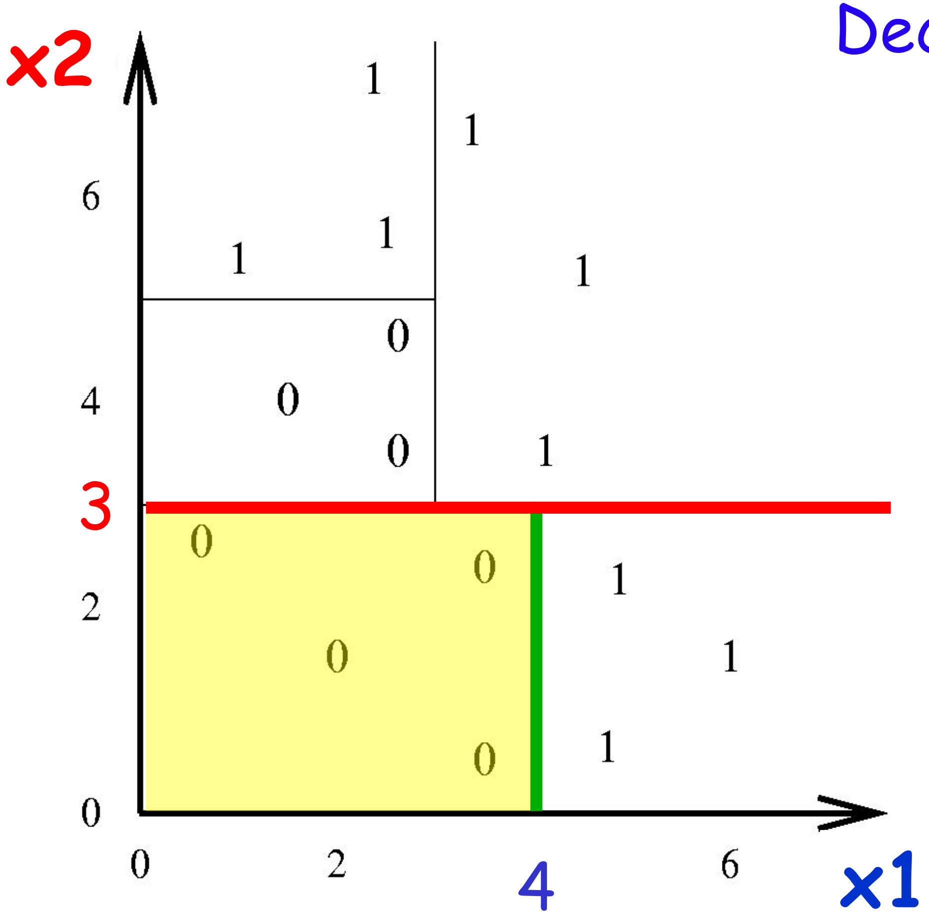
Input real number attributes (x1,x2), Classification output: 0 or 1



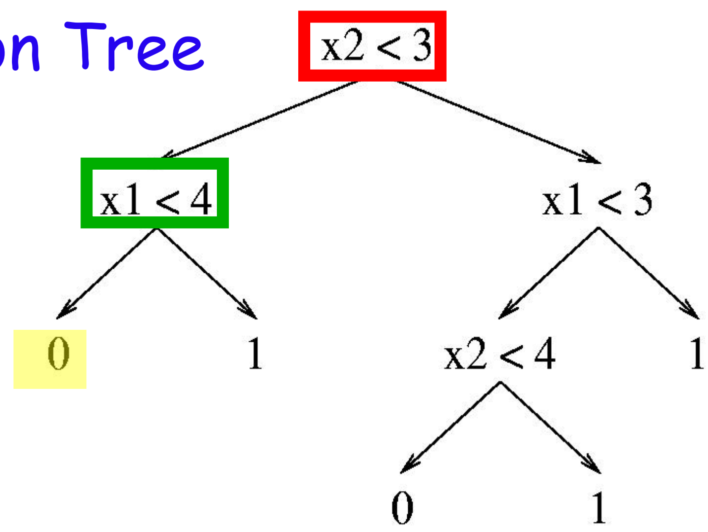
How do we branch using attribute values x1 and x2 to partition the space correctly?

Example: Classification of Continuous Valued Inputs

Decision trees divide the feature space into axis-parallel rectangles, and label each rectangle with one of the K classes.



Decision Tree

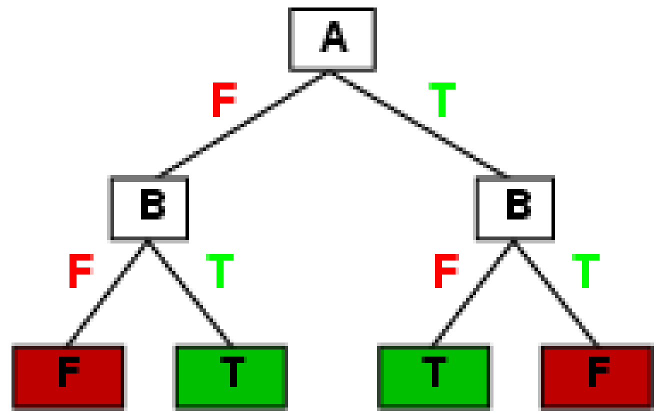


Expressiveness of Decision Trees

Decision trees can express any function of the input attributes.

E.g., for Boolean functions, truth table row = path to leaf:

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



Trivially, there is a consistent decision tree for any training set with one path to leaf for each example

- But most likely won't generalize to new examples

Prefer to find more compact decision trees

Learning Decision Trees

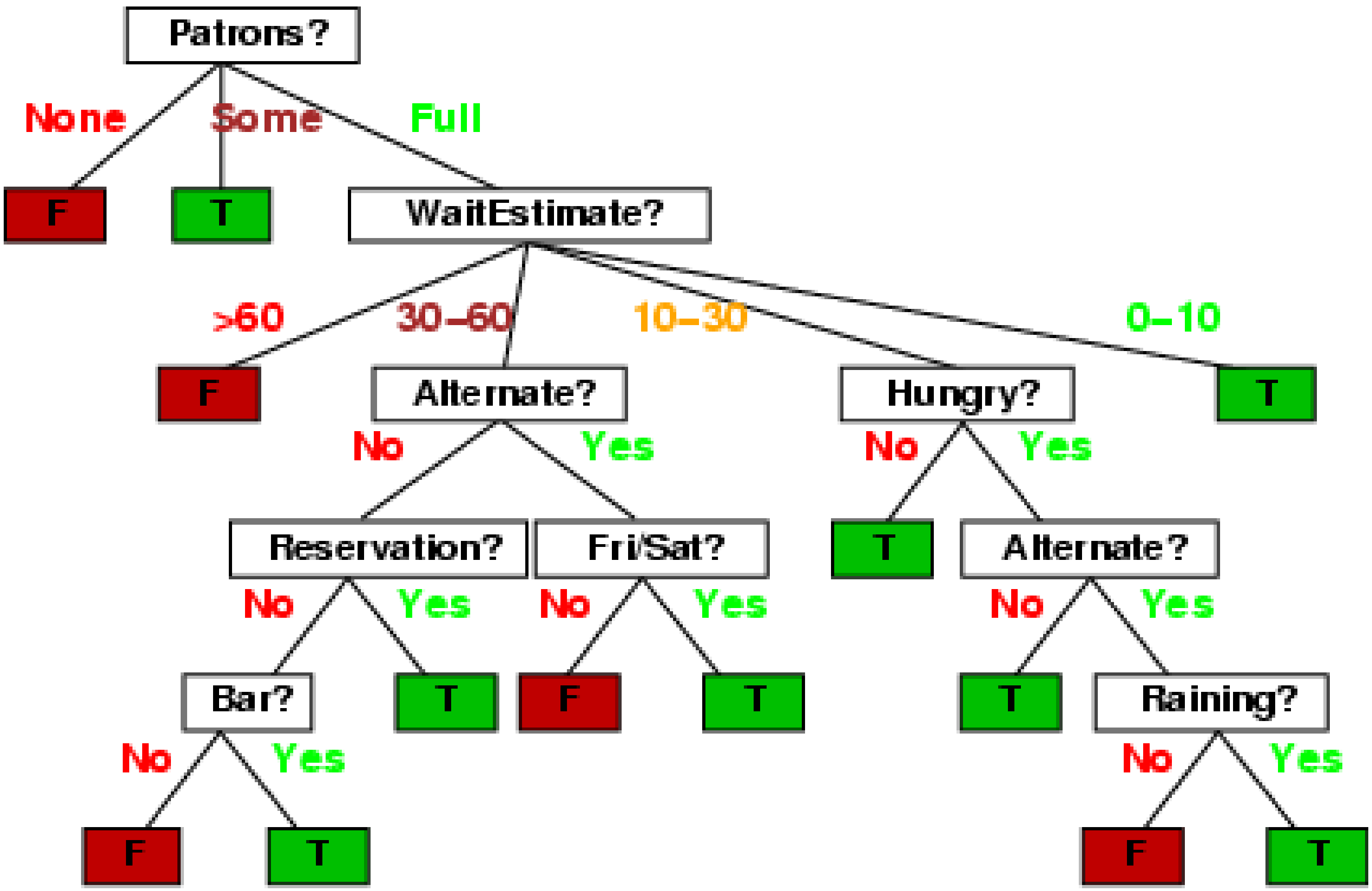
Example: When should I wait for a table at a restaurant?

Attributes (features) relevant to *Wait?* decision:

1. Alternate: is there an alternative restaurant nearby?
2. Bar: is there a comfortable bar area to wait in?
3. Fri/Sat: is today Friday or Saturday?
4. Hungry: are we hungry?
5. Patrons: number of people in the restaurant (None, Some, Full)
6. Price: price range ($, $$, $$$)
7. Raining: is it raining outside?
8. Reservation: have we made a reservation?
9. Type: kind of restaurant (French, Italian, Thai, Burger)
10. WaitEstimate: estimated waiting time (0-10, 10-30, 30-60, >60)

Example Decision tree

A decision tree for *Wait?* based on personal "rules of thumb":



Input Data for Learning

Past examples when I did/did not wait for a table:

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	\$\$\$	F	T	French	0–10	T
X_2	T	F	F	T	Full	\$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	\$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	\$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
X_6	F	T	F	T	Some	\$\$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	\$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	\$\$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	\$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	\$\$\$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	\$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	\$	F	F	Burger	30–60	T

Classification of examples is positive (T) or negative (F)

Decision Tree Learning

Aim: find a small tree consistent with training examples

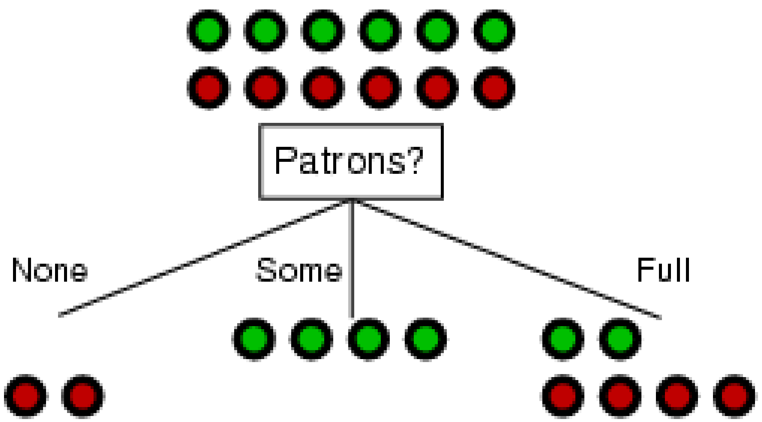
Idea: (recursively) choose "most significant" attribute as root of (sub)tree

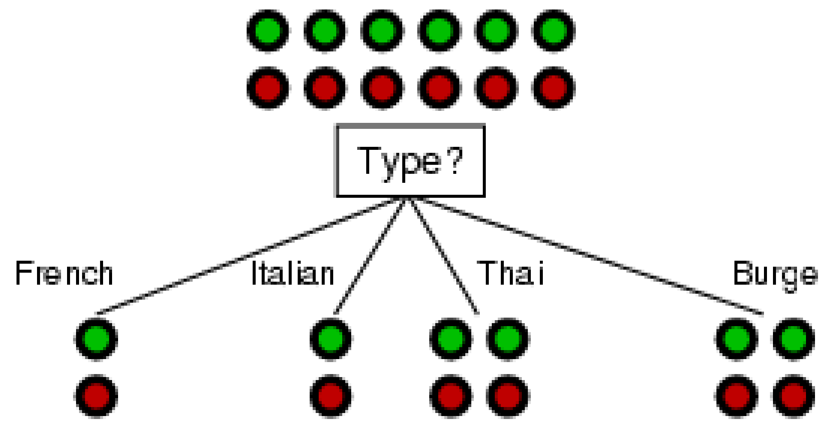
```
function DTL(examples, attributes, default) returns a decision tree
   if examples is empty then return default
   else if all examples have the same classification then return the classification
   else if attributes is empty then return MODE(examples)
   else
      best ← CHOOSE-ATTRIBUTE(attributes, examples)
      tree ← a new decision tree with root test best
      for each value v_i of best do
         examples_i ← {elements of examples with best = v_i}
         subtree ← DTL(examples_i, attributes − best, MODE(examples))
         add a branch to tree with label v_i and subtree subtree
      return tree
```

Choosing an attribute to split on

Idea: a good attribute should reduce uncertainty

- E.g., splits the examples into subsets that are (ideally) "all positive" or "all negative"





Patrons? is a better choice

For *Type?*, to wait or not to wait is still at 50%

How do we quantify uncertainty?

http://a.espncdn.com/media/ten/2006/0306/photo/g_mcenroe_195.jpg

Using information theory to quantify uncertainty

Entropy measures the amount of uncertainty in a probability distribution

Entropy (or Information Content) of an answer to a question with possible answers $v_1, \ldots, v_n$:

$$I(P(v_1), \ldots, P(v_n)) = \Sigma_{i=1} -P(v_i) \log_2 P(v_i)$$

Using information theory

Imagine we have p examples with Wait = True (positive) and n examples with Wait = false (negative).

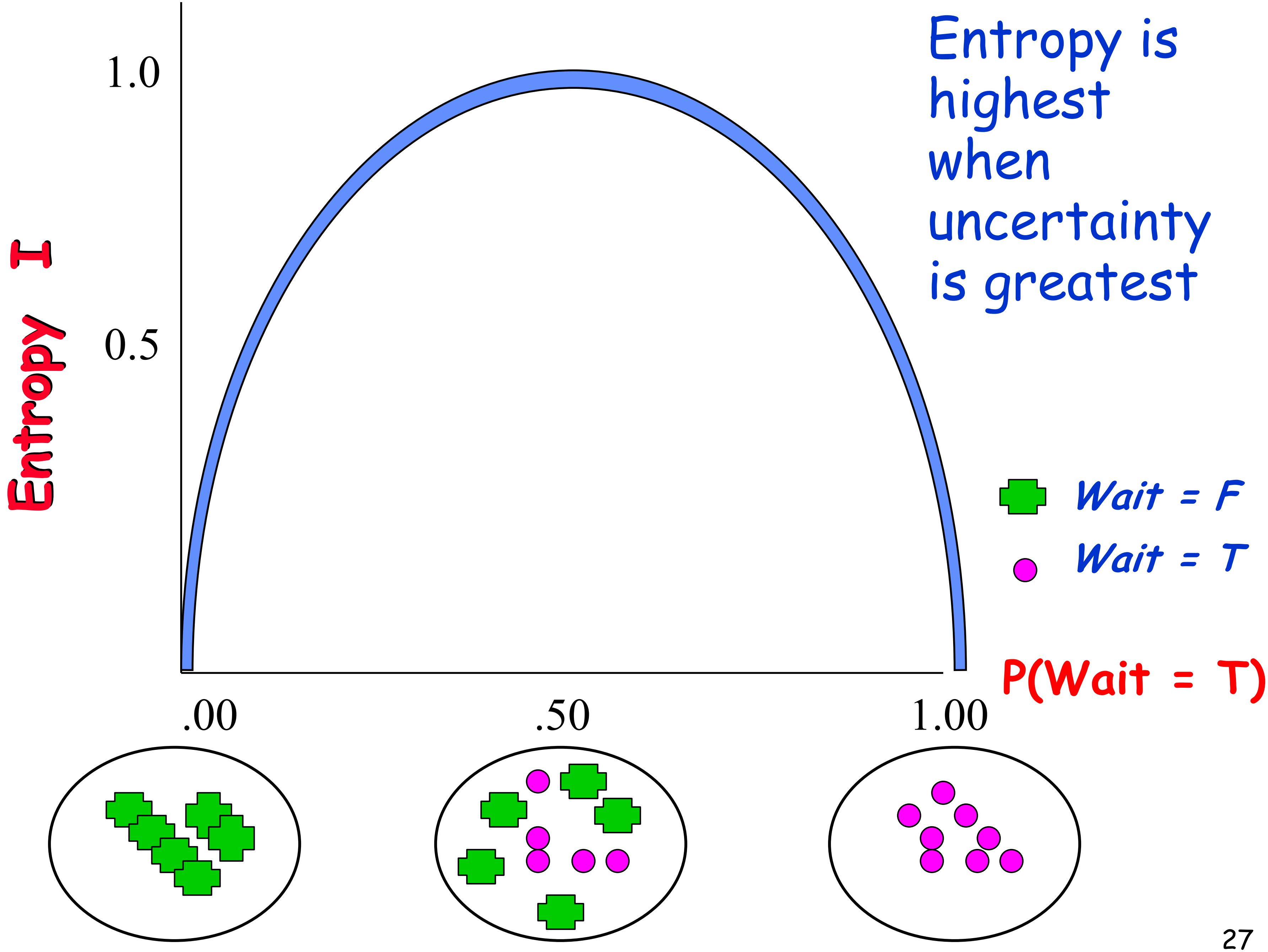
Our best estimate of the probabilities of Wait = true or false is given by:

$$P(true) \approx p / p + n$$

$$p(false) \approx n / p + n$$

Hence the entropy of Wait is given by:

$$I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n}\log_2 \frac{p}{p+n} - \frac{n}{p+n}\log_2 \frac{n}{p+n}$$


Entropy is highest when uncertainty is greatest
Entropy I
1.0
0.5
.00
.50
1.00
P(Wait = T)
Wait = F
Wait = T

Choosing an attribute to split on

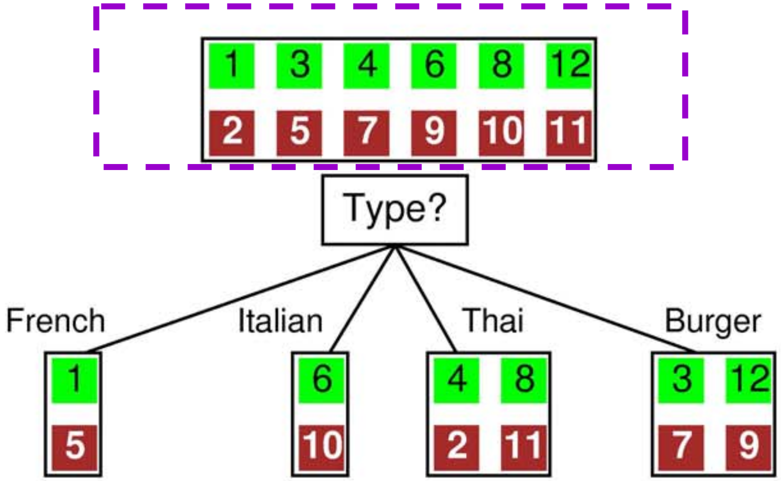
Idea: a good attribute should reduce uncertainty and result in "gain in information"

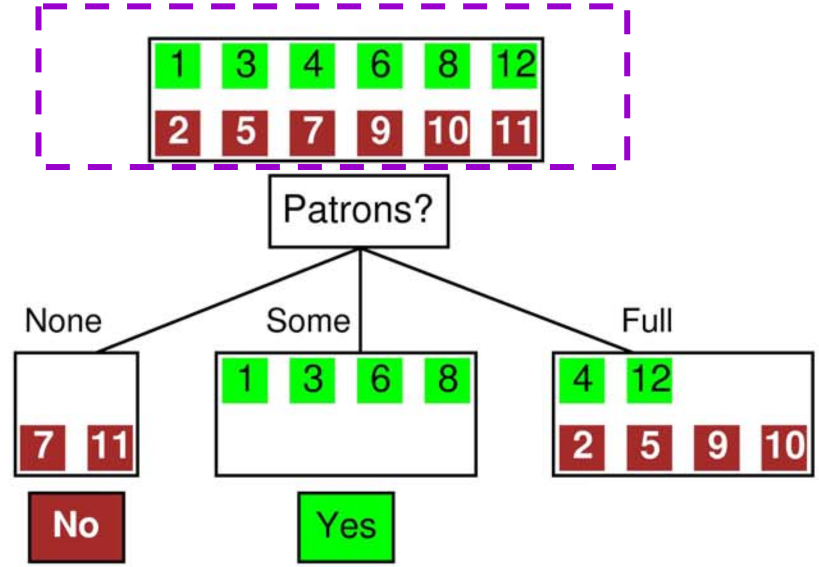
How much information do we gain if we disclose the value of some attribute?

