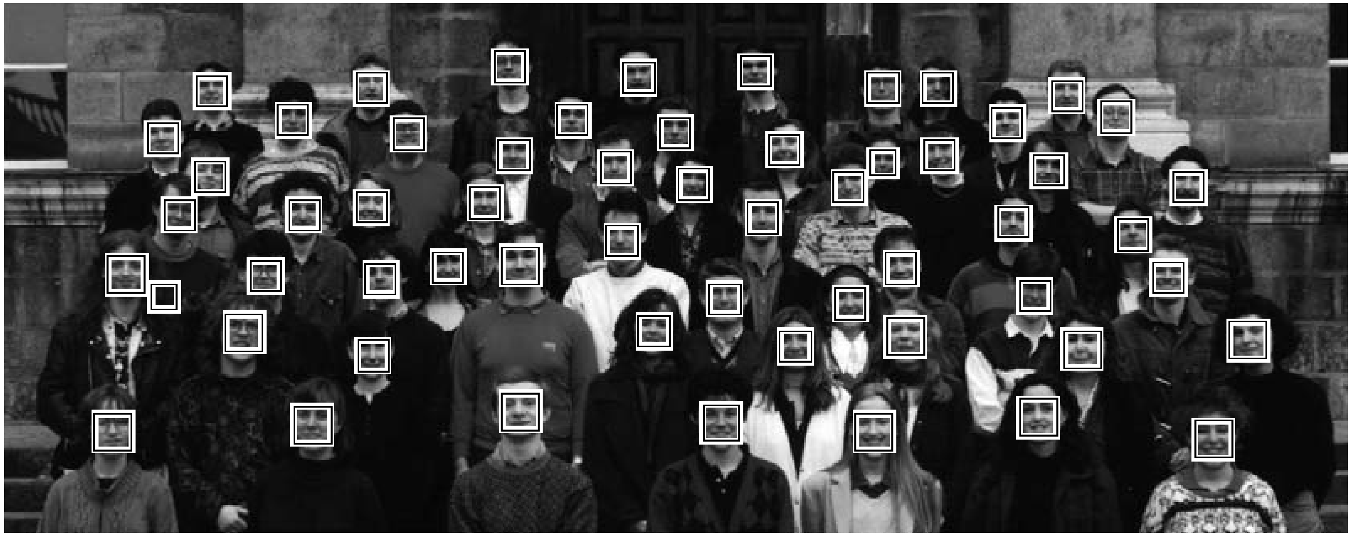
Lecture 9

Pattern Recognition & Learning

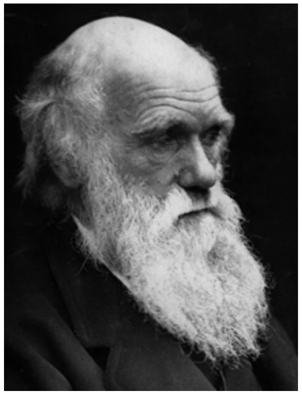
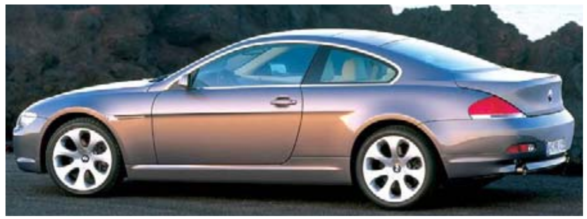


(Rowley, Baluja & Kanade, 1998)

© UW CSE vision faculty

Motivation: Object Classification

Suppose you are given a dataset of images
containing 2 classes of objects

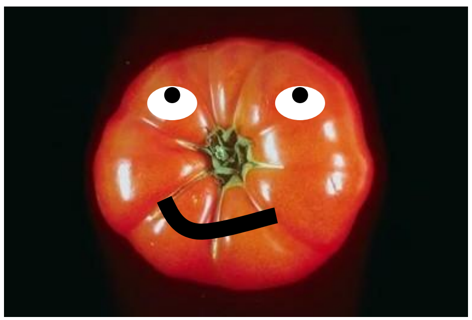


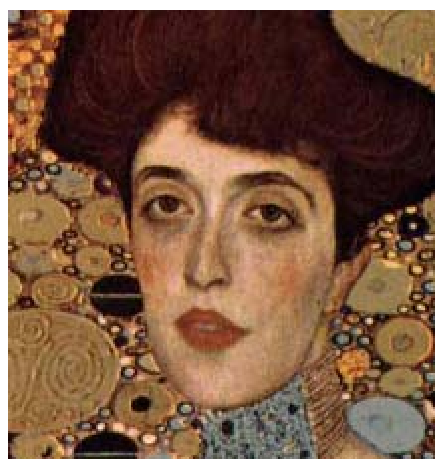





Test Set of Images

Can a computer vision system learn to automatically classify these new images?





Images as Patterns

Binary handwritten characters

```
00000000010000000000      00000000001111000000
00000000110000000000      00000001100001100000
00000000101000000000      00000011000000110000
00000001000010000000      00001100000000011000
00000010000010000000      00001000000000001000
00000100000001000000      00001100000000010000
00001000000000100000      00000111000000100000
00001100111111110000      00000011100111100000
00001111110000010000      00000000011110000000
00011000000000011000      00000011000111000000
00010000000000001100      00001100000000110000
00110000000000000100      00011000000000011000
00110000000000000110      00110000000000001000
00100000000000000010      00100000000000001100
00100000000000000010      00010000000000011000
01100000000000000010      00011000000000010000
01000000000000000000      00001000000000110000
00000000000000000000      00000011111110000000
```

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

.....

⋮

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

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

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

Feature representation

- Trying to classify raw images directly may be
  - inefficient (huge number of pixels N)
  - error-prone (raw pixel values not invariant to transformations)
- Better to extract features from the image and use these for classification
- Represent each image I by a vector of features:

$$\mathbf{F}_{\mathbf{I}} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}$$

n is typically much smaller than N (though doesn't have to be)

Types of Features: Binary Images

- Features for binary characters ('A', 'B', 'C', ..) could be number of strokes, number of holes, area, etc.

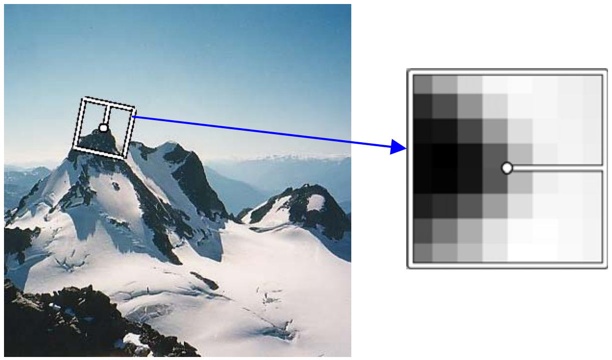
```
00000000010000000000
00000000110000000000
00000000101000000000
00000001000010000000
00000010000010000000
00000100000001000000
00001000000000100000
00001100111111110000
00001111110000010000
00011000000000011000
00010000000000001100
00110000000000000100
00110000000000000110
00100000000000000010
00100000000000000010
01100000000000000010
01000000000000000000
00000000000000000000
```