Answer:

uncertainty before – uncertainty after

Back at the Restaurant





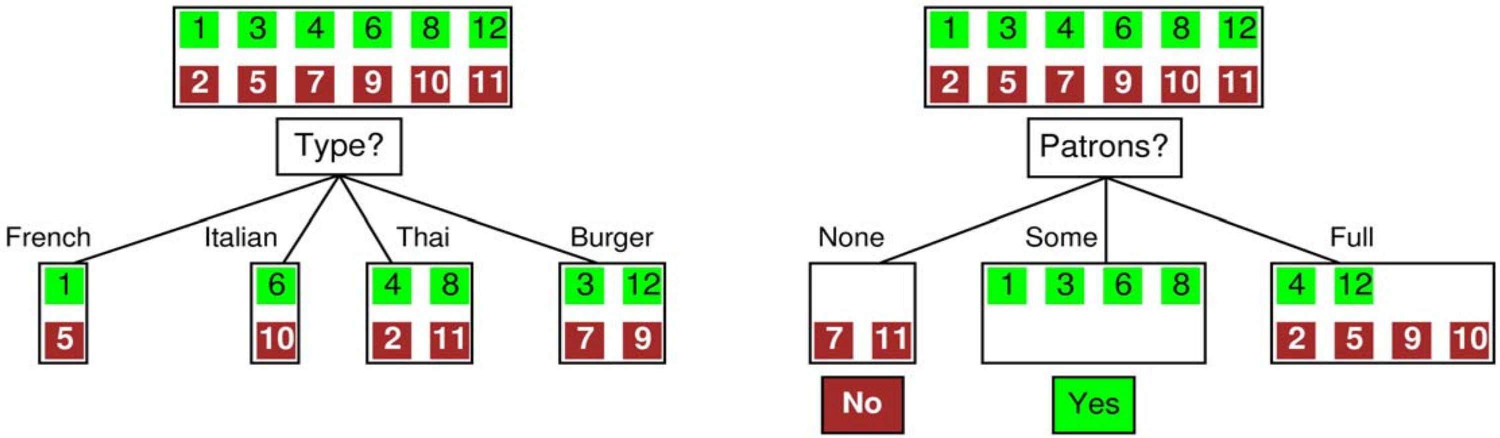
Before choosing an attribute:

Entropy = - 6/12 log(6/12) – 6/12 log(6/12)

= - log(1/2) = log(2) = 1 bit

There is “1 bit of information to be discovered”

Back at the Restaurant



If we choose Type: Go along branch "French": we have entropy = 1 bit; similarly for the others.

Information gain = 1-1 = 0 along any branch

If we choose Patrons:

In branch "None" and "Some", entropy = 0

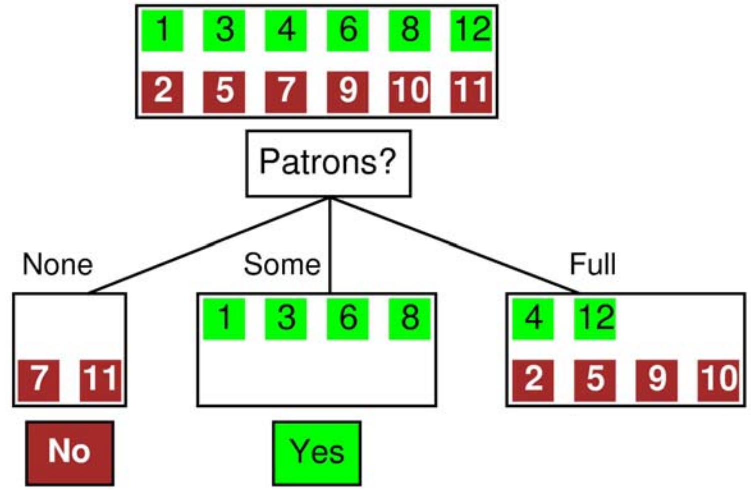
For "Full", entropy = -2/6 log(2/6)-4/6 log(4/6) = 0.92

Info gain = (1-0) or (1-0.92) bits > 0 in both cases

So choosing Patrons gains more information!

Entropy across branches

- How do we combine entropy of different branches?
- Answer: Compute average entropy
- Weight entropies according to probabilities of branches
  - 2/12 times we enter "None", so weight for "None" = 1/6
  - "Some" has weight: 4/12 = 1/3
  - "Full" has weight 6/12 = ½



$$AvgEntropy = \sum_{i=1}^{n} \frac{p_i + n_i}{p + n} Entropy\left(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}\right)$$

weight for each branch

entropy for each branch

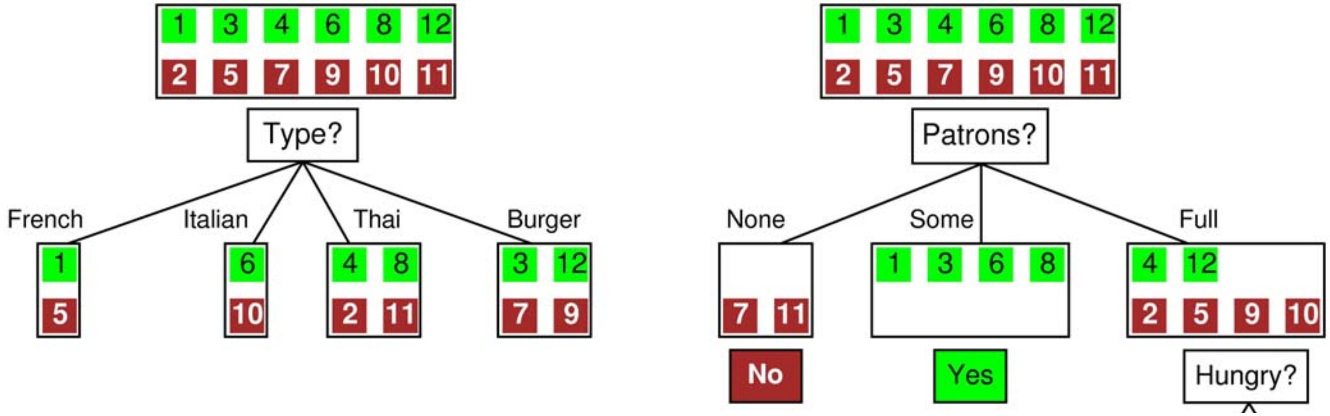
Information gain

Information Gain (IG) or reduction in entropy from using attribute A:

IG(A) = Entropy before – AvgEntropy after choosing A

Choose the attribute with the largest IG

Information gain in our example



$$IG(Patrons) = 1 - [\frac{2}{12} I(0,1) + \frac{4}{12} I(1,0) + \frac{6}{12} I(\frac{2}{6}, \frac{4}{6})] = .541 \text{ bits}$$

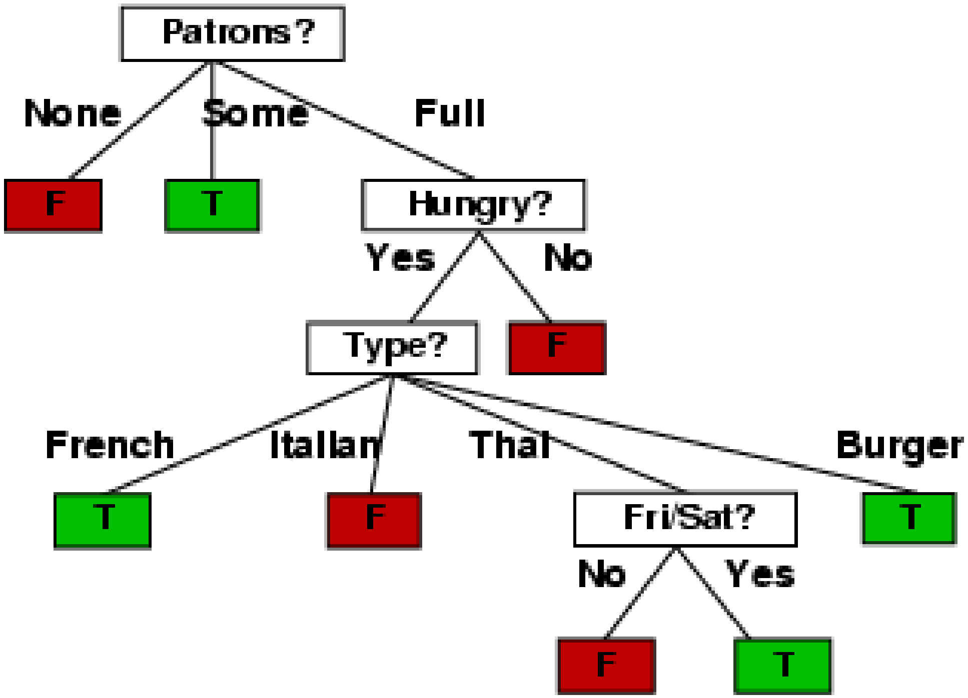
$$IG(Type) = 1 - [\frac{2}{12} I(\frac{1}{2}, \frac{1}{2}) + \frac{2}{12} I(\frac{1}{2}, \frac{1}{2}) + \frac{4}{12} I(\frac{2}{4}, \frac{2}{4}) + \frac{4}{12} I(\frac{2}{4}, \frac{2}{4})] = 0 \text{ bits}$$

Patrons has the highest IG of all attributes

$\Rightarrow$ DTL algorithm chooses *Patrons* as the root

Should I stay or should I go?
Learned Decision Tree

Decision tree learned from the 12 examples:



Substantially simpler than "rules-of-thumb" tree

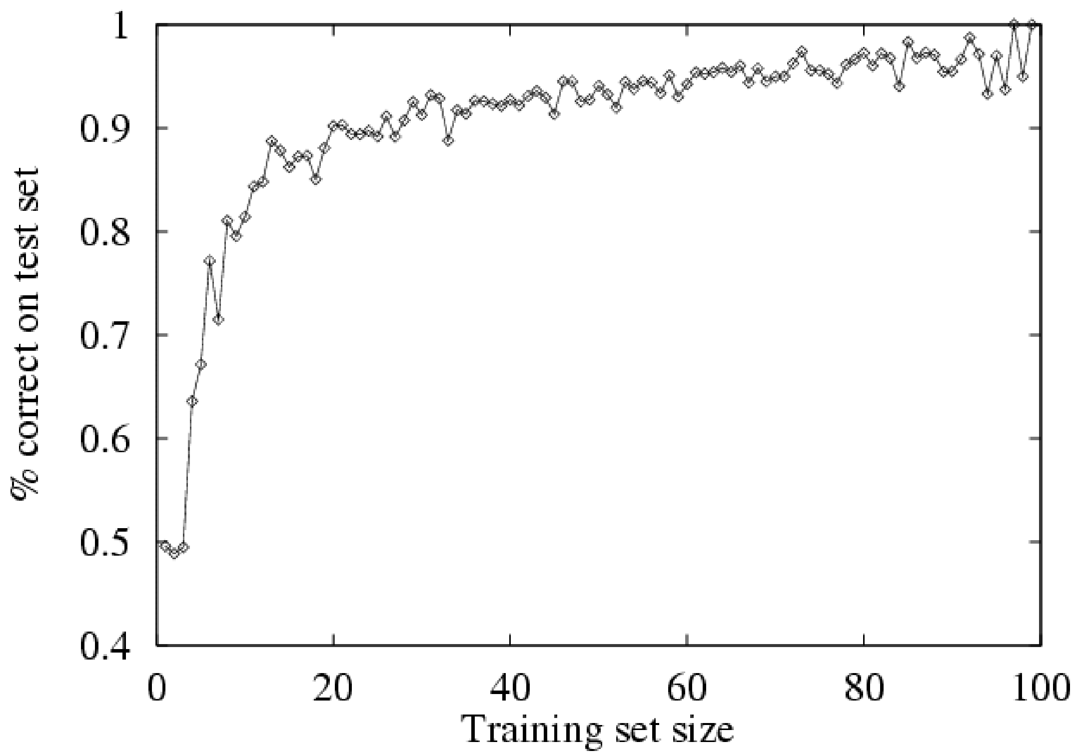
- more complex hypothesis not justified by small amount of data

Performance Evaluation

How do we know that the learned tree $h \approx f$?

Answer: Try h on a new test set of examples

Learning curve = % correct on test set as a function of training set size



Generalization

How do we know the classifier function we have learned is good?

- Look at generalization error on test data
  - Method 1: Split data in training vs test set (the "hold out" method)
  - Method 2: Cross-Validation

Cross-validation

K-fold cross-validation:

- Divide data into k subsets of equal size
- Train learning algorithm K times, leaving out one of the subsets. Compute error on left-out subset
- Report average error over all subsets

Leave-1-out cross-validation:

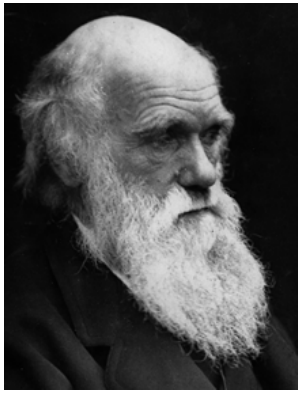
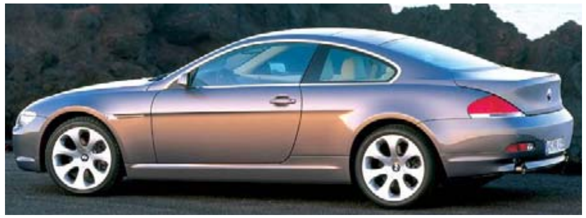
- Train on all but 1 data point, test on that data point; repeat for each point
- Report average error over all points

Decision trees are for girlie men – let’s move on to more powerful learning algorithms

http://www.ipjnet.com/schwarzenegger2/pages/arnold_01.htm

Example Problem: Face Detection

How do we build a classifier to distinguish between faces and other objects?








Images as Vectors

Binary handwritten characters

```
00000000010000000000    00000000011110000000
00000000110000000000    00000001100001100000
00000000101000000000    00000011000000110000
00000001000010000000    00001100000000011000
00000010000010000000    00001000000000001000
00000100000001000000    00001100000000010000
00001000000001000000    00000111000000100000
00001100111111110000    00000011100111100000
00001111110000010000    00000000011110000000
00011000000000011000    00000011000111000000
00010000000000001100    00001100000000110000
00110000000000000100    00011000000000011000
00110000000000000110    00110000000000001000
00100000000000000010    00100000000000001100
00100000000000000010    00010000000000011000
01100000000000000010    00011000000000010000
01000000000000000000    00001000000000110000
00000000000000000000    00000011111111000000
```

Treat an image as a high-dimensional vector
(e.g., by reading pixel values left to right, top to bottom row)

$$\mathbf{I} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_{N-2} \\ p_N \end{bmatrix}$$

Greyscale images

62	79	23	119	120	105	4	0
10	10	9	62	12	78	34	0
10	58	197	46	46	0	0	48
176	135	5	188	191	68	0	49
2	1	1	29	26	37	0	77
0	89	144	147	187	102	62	208
255	252	0	166	123	62	0	31
166	63	127	17	1	0	99	30

.....

⋮

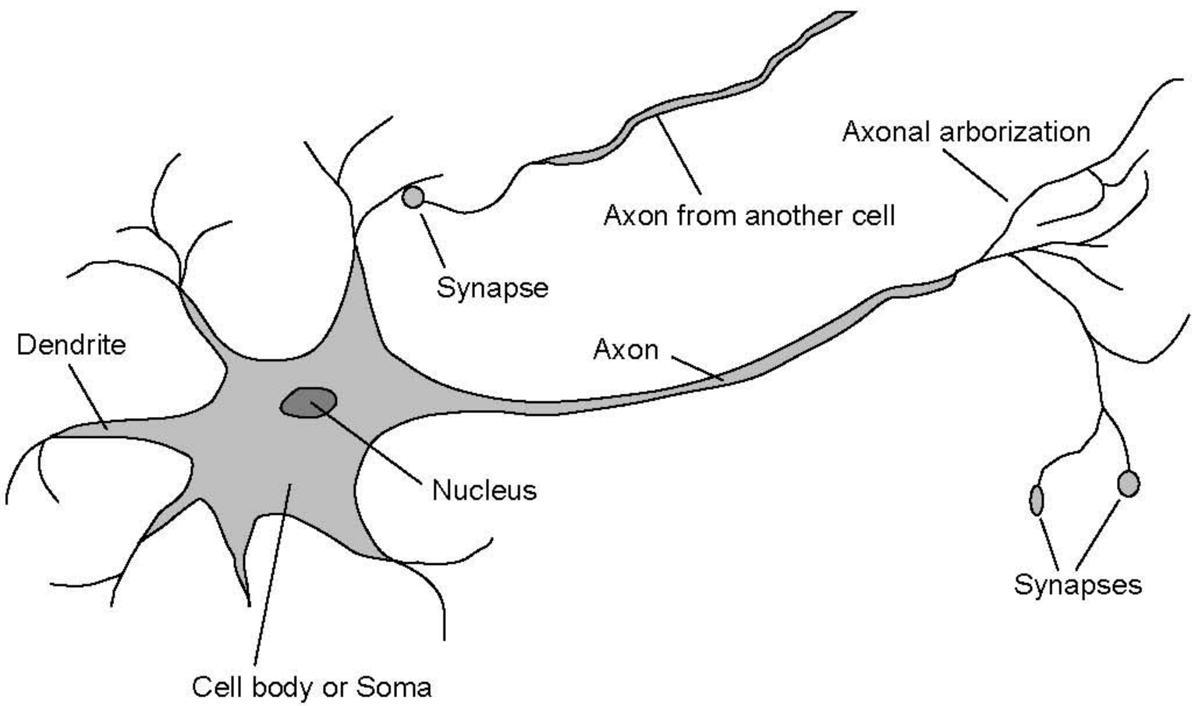
Pixel value p_i can be 0 or 1 (binary image) or 0 to 255 (greyscale)

The human brain is extremely good at classifying images

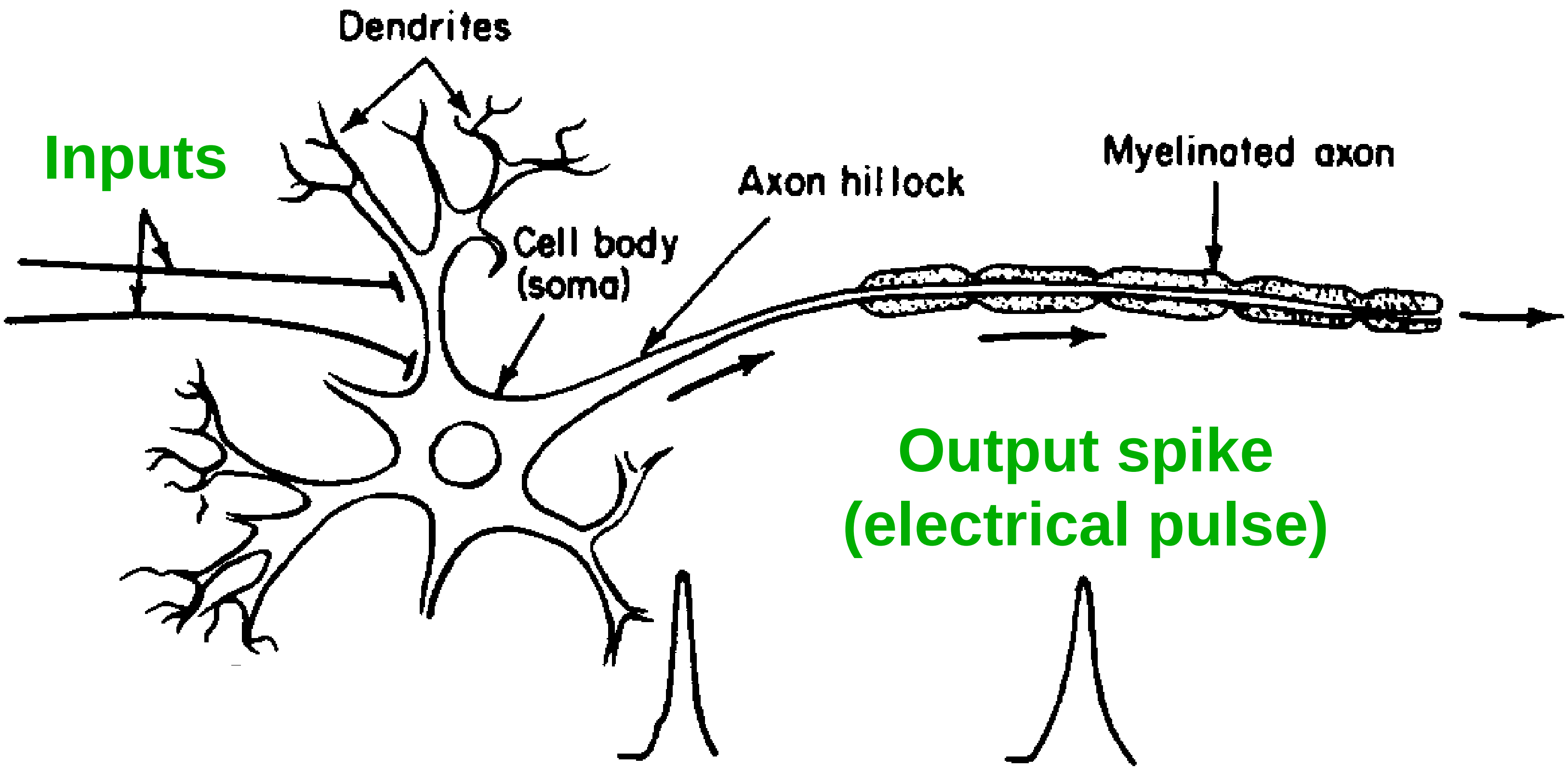
Can we develop classification methods by emulating the brain?

Brains

10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



Neurons communicate via spikes

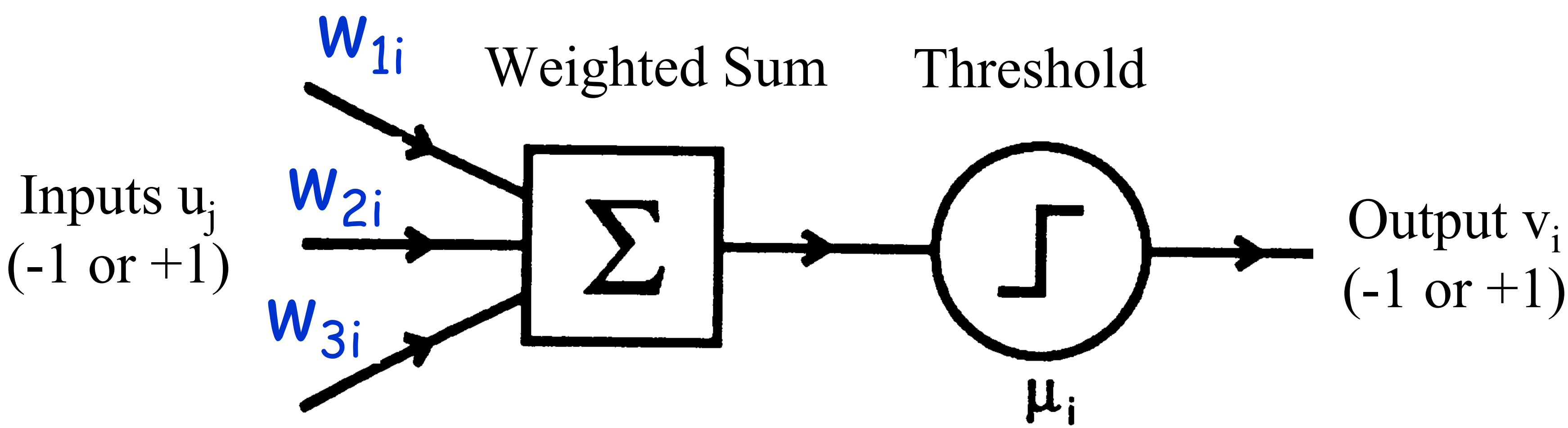


Output spike roughly dependent on whether sum of all inputs reaches a threshold

Neurons as "Threshold Units"

Artificial neuron:

- m binary inputs (-1 or 1), 1 output (-1 or 1)
- Synaptic weights w_{ji}
- Threshold μ_i



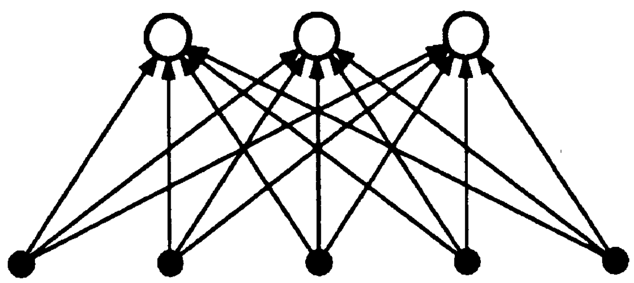
$$v_i = \Theta\left(\sum_j w_{ji} u_j - \mu_i\right)$$

$\Theta(x) = 1$ if $x > 0$ and -1 if $x \le 0$

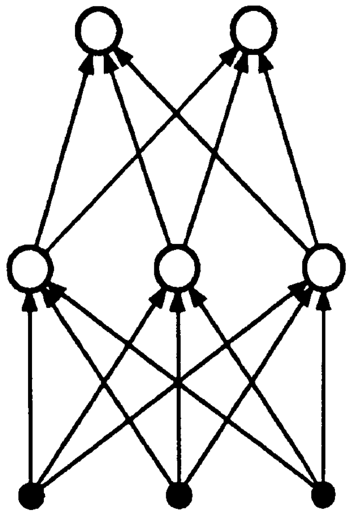
"Perceptrons" for Classification

- Fancy name for a type of layered "feed-forward" networks (no loops)
- Uses artificial neurons ("units") with binary inputs and outputs

Single-layer



Multilayer



Perceptrons and Classification

Consider a single-layer perceptron

- Weighted sum forms a *linear hyperplane*

$$\sum_j w_{ji} u_j - \mu_i = 0$$

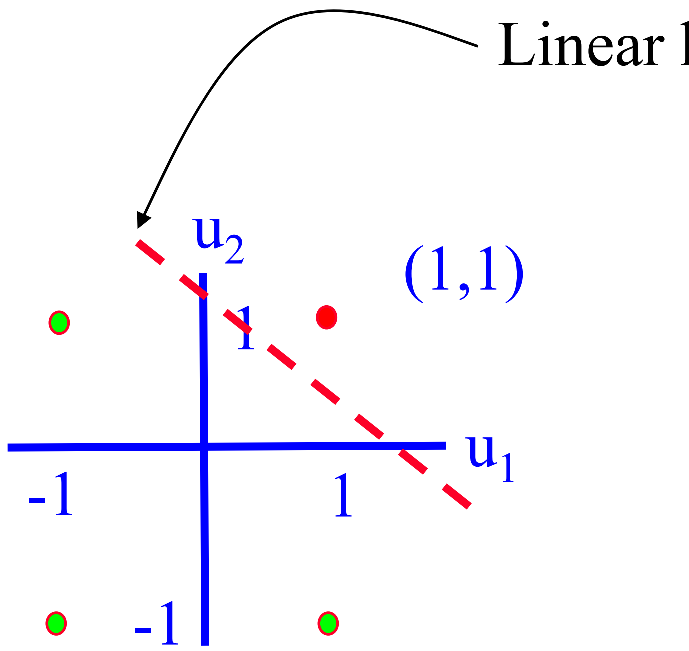
- Everything *on one side* of this hyperplane is in class 1 (output = +1) and everything *on other side* is class 2 (output = -1)

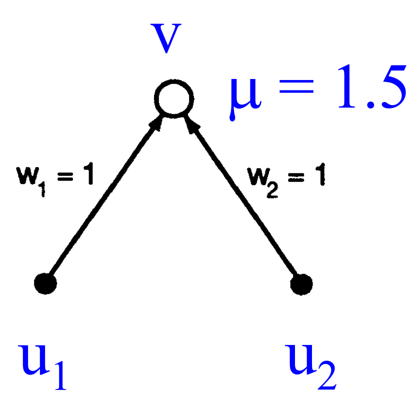
Any function that is linearly separable can be computed by a perceptron

Linear Separability

Example: AND is linearly separable

u_1	u_2	AND
-1	-1	-1
1	-1	-1
-1	1	-1
1	1	1





$v = 1$ iff $u_1 + u_2 - 1.5 > 0$

Similarly for OR and NOT

How do we *learn* the appropriate weights given only examples of (input,output)?

Idea: Change the weights to decrease the error in ouput

Perceptron Learning Rule

Given input pair (u, v^d) where $v^d \in \{+1,-1\}$ is the desired output, adjust w and μ as follows:

1. **Calculate current output v of neuron**

$$v = \Theta\left(\sum_j w_j u_j - \mu\right) = \Theta(\mathbf{w}^T \mathbf{u} - \mu)$$

2. **Compute <u>error signal</u> *e* = (v^d - v)**

Perceptron Learning Rule

3. Change w and μ according to error $(v^d - v)$:

   If input is positive and error is positive, then w not large enough $\Rightarrow$ increase w

   If input is positive and error is negative, then w too large $\Rightarrow$ decrease w

   Similar reasoning for other cases yields:

$$\mathbf{w} \to \mathbf{w} + \alpha(v^d - v)\mathbf{u}$$

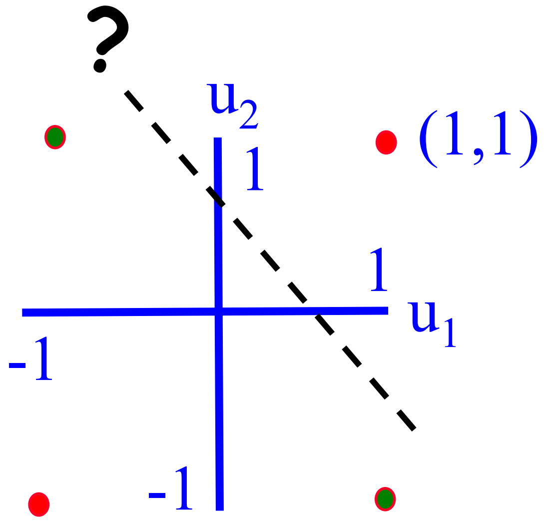
$$\mu \to \mu - \alpha(v^d - v)$$

$A \to B$ means replace A with B

α is the "learning rate" (a small positive number, e.g., 0.05)

What about the XOR function?

u_1	u_2	XOR
-1	-1	1
1	-1	-1
-1	1	-1
1	1	1



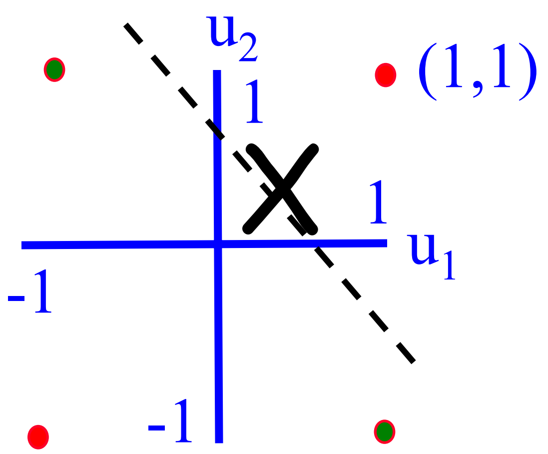
Can a perceptron separate the +1 outputs from the -1 outputs?

Linear Inseparability

Perceptron with threshold units fails if classification task is not linearly separable

- Example: XOR
- No single line can separate the "yes" (+1) outputs from the "no" (-1) outputs!

Minsky and Papert's book showing such negative results put a damper on neural networks research for over a decade!



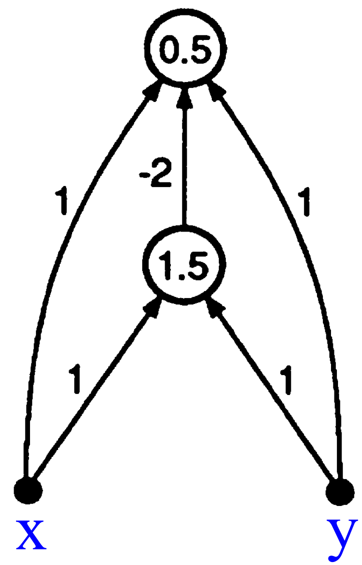
How do we deal with linear inseparability?

Idea 1: Multilayer Perceptrons

Removes limitations of single-layer networks

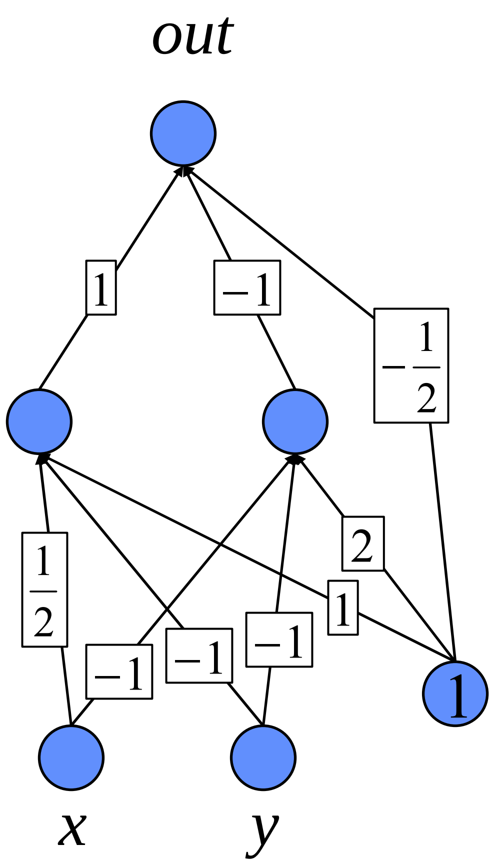
- Can solve XOR

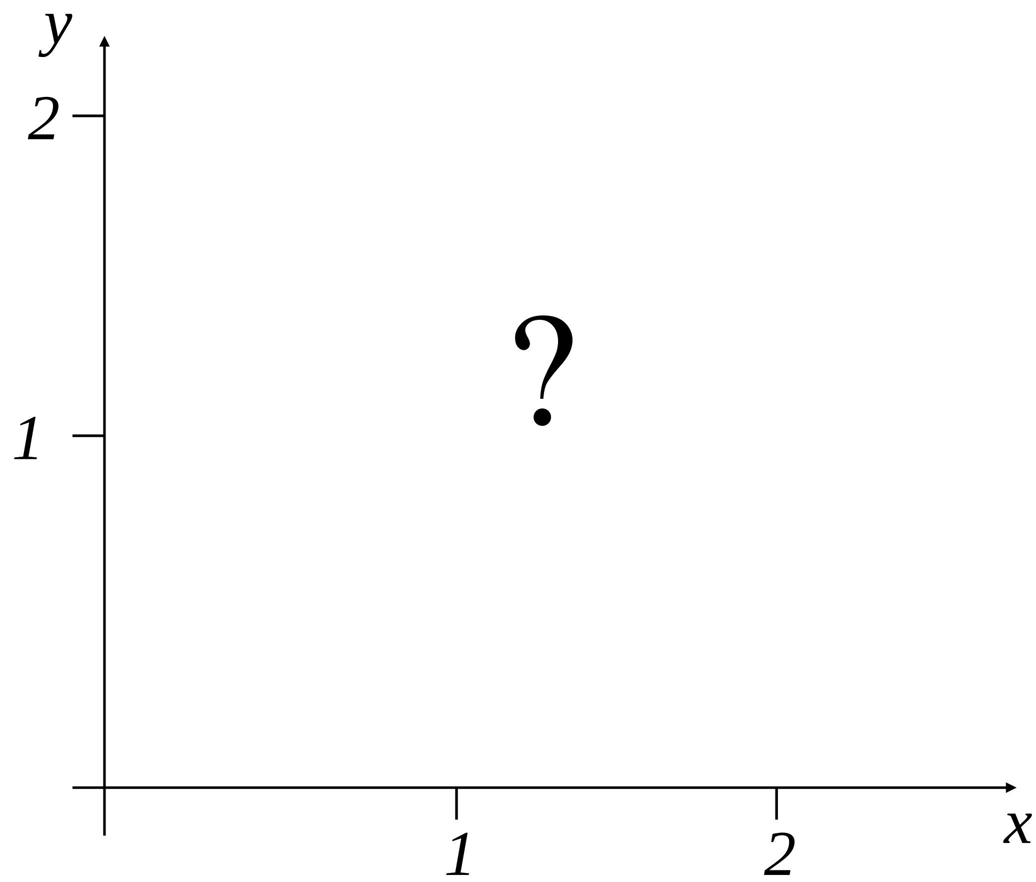
Example: Two-layer perceptron that computes XOR



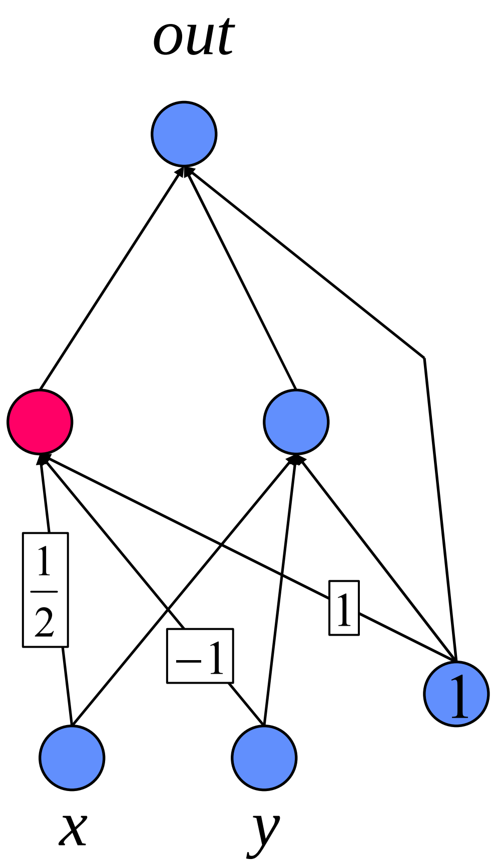
Output is +1 if and only if $x + y - 2\Theta(x + y - 1.5) - 0.5 > 0$

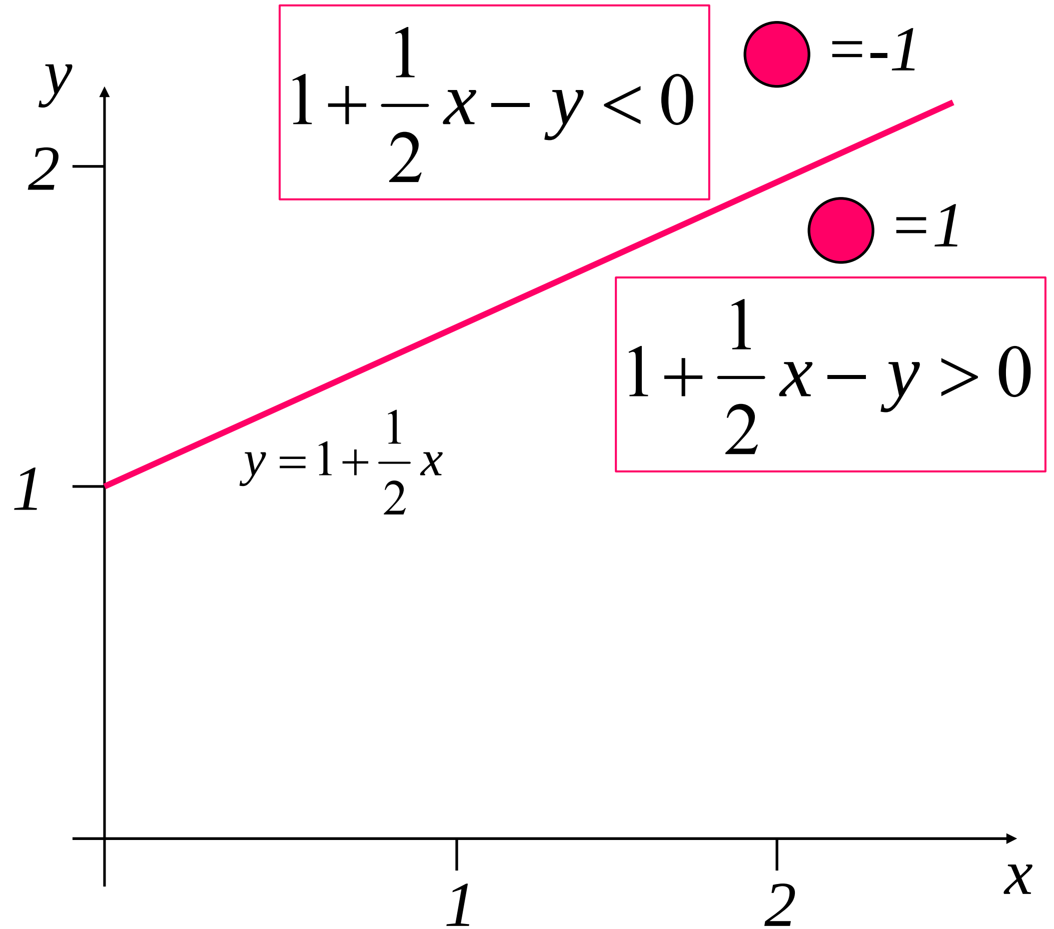
Multilayer Perceptron: What does it do?