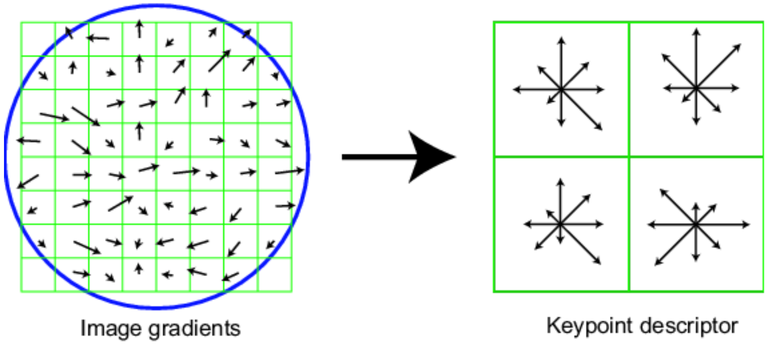
```
00000000011110000000
00000001100001100000
00000011000000110000
00001100000000011000
00001000000000001000
00001100000000010000
00000111000000100000
00000011100111100000
00000000111100000000
00000011000111000000
00001100000000110000
00011000000000011000
00110000000000001000
00100000000000001100
00010000000000011000
00011000000000010000
00001000000000110000
00000011111110000000
```

(class) character	area	height	width	number #holes	number #strokes	(cx,cy) center	best axis
'A'	medium	high	3/4	1	3	1/2,2/3	90
'B'	medium	high	3/4	2	1	1/3,1/2	90
'8'	medium	high	2/3	2	0	1/2,1/2	90
'0'	medium	high	2/3	1	0	1/2,1/2	90
'1'	low	high	1/4	0	1	1/2,1/2	90
'W'	high	high	1	0	4	1/2,2/3	90
'X'	high	high	3/4	0	2	1/2,1/2	?
'*'	medium	low	1/2	0	0	1/2,1/2	?
'-'	low	low	2/3	0	1	1/2,1/2	0
'/'	low	high	2/3	0	1	1/2,1/2	60

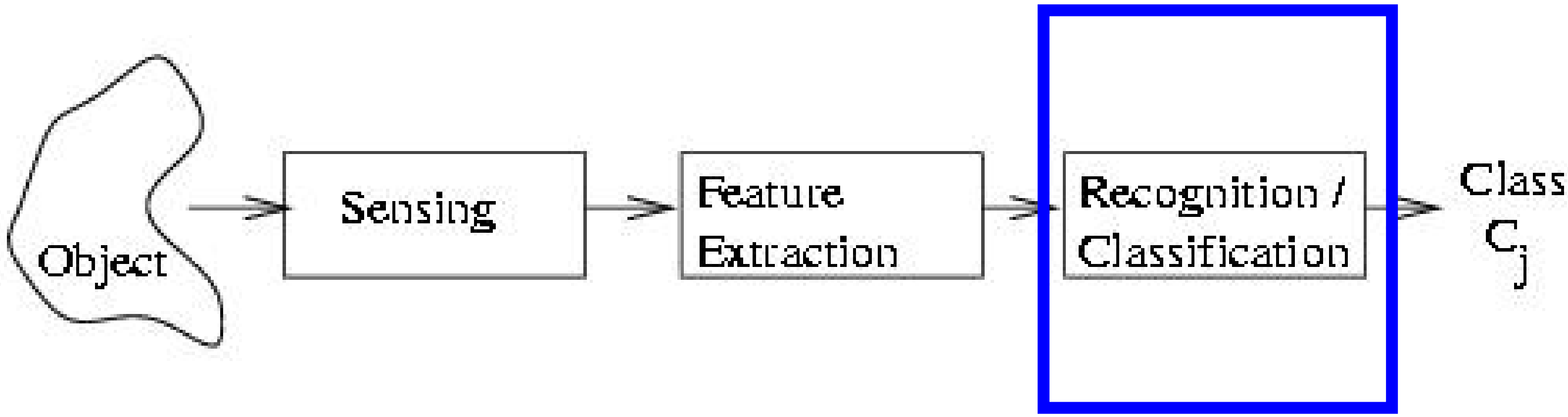
Types of Features: Grayscale and Color

- Features for greyscale images could be oriented gradient features, multiscale oriented patches (MOPS), SIFT features, etc.
- Features for color images could be above features applied to R, G, B images, or opponent images (R-G image, B-(R+G)/2 image)