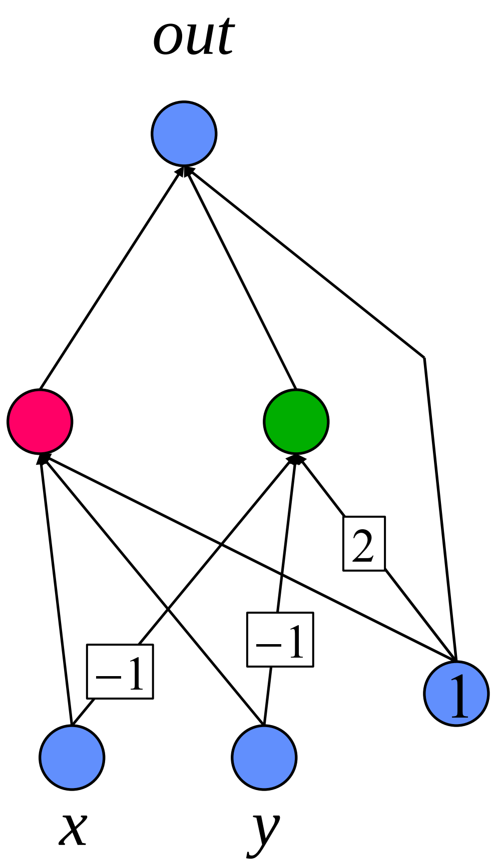


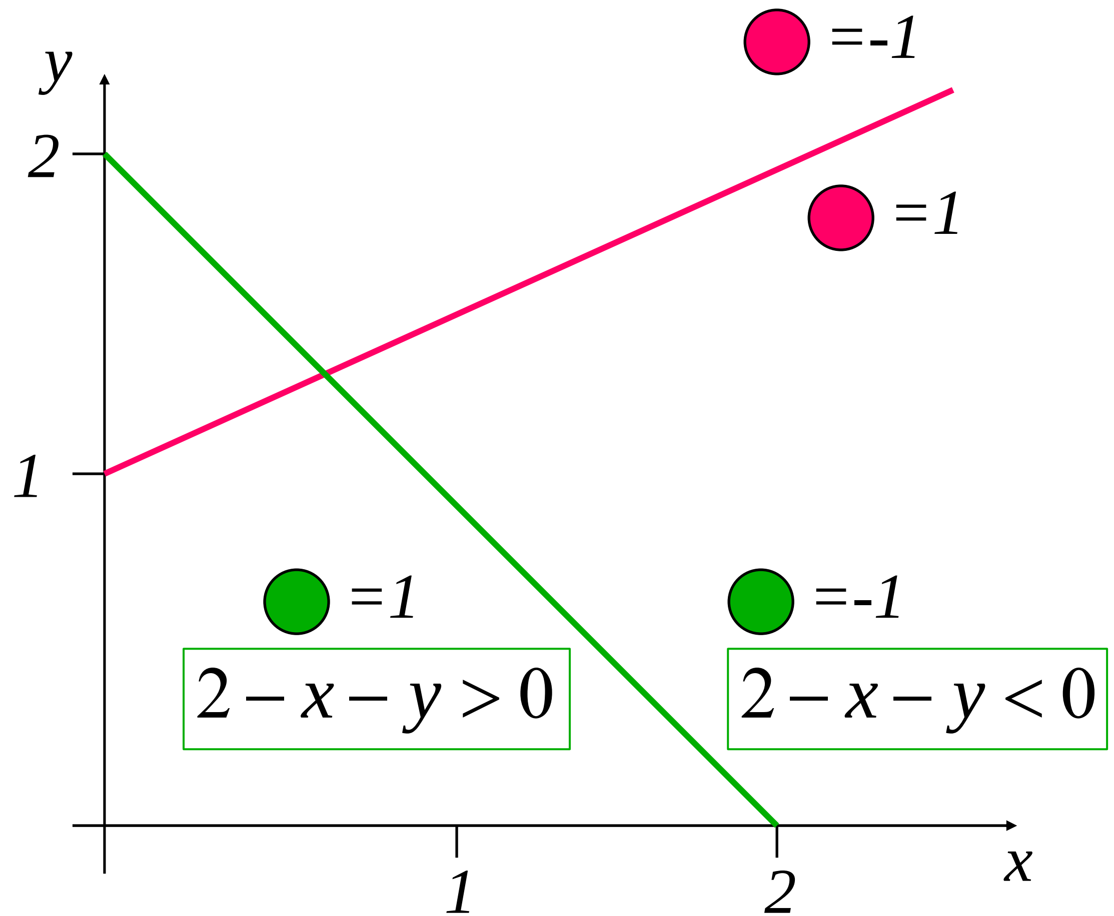
Multilayer Perceptron: What does it do?



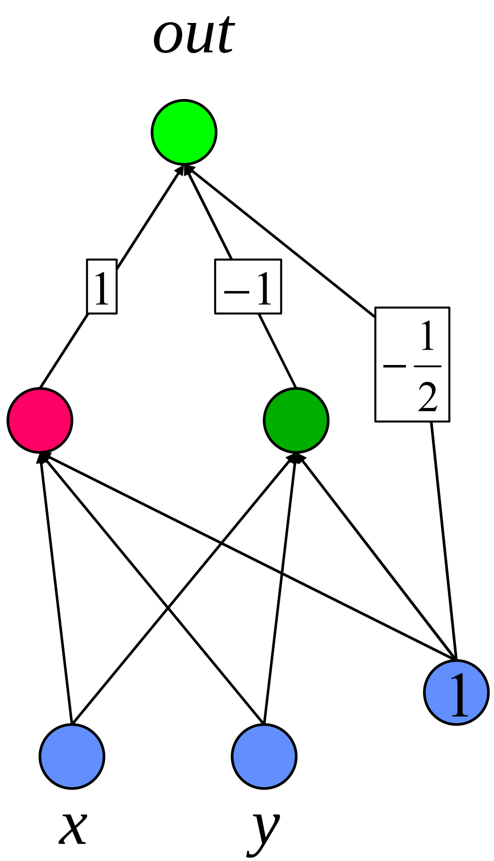


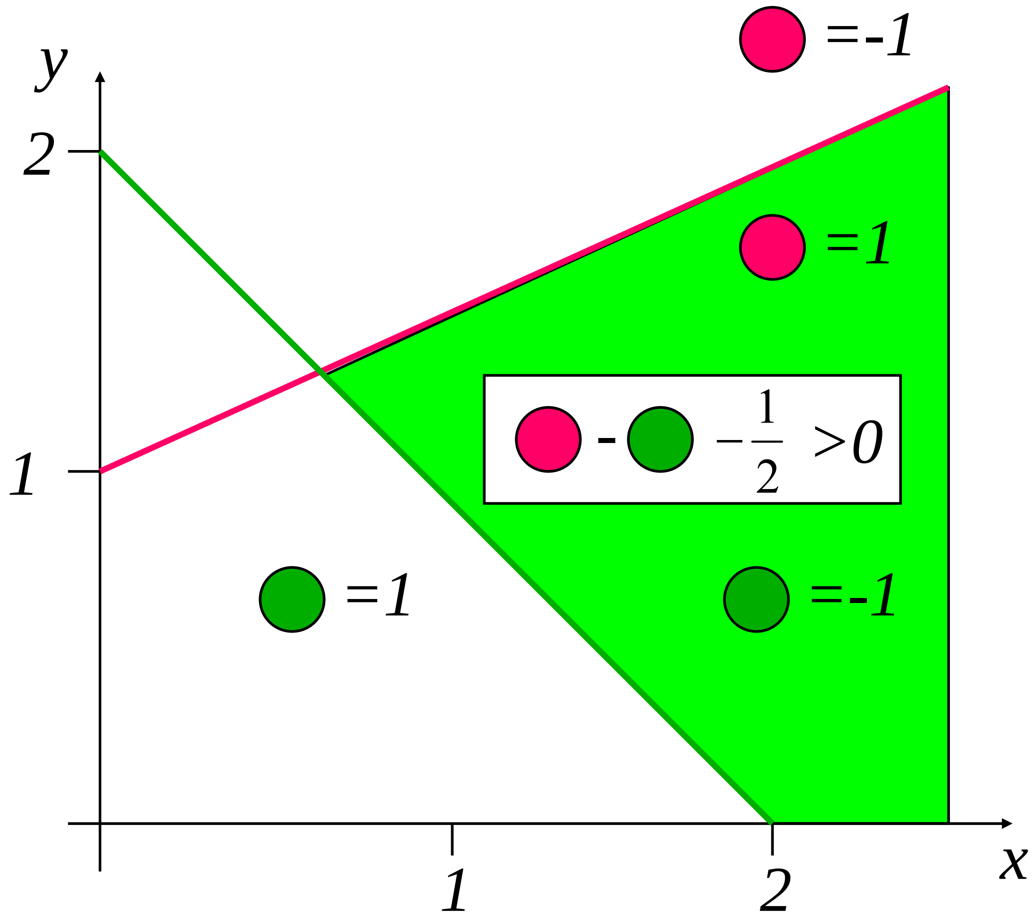
Multilayer Perceptron: What does it do?



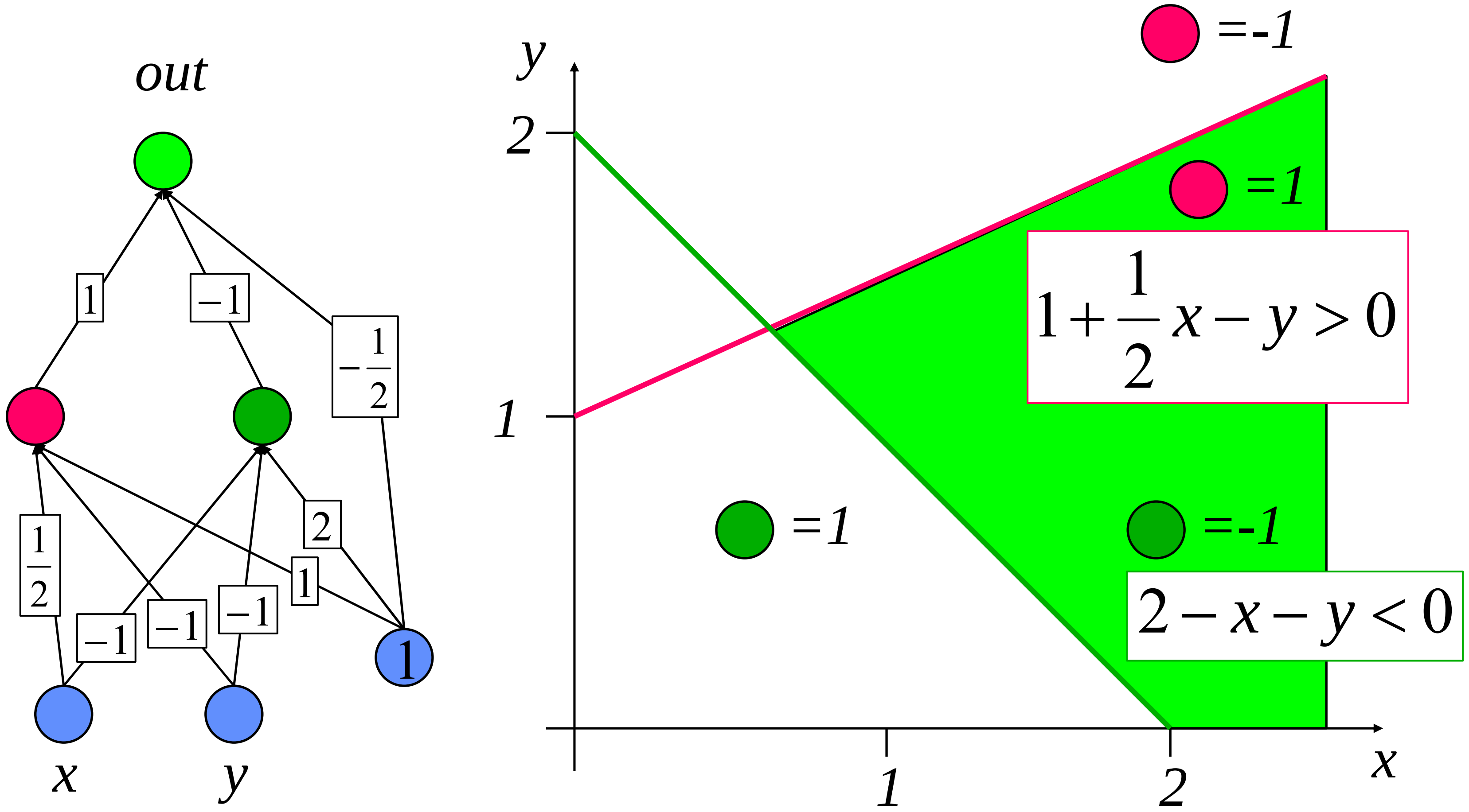


Multilayer Perceptron: What does it do?





Perceptrons as Constraint Satisfaction Networks



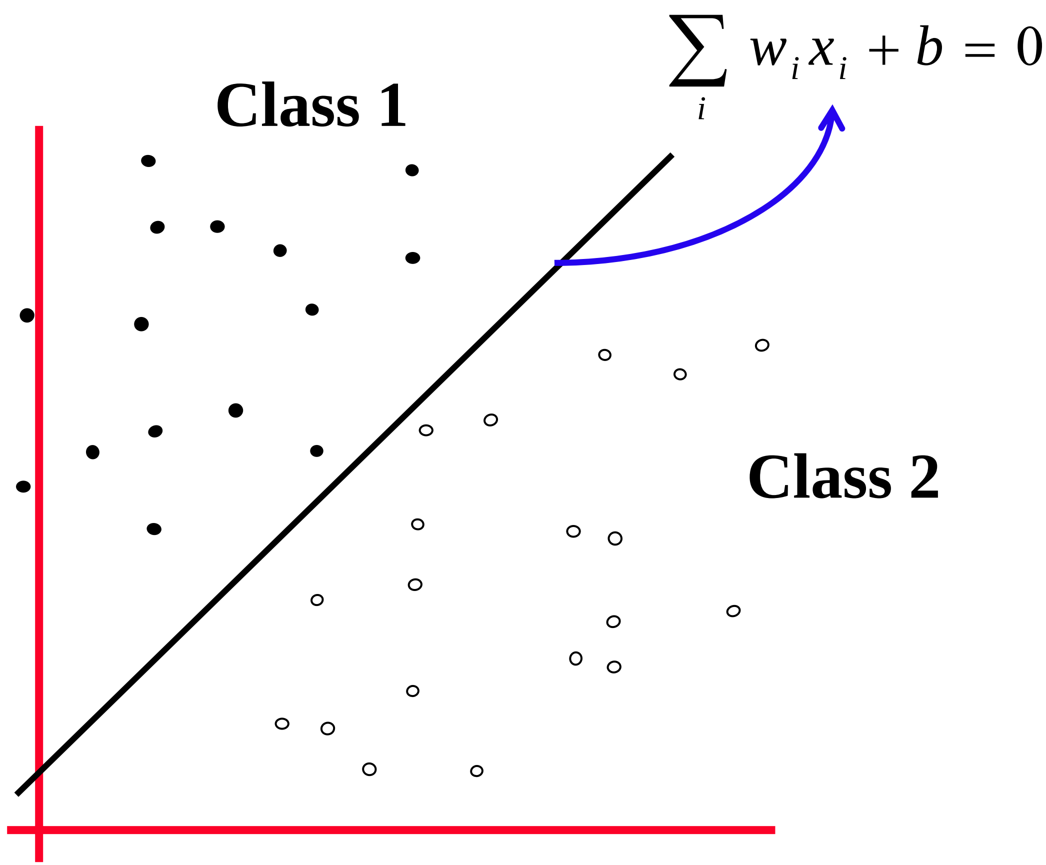
Back to Linear Separability

- **Recall: Weighted sum in perceptron forms a *linear hyperplane***

$$\sum_i w_i x_i + b = 0$$

- Due to threshold function, everything *on one side* of this hyperplane is labeled as class 1 (output = +1) and everything *on other side* is labeled as class 2 (output = -1)

Separating Hyperplane

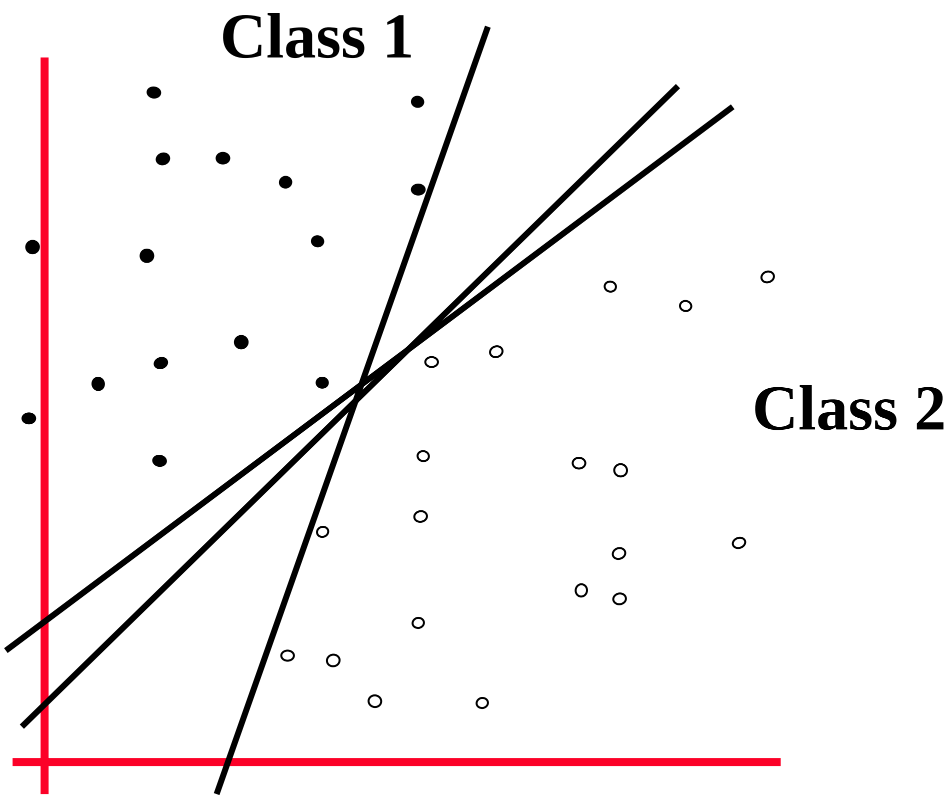


- • denotes +1 output
- ◦ denotes -1 output

Need to choose **w** and *b* based on training data

Separating Hyperplanes

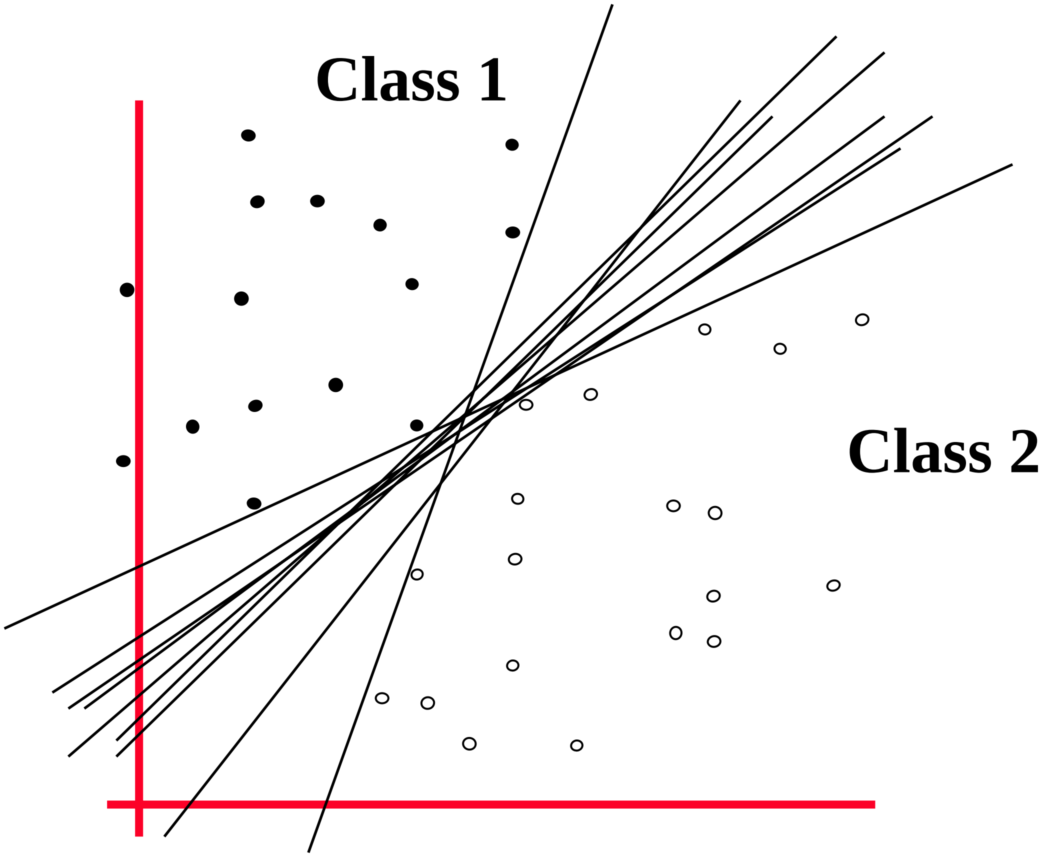
Different choices of **w** and *b* give different hyperplanes



- • denotes +1 output
- ◦ denotes -1 output

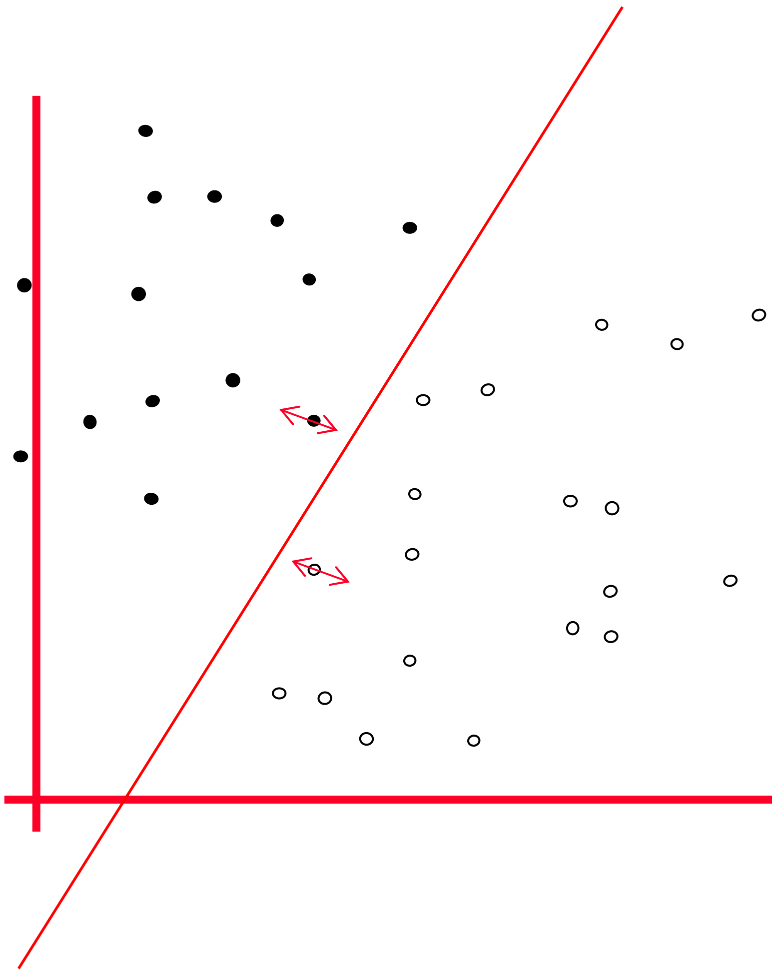
(This and next few slides adapted from Andrew Moore's)

Which hyperplane is best?



- • denotes +1 output
- ◦ denotes -1 output

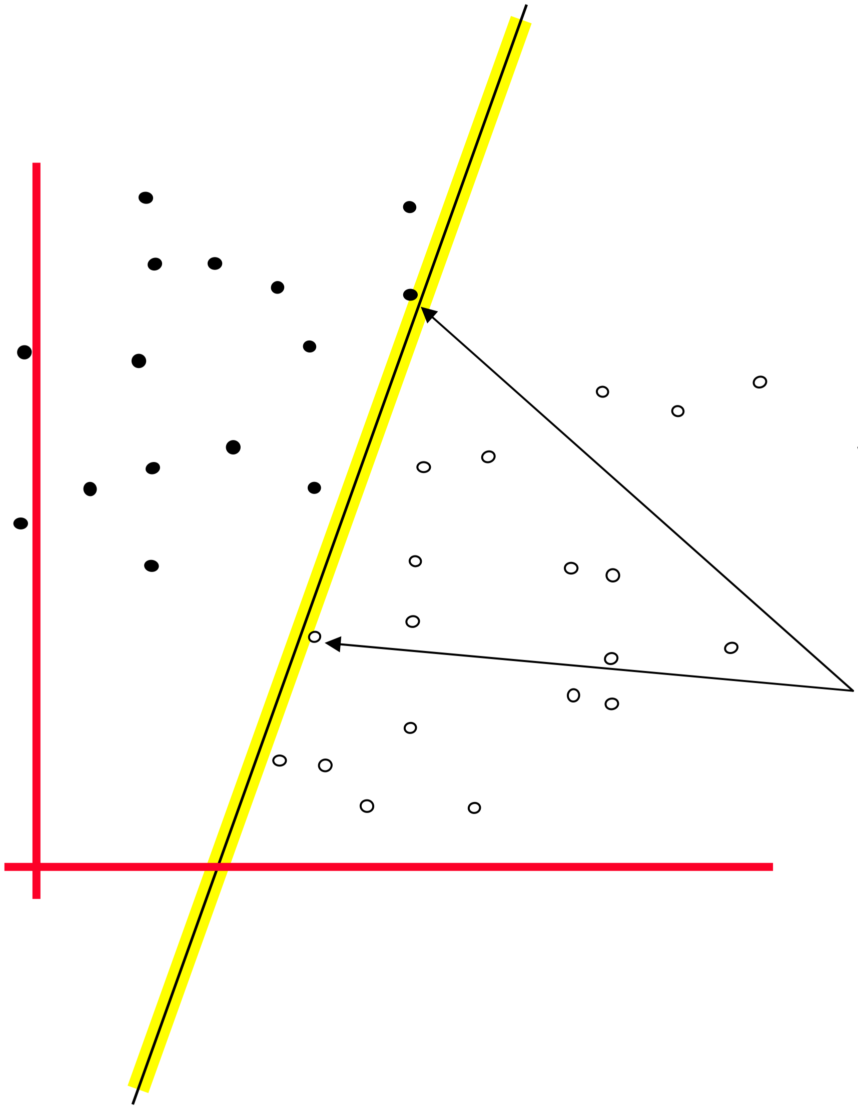
How about the one right in the middle?



Intuitively, this boundary seems good

Avoids misclassification of new test points if they are generated from the same distribution as training points

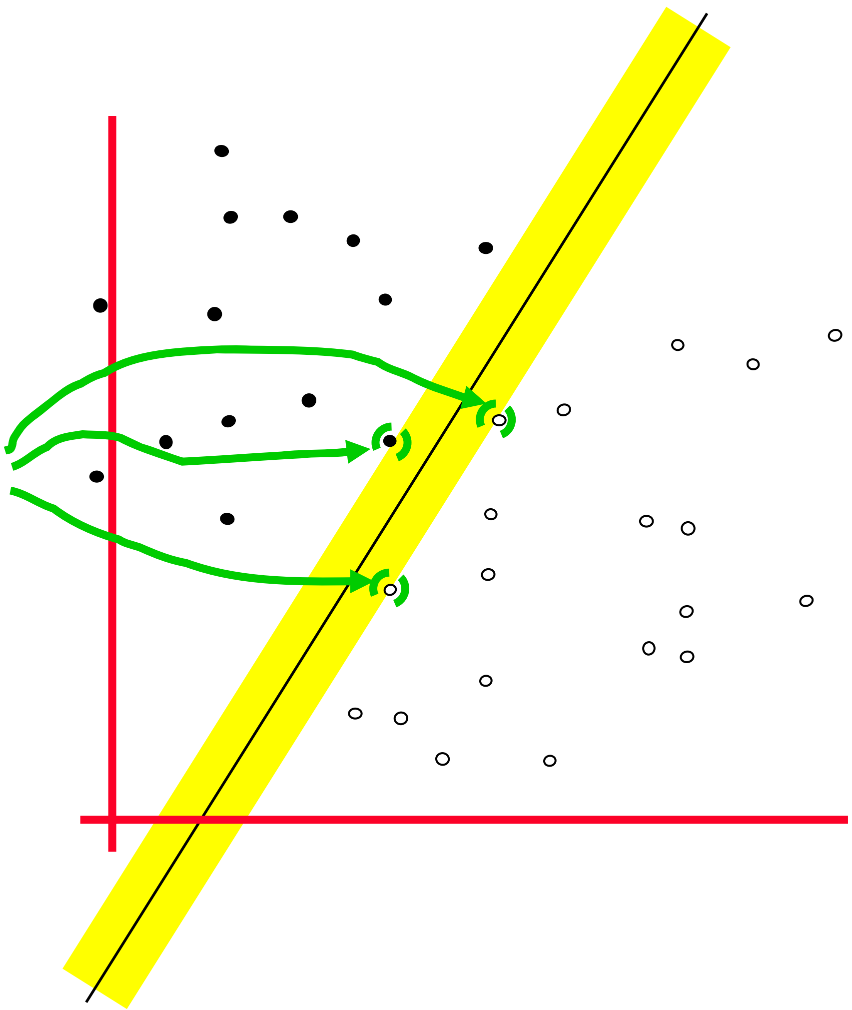
Margin



Define the margin of a linear classifier as the width that the boundary could be increased by before hitting a datapoint.

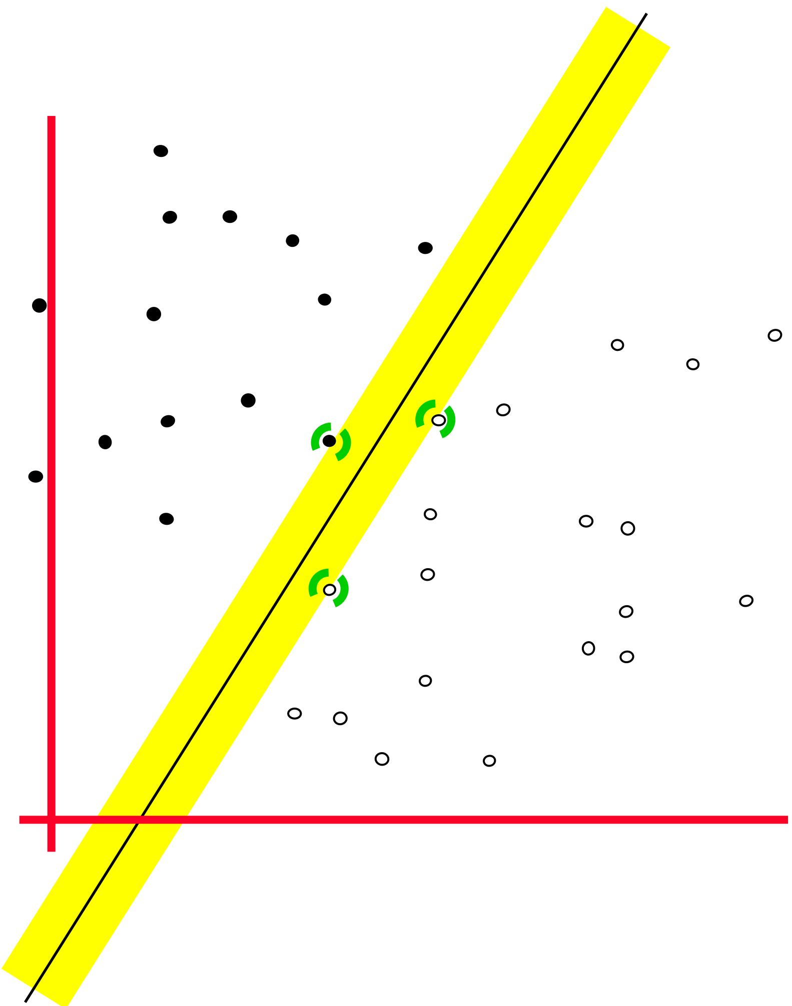
Maximum Margin and Support Vector Machine

Support Vectors are those datapoints that the margin pushes up against



The maximum margin classifier is called a Support Vector Machine (in this case, a Linear SVM or LSVM)

Why Maximum Margin?



- Robust to small perturbations of data points near boundary
- There exists theory showing this is best for generalization to new points
- Empirically works great

Support Vector Machines: The Math

Suppose the training data points $(\mathbf{x}_i, y_i)$ satisfy :

$$\mathbf{w} \cdot \mathbf{x}_i + b \geq +1 \text{ for } y_i = +1$$

$$\mathbf{w} \cdot \mathbf{x}_i + b \leq -1 \text{ for } y_i = -1$$

This can be rewritten as

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq +1$$

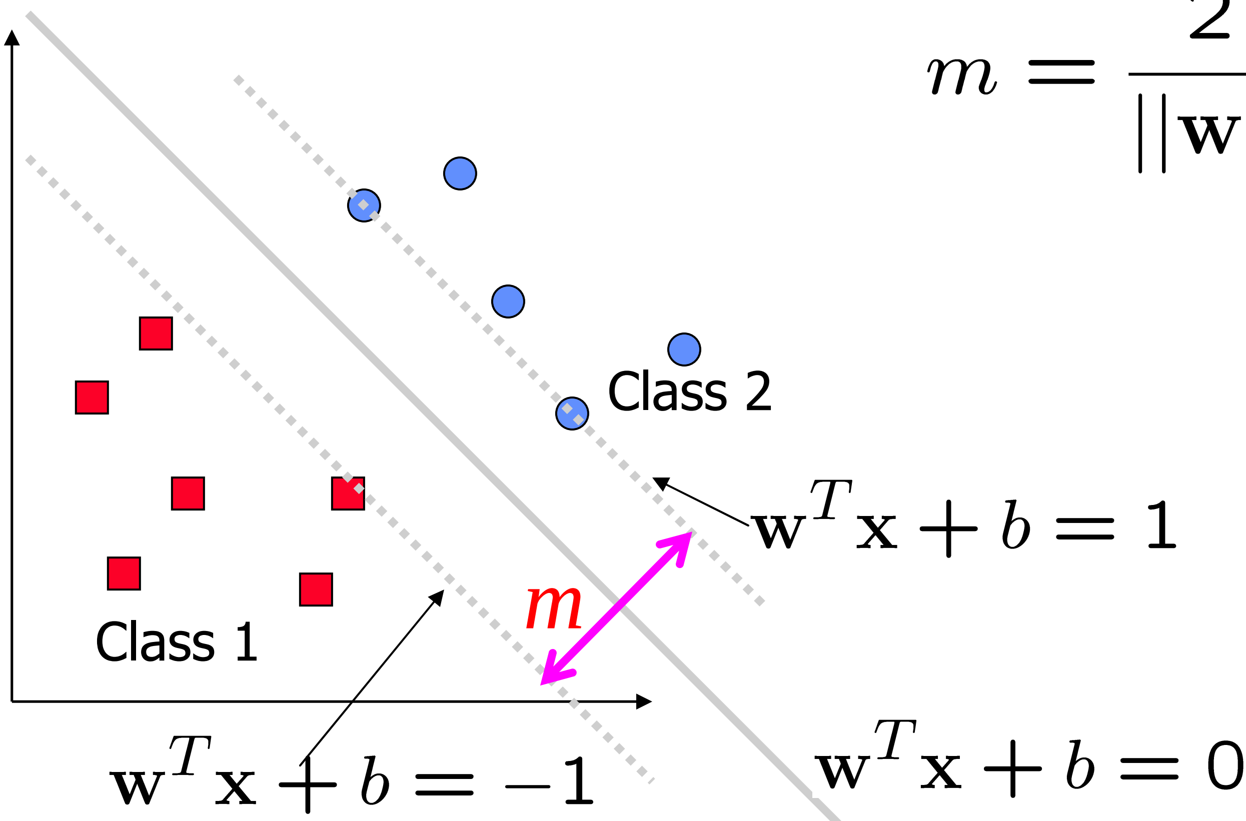
We can always do this by rescaling **w** and *b*, without affecting the separating hyperplane:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

Estimating the Margin

The margin is given by (see Burges tutorial online):

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



Margin can be calculated based on expression for distance from a point to a line, see, e.g., http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html

Learning the Maximum Margin Classifier

Want to maximize margin:

$$2/\|\mathbf{w}\| \text{ subject to } \quad y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq +1, \forall i$$

Equivalent to finding *w* and *b* that minimize:

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

Constrained optimization problem that can be solved using Lagrange multiplier method

Learning the Maximum Margin Classifier

Using Lagrange formulation and Lagrangian multipliers α_i, we get (see Burges tutorial online):