Typical Pattern Recognition System



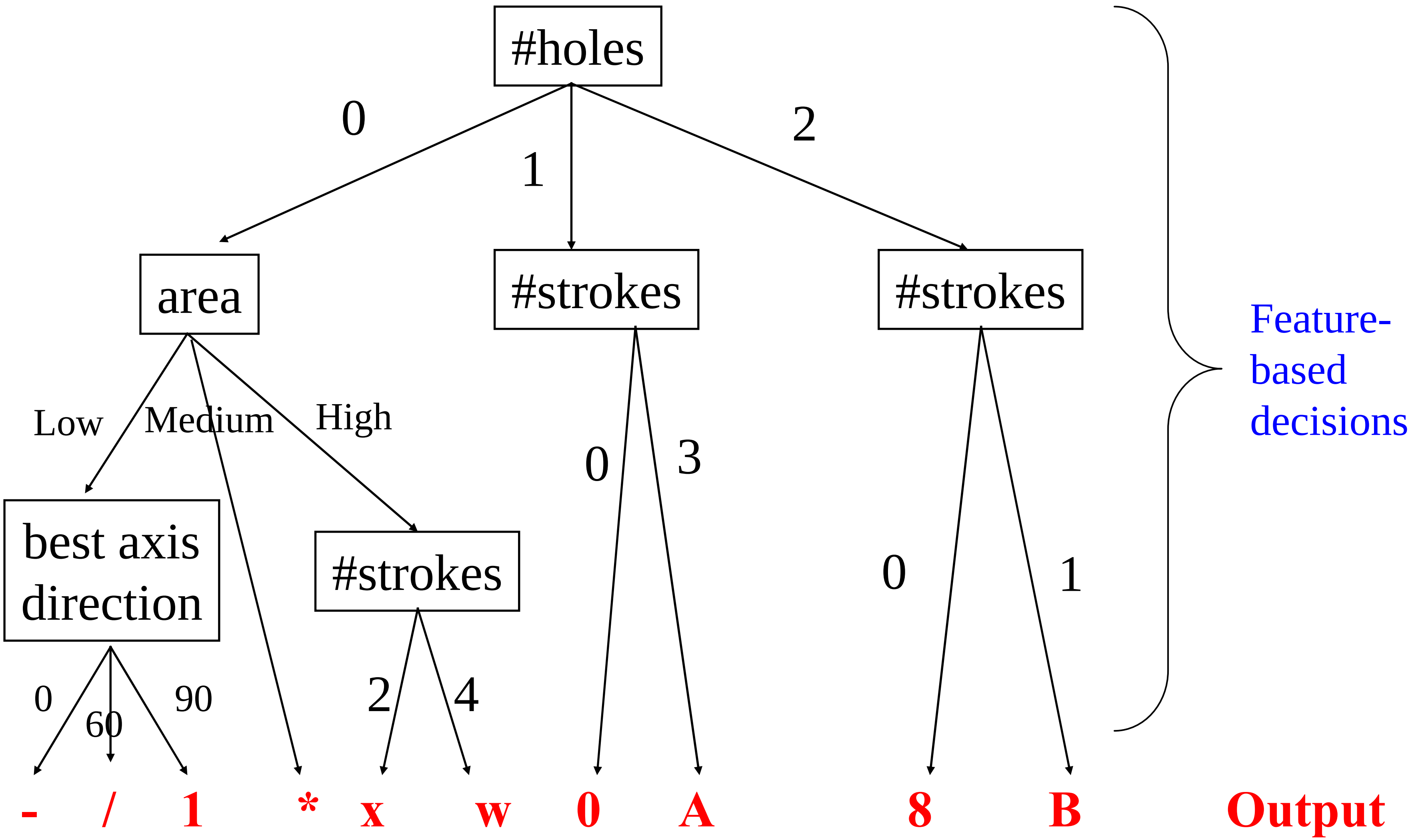
Pattern recognition or classification problem: Given a training dataset of (input image, output class) pairs, build a classifier that outputs a class for any new input image