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

where the α_i are obtained by maximizing:

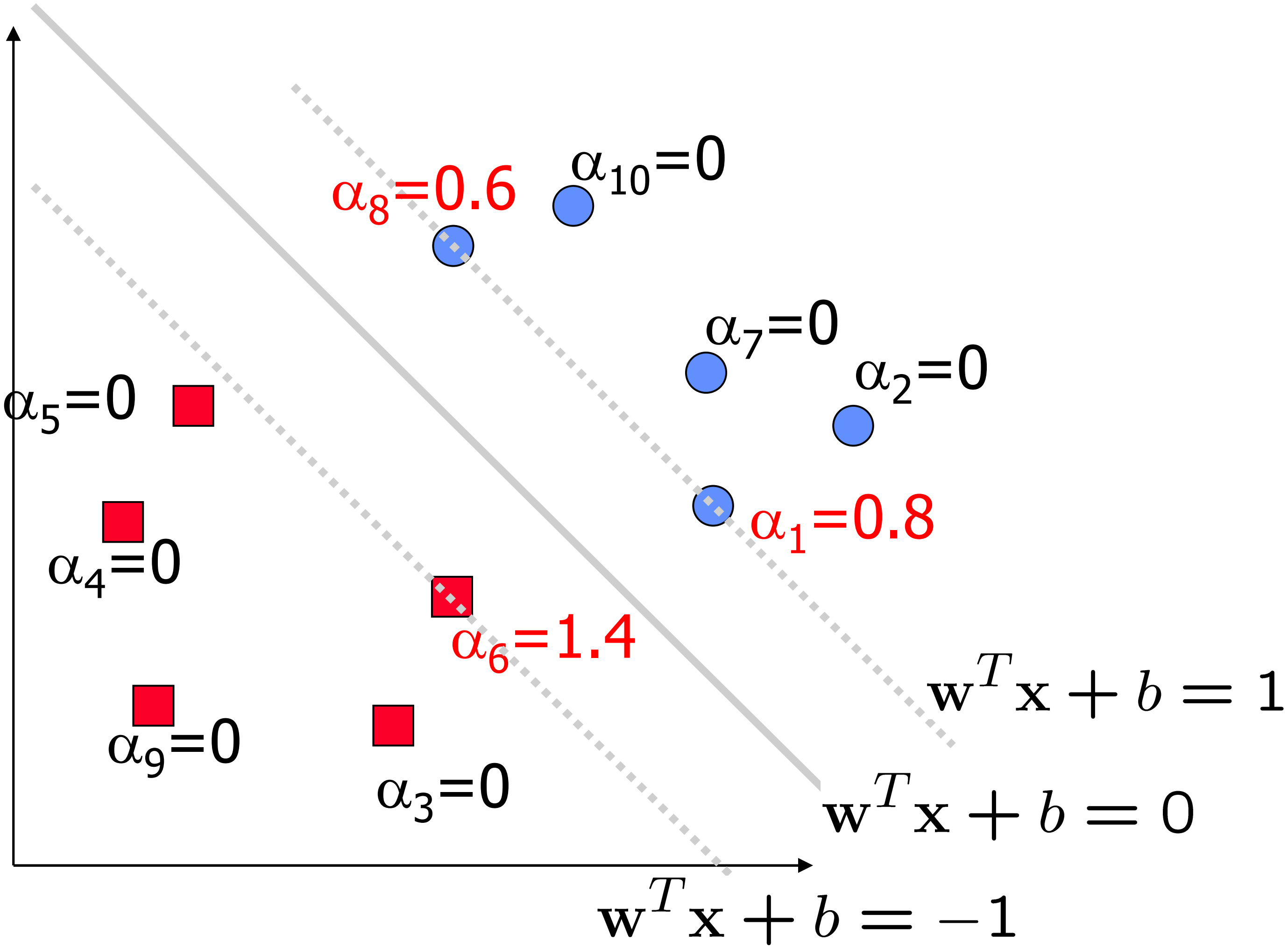
$$\sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$\text{subject to} \quad \alpha_i \geq 0 \text{ and } \sum_i \alpha_i y_i = 0$$

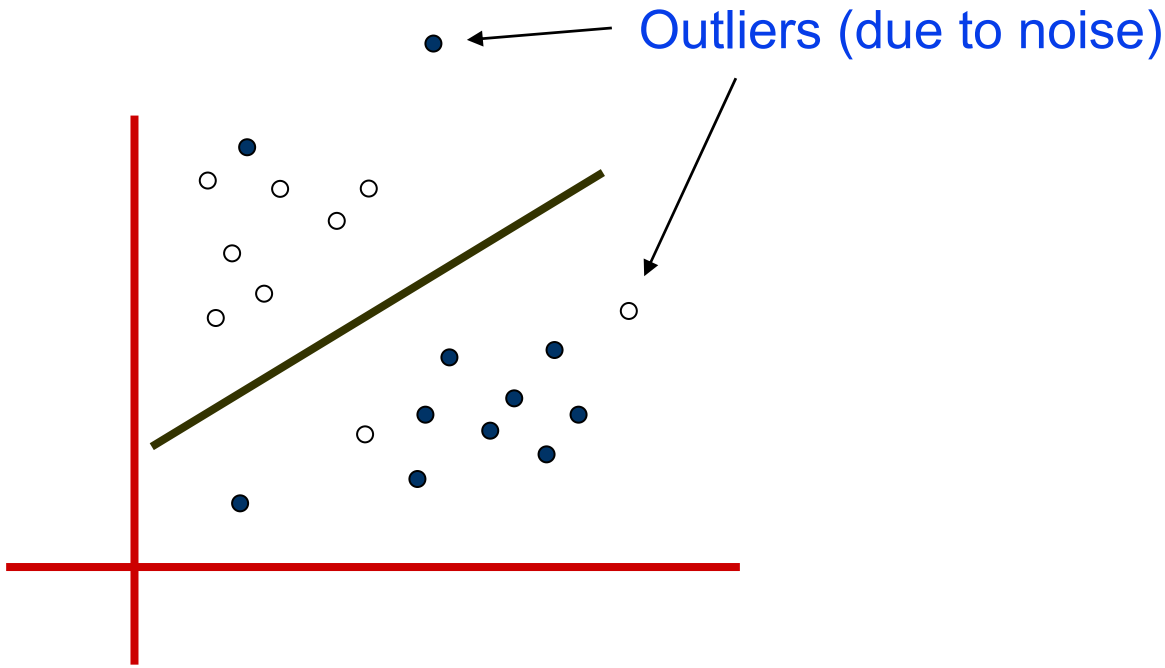
This is a quadratic programming (QP) problem
- A global maximum can always be found

Geometrical Interpretation

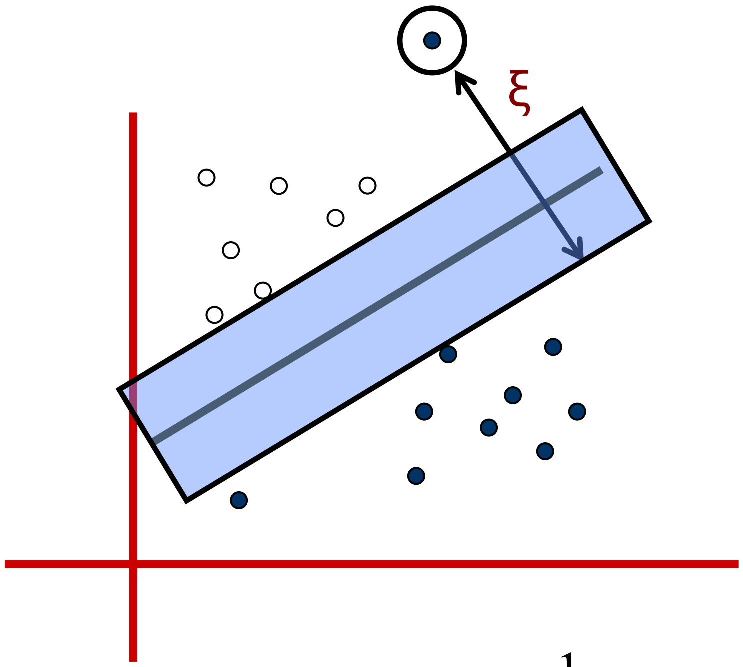
$\mathbf{x}_i$ with non-zero α_i are called support vectors



What if data is not linearly separable?



Approach 1: Soft Margin SVMs



Allow *errors* ξ_i (deviations from margin)

Trade off margin with errors.

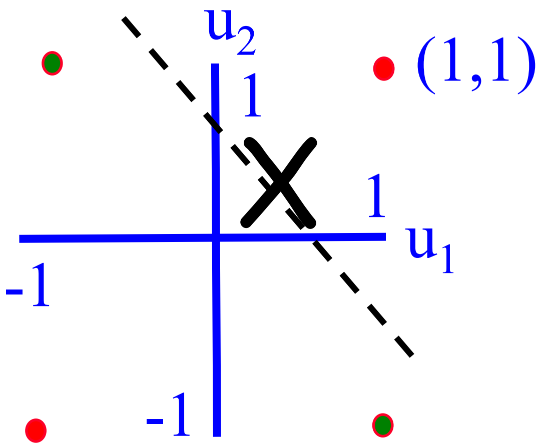
Minimize:

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i \text{ subject to } :$$

$$y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0, \forall i$$

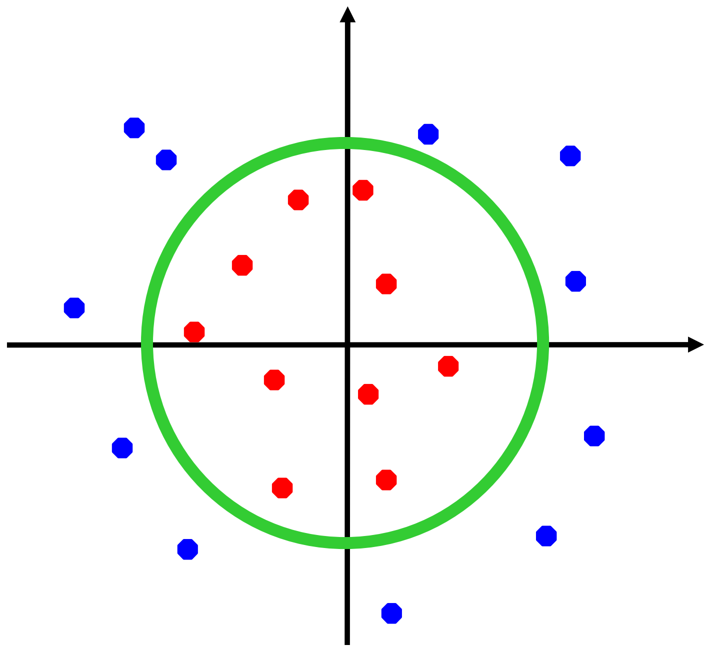
What if data is not linearly separable: Other ideas?

u_1	u_2	XOR
-1	-1	1
1	-1	-1
-1	1	-1
1	1	1



Can we do something to the inputs?

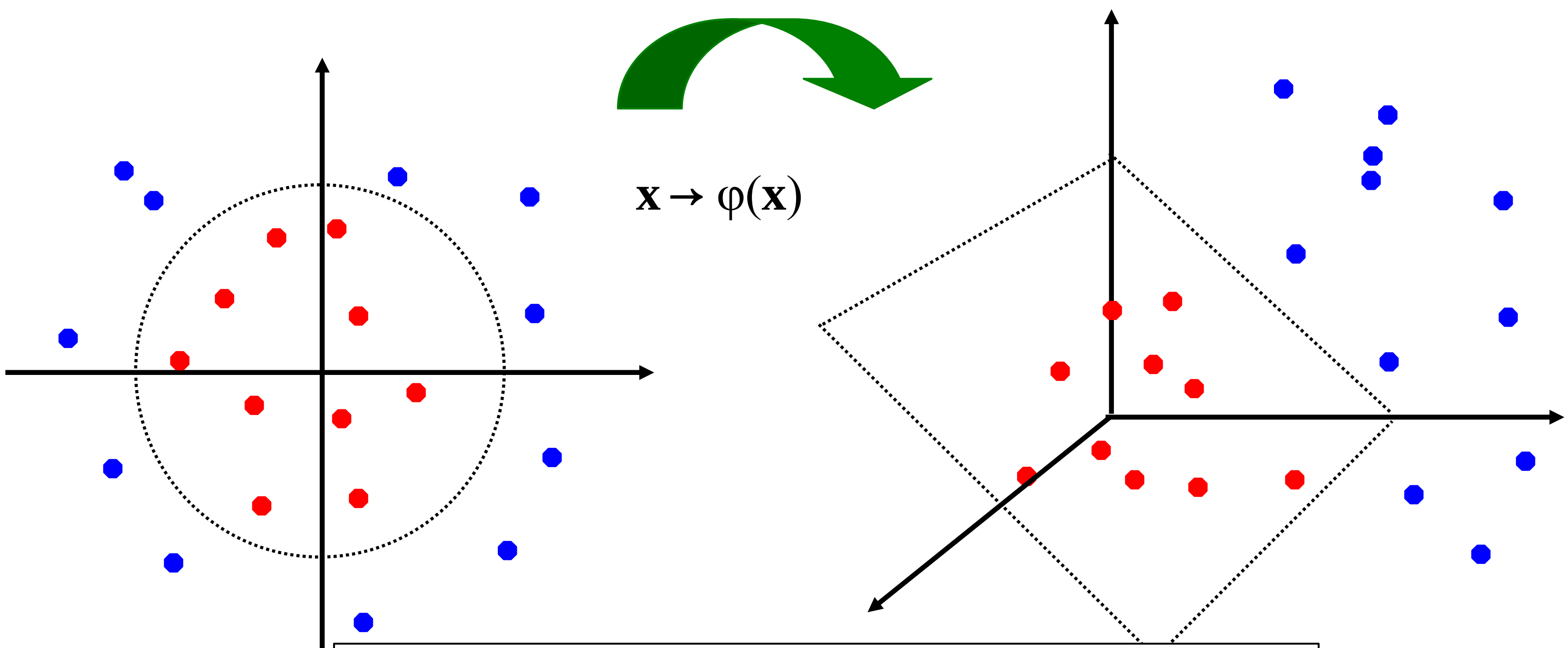
Another Example



Not linearly separable

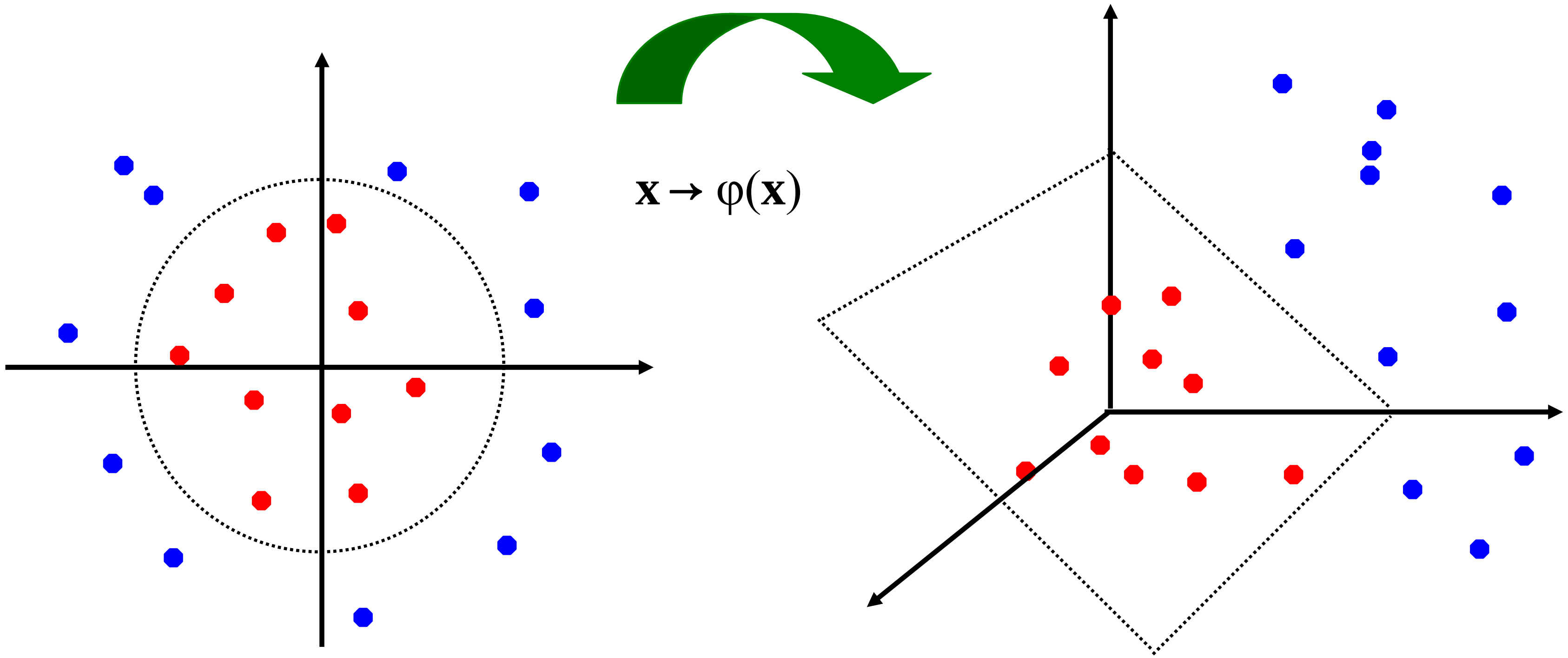
What if data is not linearly separable?

Approach 2: Map original input space to higher-dimensional feature space; use linear classifier in higher-dim. space



$$\Phi : \mathbb{R}^2 \to \mathbb{R}^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{2}\, x_1 x_2, x_2^2)$$

Problem with high dimensional spaces



Computation in high-dimensional feature space can be costly

The high dimensional projection function $\varphi(x)$ may be too complicated to compute

Kernel trick to the rescue!