Example: Dataset of Binary Character Images

Feature values extracted from input image **Class**

area	height	width	number #holes	number #strokes	(cx,cy) center	best axis	Class
medium	high	3/4	1	3	1/2,2/3	90	'A'
medium	high	3/4	2	1	1/3,1/2	90	'B'
medium	high	2/3	2	0	1/2,1/2	90	'8'
medium	high	2/3	1	0	1/2,1/2	90	'0'
low	high	1/4	0	1	1/2,1/2	90	'1'
high	high	1	0	4	1/2,2/3	90	'W'
high	high	3/4	0	2	1/2,1/2	?	'X'
medium	low	1/2	0	0	1/2,1/2	?	'*'
low	low	2/3	0	1	1/2,1/2	0	'-'
low	high	2/3	0	1	1/2,1/2	60	'/'

Decision Tree



Decision Trees

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

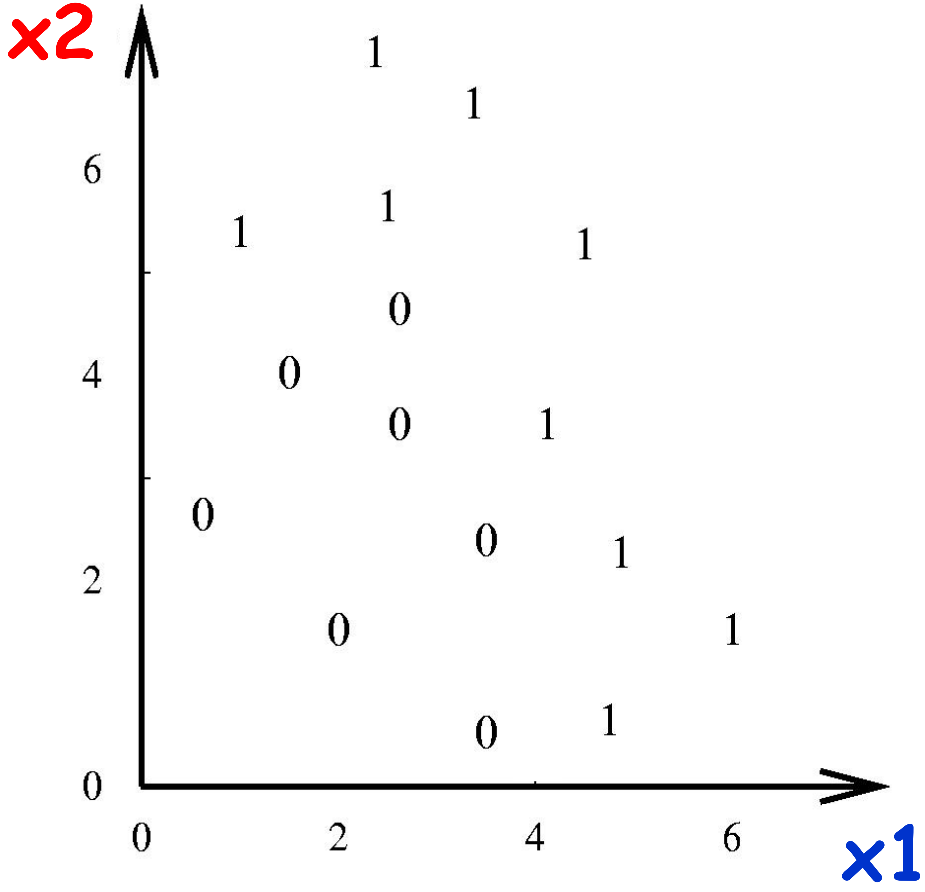
Output: a **decision** that is the predicted output value for the input

Advantages:

- Not all features need be evaluated for every input
- Feature extraction may be interleaved with classification decisions
- Can be easy to design and efficient in execution