The Kernel Trick

Recall: SVM maximizes the quadratic function:

$$\sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$\text{subject to} \quad \alpha_i \geq 0 \text{ and } \sum_i \alpha_i y_i = 0$$

Insight:

The data points only appear as inner product

- No need to compute high-dimensional $\varphi(x)$ explicitly! Just replace inner product $x_i \cdot x_j$ with a kernel function $K(x_i,x_j) = \varphi(x_i) \cdot \varphi(x_j)$
- E.g., Gaussian kernel

  $K(x_i,x_j) = \exp(-||x_i-x_j||^2/2\sigma^2)$
- E.g., Polynomial kernel

  $K(x_i,x_j) = (x_i \cdot x_j+1)^d$

An Example for $\phi(.)$ and $K(.,.)$

Suppose $\phi(.)$ is given as follows

$$\phi\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

An inner product in the feature space is

$$\left\langle \phi\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right), \phi\left(\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}\right) \right\rangle = (1 + x_1y_1 + x_2y_2)^2$$

So, if we define the kernel function as follows, there is no need to compute $\phi(.)$ explicitly

$$K(\mathbf{x}, \mathbf{y}) = (1 + x_1y_1 + x_2y_2)^2$$

This use of kernel function to avoid computing $\phi(.)$ explicitly is known as the kernel trick

Summary: Steps for Classification using SVMs

Prepare the data matrix

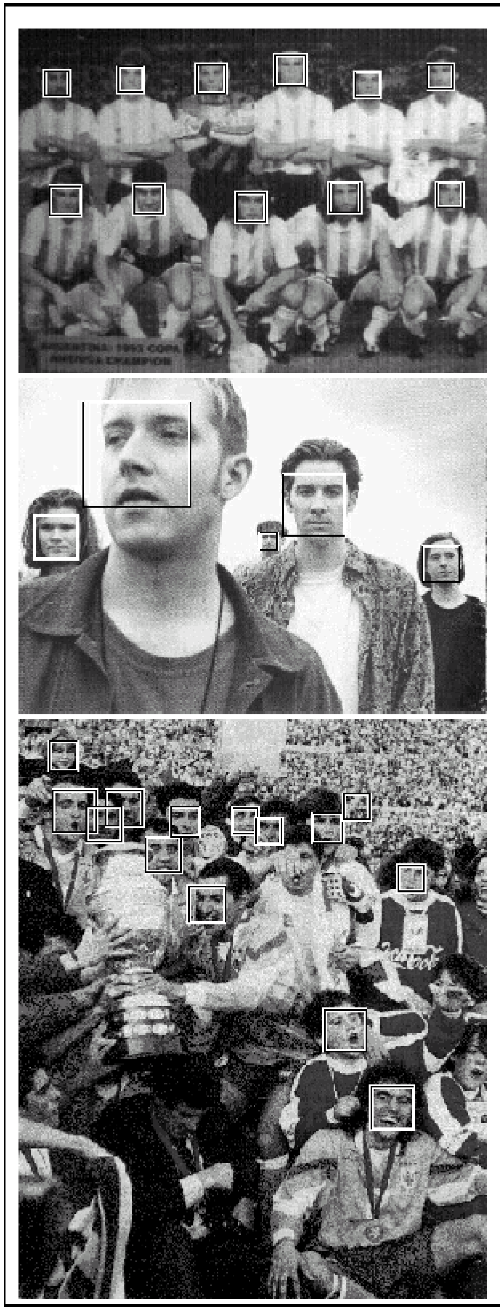
Select the kernel function to use

Select parameters of the kernel function

- You can use the values suggested by the SVM software, or use cross-validation

Execute the training algorithm and obtain the parameters α_i

Classify new data using the learned parameters



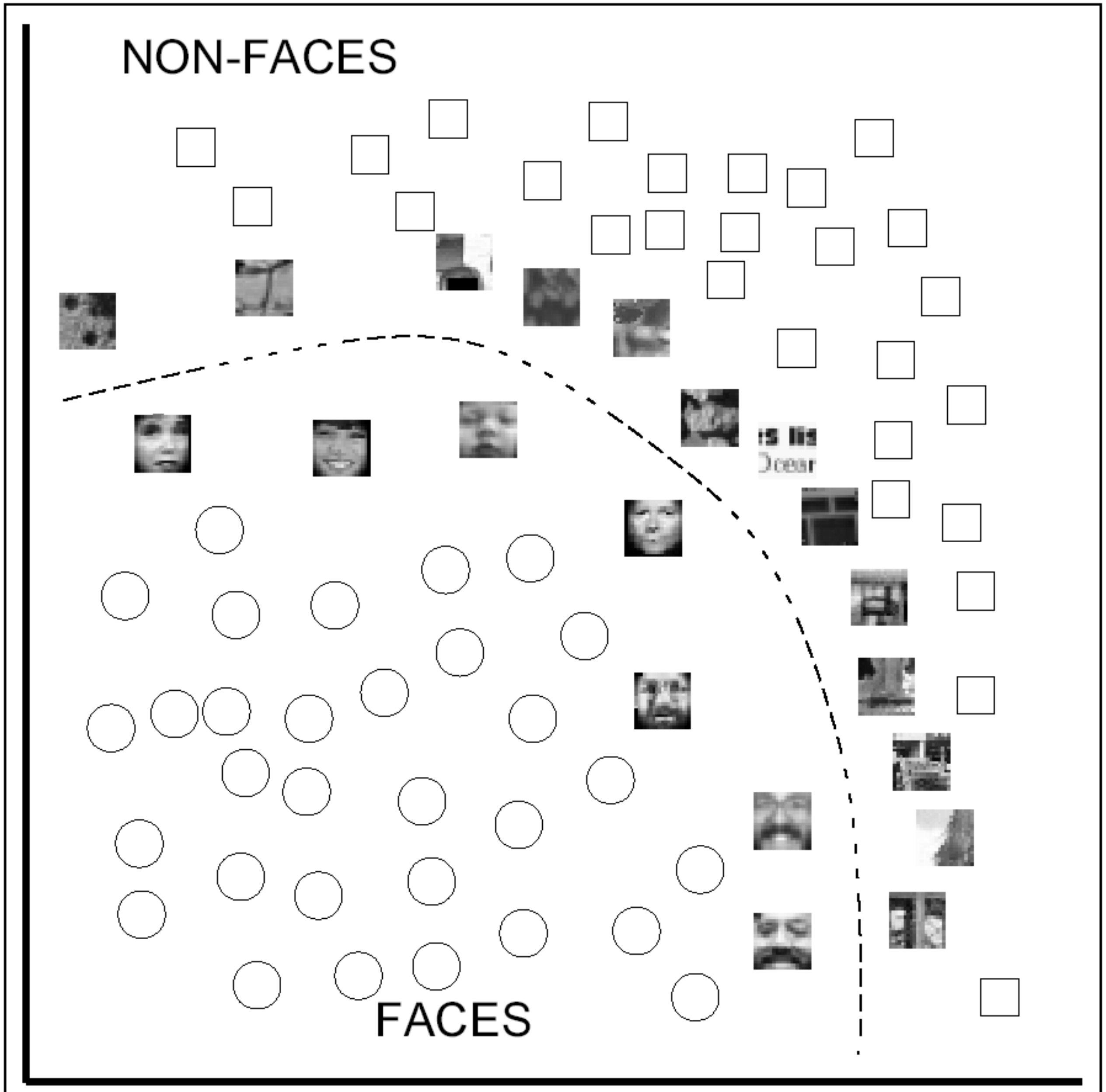
Face Detection using SVMs

	Test Set A		Test Set B	
	Detect Rate	False Alarms	Detect Rate	False Alarms
SVM	97.1 %	4	74.2%	20
Sung *et al.*	94.6 %	2	74.2%	11

Kernel used: Polynomial of degree 2

(Osuna, Freund, Girosi, 1998)

Support Vectors



K-Nearest Neighbors

A simple non-parametric classification algorithm

Idea:

- Look around you to see how your neighbors classify data
- Classify a new data-point according to a *majority vote* of your k nearest neighbors

Distance Metric

How do we measure what it means to be a neighbor (what is "close")?

Appropriate distance metric depends on the problem

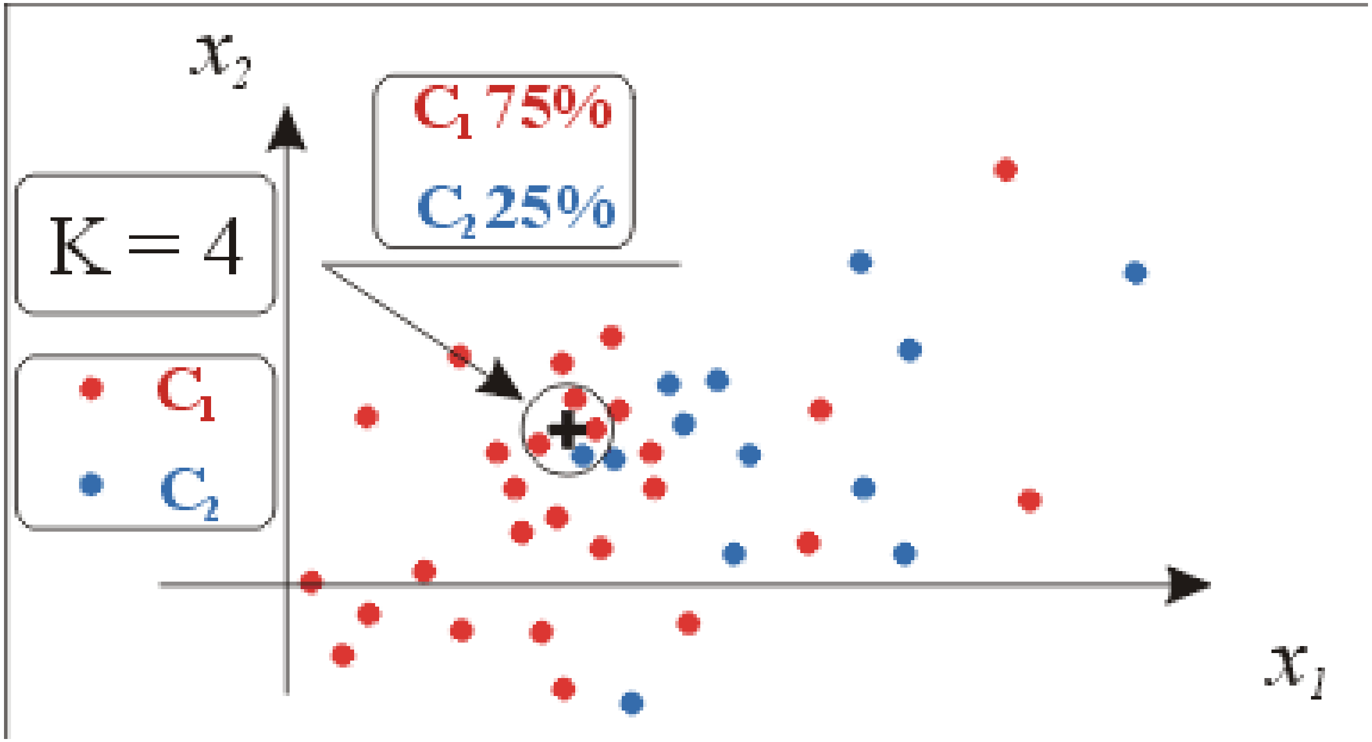
Examples:

- x discrete (e.g., strings): Hamming distance
  - $d(x_1,x_2)$ = # features on which x_1 and x_2 differ
- x continuous (e.g., vectors over reals): Euclidean distance
  - $d(x_1,x_2) = || x_1 - x_2 ||$ = square root of sum of squared differences between corresponding elements of data vectors

Example

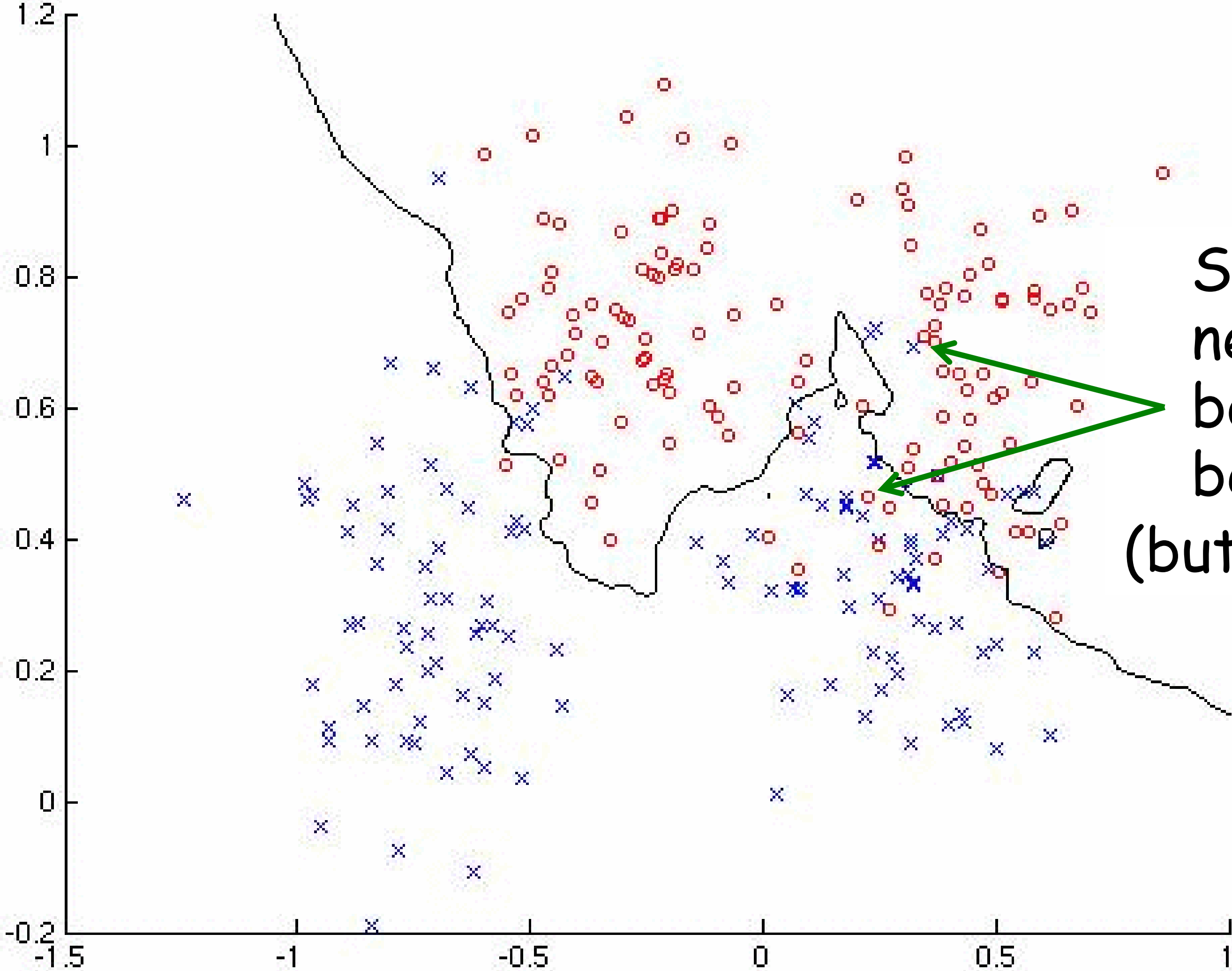
Input Data: 2-D points (x_1, x_2)

Two classes: C_1 and C_2. New Data Point +



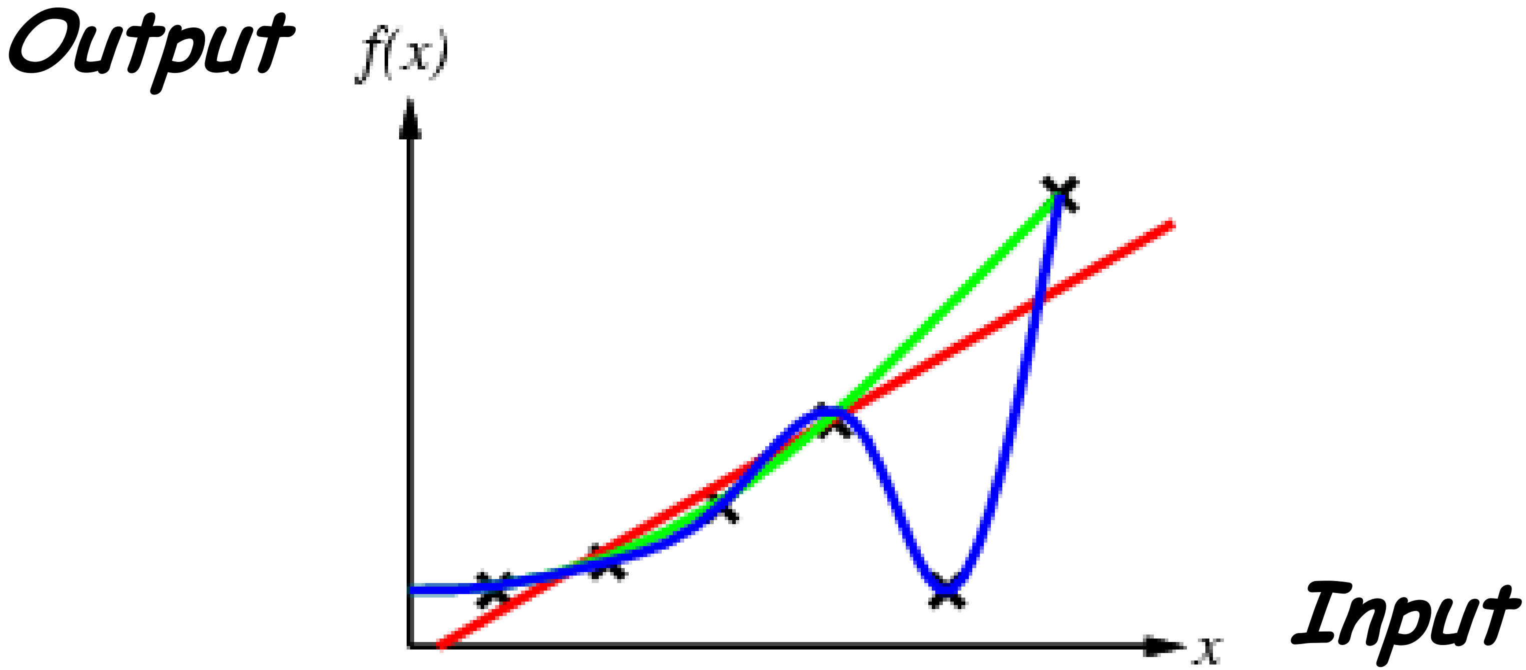
K = 4: Look at 4 nearest neighbors of +
3 are in C_1, so classify + as C_1

Decision Boundary using K-NN



Some points near the boundary may be misclassified (but maybe noise)

What if we want to learn continuous-valued functions?



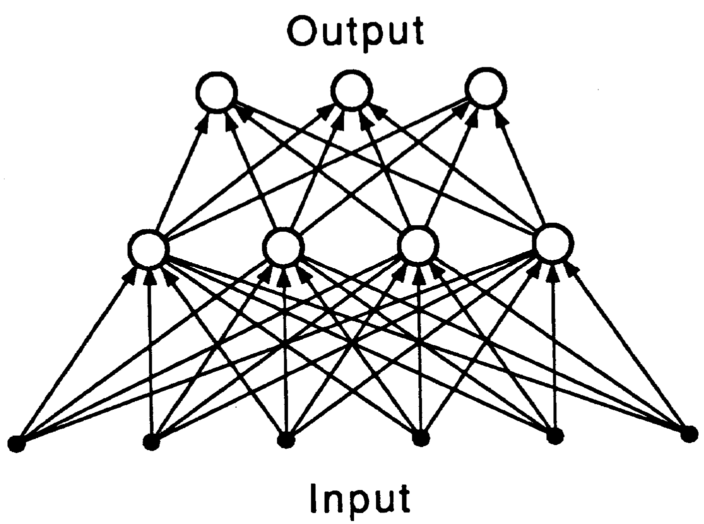
Example: Learning to Drive

Can you use a neural network to drive?

Regression using Networks

We want networks that can <u>learn a function</u>

- Network maps real-valued inputs to real-valued output
- <u>Idea</u>: Given data, *minimize errors* between network's output and desired output by changing weights



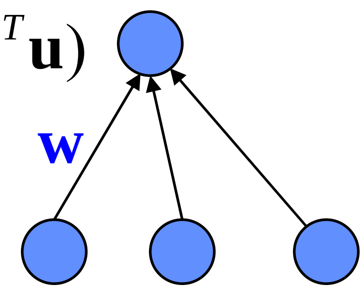
Continuous output values → Can't use binary threshold units anymore

To minimize errors, a *differentiable* output function is desirable

Sigmoidal Networks

The most common activation function:

$$v = g(\mathbf{w}^T \mathbf{u})$$



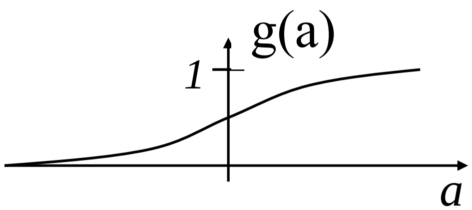
Output

Input nodes

$\mathbf{u} = (u_1 \quad u_2 \quad u_3)^T$

Sigmoid function:

$$g(a) = \frac{1}{1 + e^{-\beta a}}$$



Non-linear "squashing" function: Squashes input to be between 0 and 1. The parameter β controls the slope.

Gradient-Descent Learning ("Hill-Climbing")

Given training examples (u^m, d^m) (m = 1, …, N), define an error function (cost function or "energy" function)

$$E(\mathbf{w}) = \frac{1}{2}\sum_m (d^m - v^m)^2$$

where $v^m = g(\mathbf{w}^T \mathbf{u}^m)$

Gradient-Descent Learning ("Hill-Climbing")

Would like to change w so that *E*(w) is minimized

- Gradient Descent: Change w in proportion to –d*E*/dw (why?)

$$\mathbf{w} \rightarrow \mathbf{w} - \varepsilon \frac{dE}{d\mathbf{w}}$$

$$\frac{dE}{d\mathbf{w}} = -\sum_m (d^m - v^m) \frac{dv^m}{d\mathbf{w}} = -\sum_m (d^m - v^m) g'(\mathbf{w}^T \mathbf{u}^m) \mathbf{u}^m$$

Derivative of sigmoid

"Stochastic" Gradient Descent

What if the inputs only arrive one-by-one?

Stochastic gradient descent approximates sum over all inputs with an "on-line" running sum:

$$\mathbf{w} \to \mathbf{w} - \varepsilon \frac{dE_1}{d\mathbf{w}}$$

$$\frac{dE_1}{d\mathbf{w}} = -\underbrace{(d^m - v^m)}_{\text{delta = error}} g'(\mathbf{w}^T \mathbf{u}^m)\mathbf{u}^m$$

Also known as the "delta rule" or "LMS (least mean square) rule"

But wait….

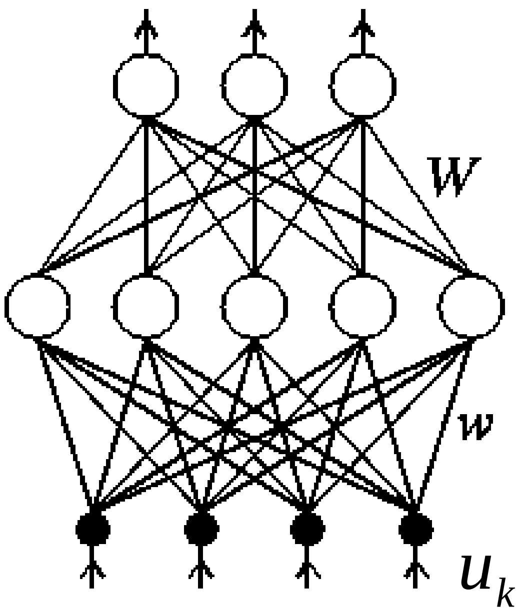
Delta rule tells us how to modify the connections from input to output (one layer network)

- One layer networks are not that interesting (remember XOR?)

What if we have multiple layers?

Learning Multilayer Networks

$$v_i = g\left(\sum_j W_{ji}\, g\left(\sum_k w_{kj} u_k\right)\right)$$



Start with random weights **W**, **w**

Given input **u**, network produces output **v**

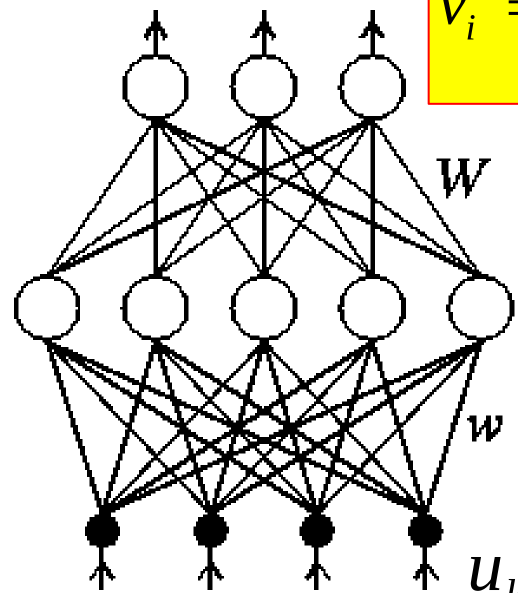
Find **W** and **w** that minimize total squared output error over all output units (labeled *i*):

$$E(\mathbf{W},\mathbf{w}) = \frac{1}{2}\sum_i (d_i - v_i)^2$$

Backpropagation: Output Weights

$$E(\mathbf{W},\mathbf{w}) = \frac{1}{2}\sum_i (d_i - v_i)^2$$

$$\boxed{v_i = g(\sum_j W_{ji} x_j)}$$



$$\boxed{x_j}$$

u_k

Learning rule for **hidden-output weights W**:

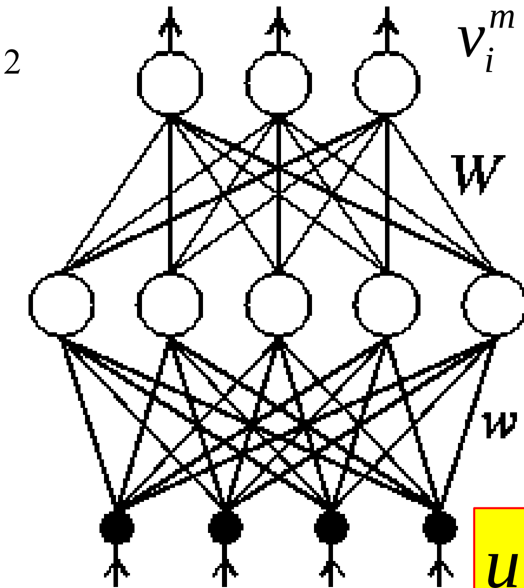
$$W_{ji} \rightarrow W_{ji} - \varepsilon \frac{dE}{dW_{ji}} \qquad \{\textit{gradient descent}\}$$

$$\frac{dE}{dW_{ji}} = -(d_i - v_i) g'(\sum_j W_{ji} x_j) x_j \qquad \{\textit{delta rule}\}$$

Backpropagation: Hidden Weights

$$E(\mathbf{W},\mathbf{w}) = \frac{1}{2}\sum_i (d_i - v_i)^2$$

$$v_i^m = g(\sum_j W_{ji} x_j)$$



$$\boxed{x_j^m = g(\sum_k w_{kj} u_k^m)}$$

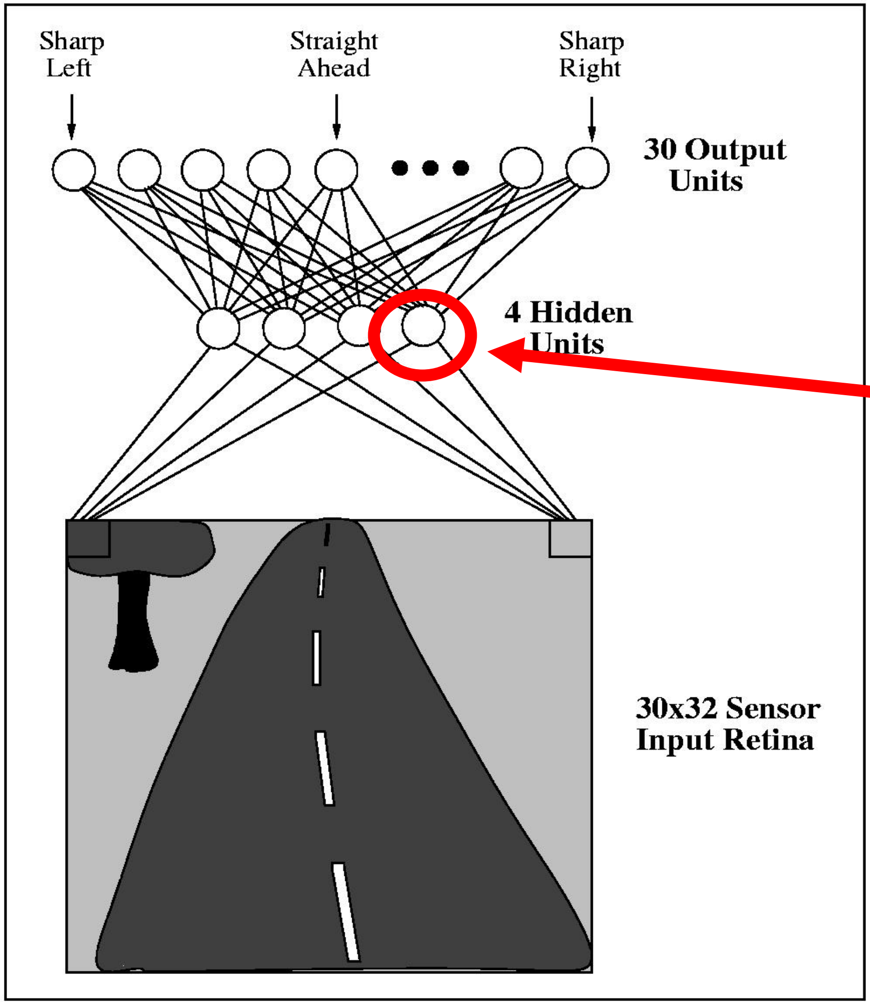
$$\boxed{u_k^m}$$

Learning rule for <u>input-hidden weights **w**</u>:

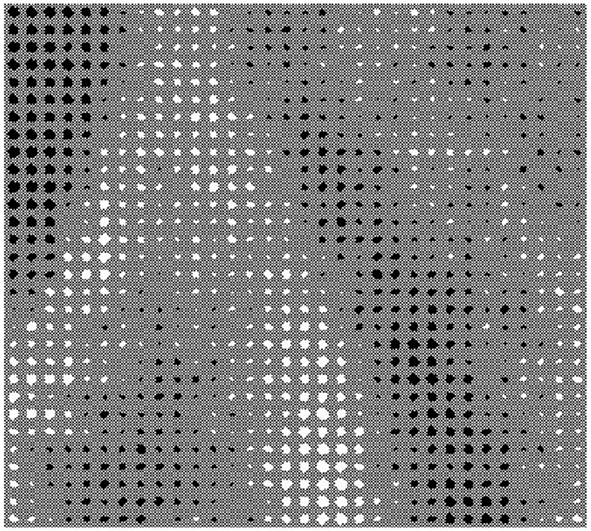
$$w_{kj} \to w_{kj} - \varepsilon \frac{dE}{dw_{kj}} \quad \text{But}: \frac{dE}{dw_{kj}} = \frac{dE}{dx_j} \cdot \frac{dx_j}{dw_{kj}} \quad \{\textit{chain rule}\}$$

$$\frac{dE}{dw_{kj}} = \left[-\sum_{m,i} (d_i^m - v_i^m) g'(\sum_j W_{ji} x_j^m) W_{ji} \right] \cdot \left[g'(\sum_k w_{kj} u_k^m) u_k^m \right]$$

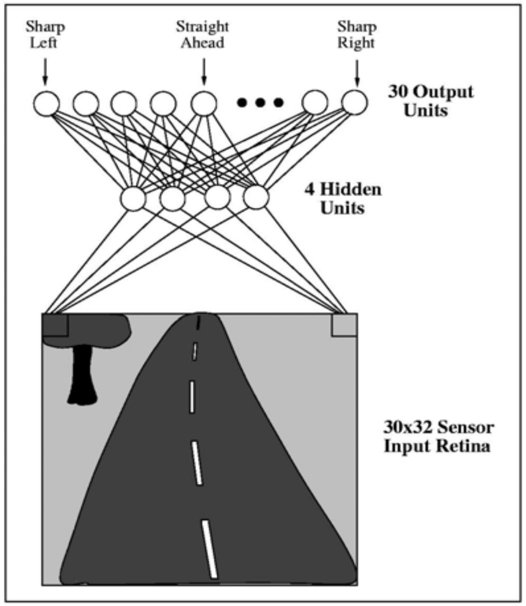
Learning to Drive using Backprop



One of the learned "road features" w_i



ALVINN (Autonomous Land Vehicle in a Neural Network)



CMU Navlab

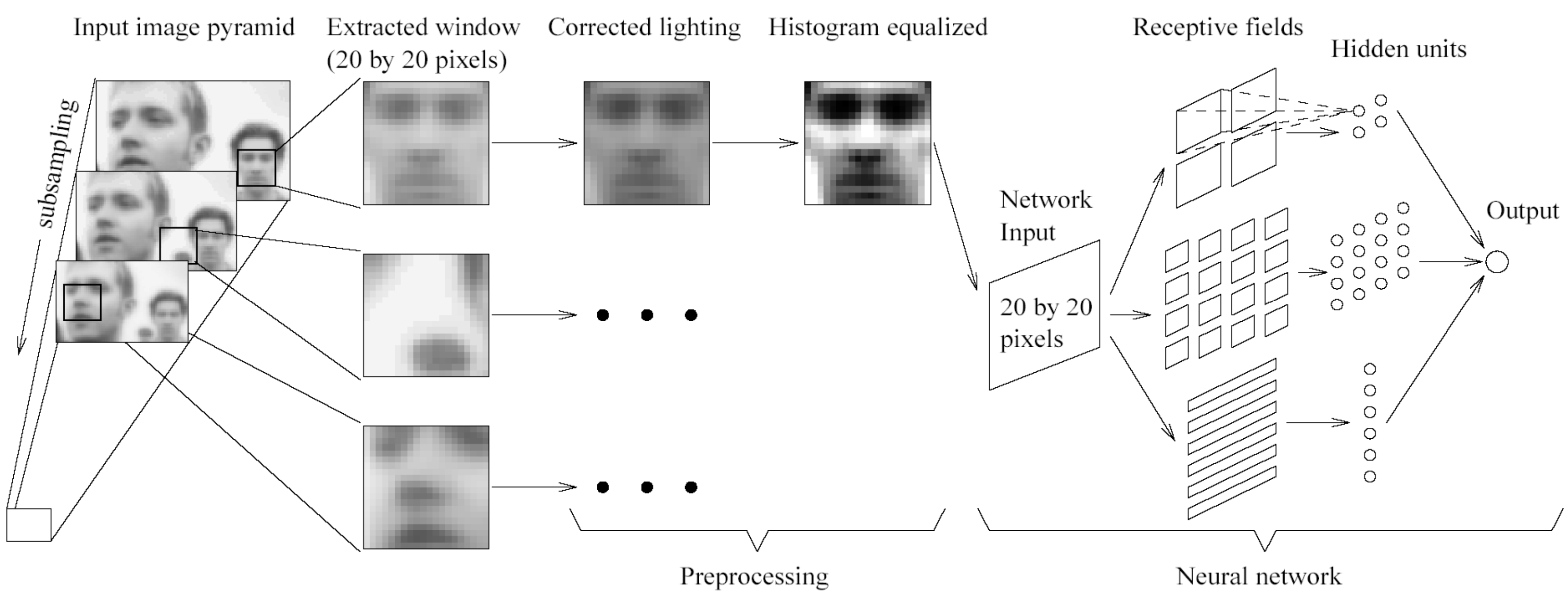
Trained using human driver + camera images

After learning:

- Drove up to 70 mph on highway
- Up to 22 miles without intervention
- Drove cross-country largely autonomously

(Pomerleau, 1992)

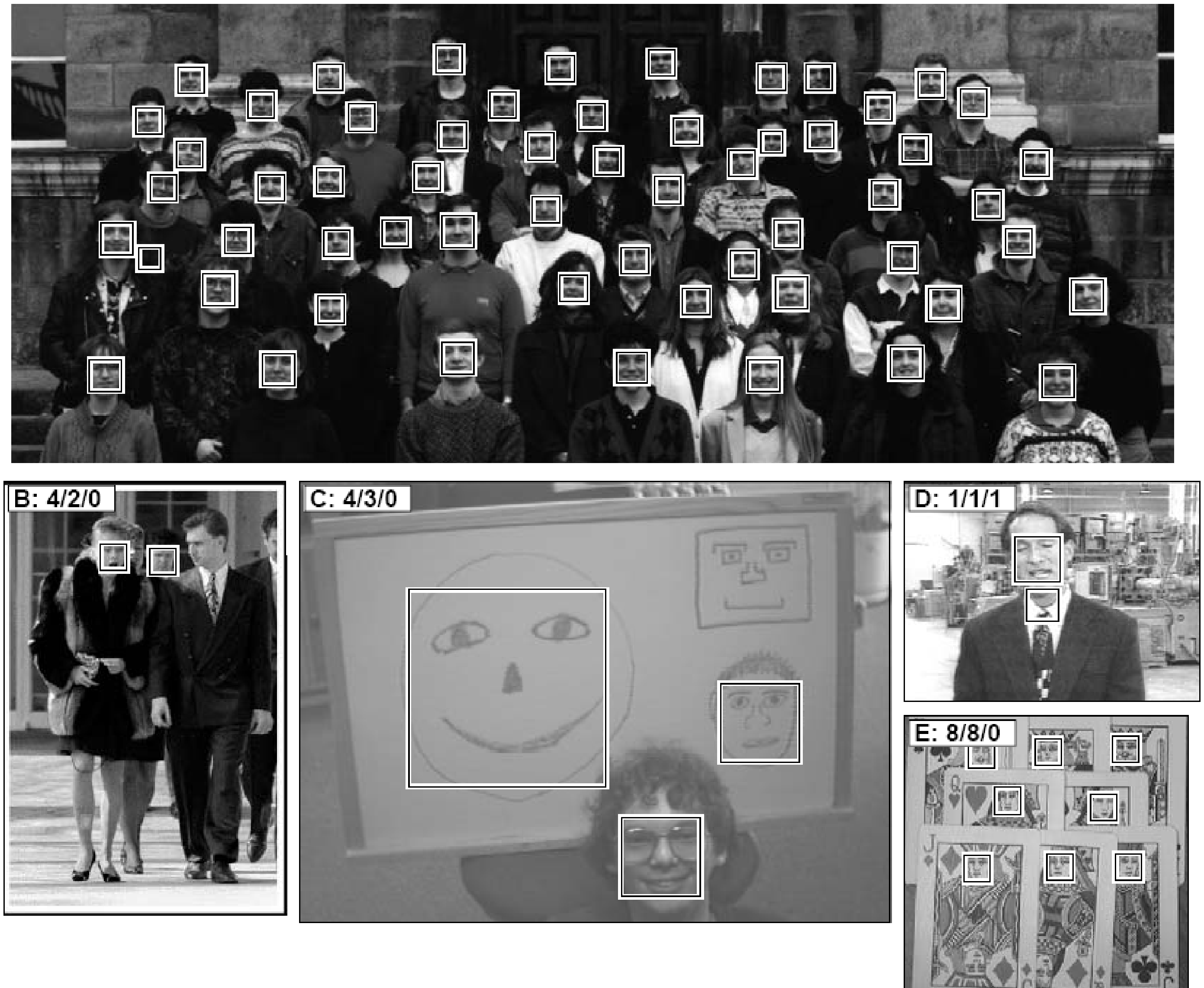
Another Example: Face Detection



Output between -1 (no face) and +1 (face present)

(Rowley, Baluja & Kanade, 1998)

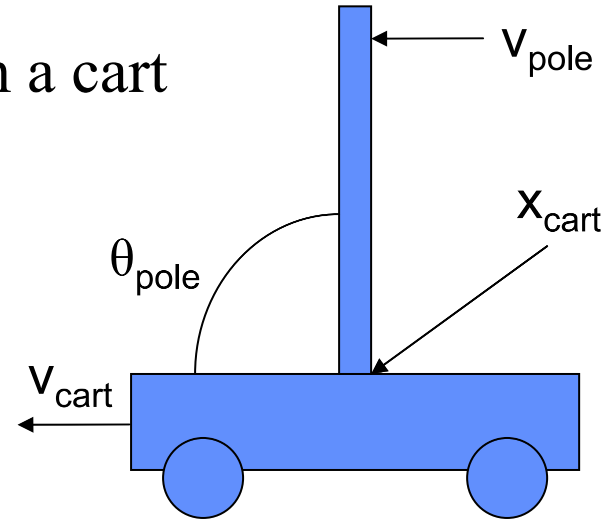
Face Detection Results



(Rowley, Baluja & Kanade, 1998)

Demos: Pole Balancing and Backing up a Truck

(courtesy of Keith Grochow, CSE 599)



- Neural network learns to balance a pole on a cart
  - System:
    - 4 state variables: x_{cart}, v_{cart}, θ_{pole}, v_{pole}
    - 1 input: Force on cart
  - Backprop Network:
    - Input: State variables
    - Output: New force on cart
- NN learns to back a truck into a loading dock
  - System (Nyugen and Widrow, 1989):
    - State variables: x_{cab}, y_{cab}, θ_{cab}
    - 1 input: new $\theta_{steering}$
  - Backprop Network:
    - Input: State variables
    - Output: Steering angle $\theta_{steering}$