Feature values **can be discrete or continuous**

Example: Decision Tree for Continuous Valued Features

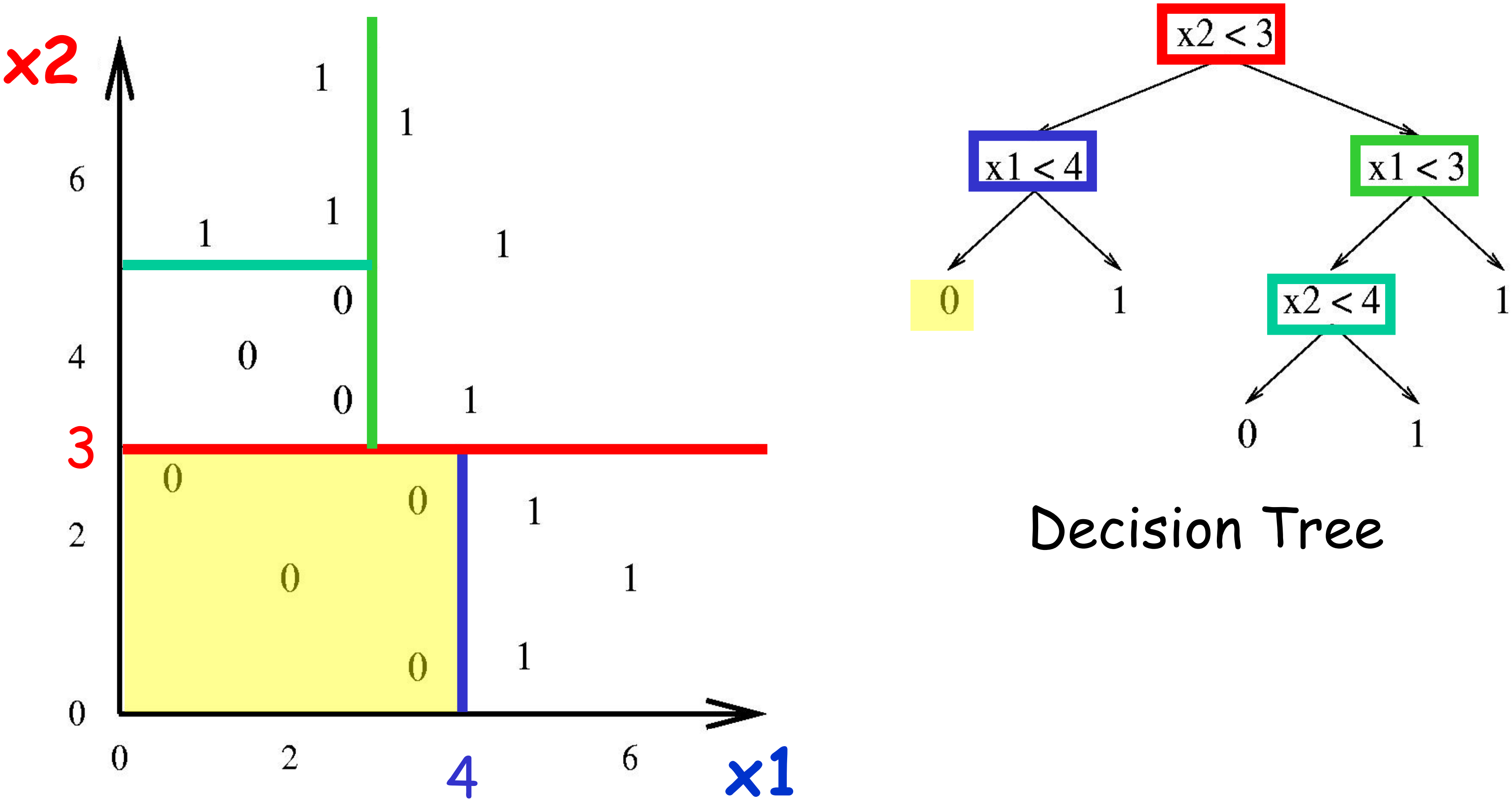


Two features x1 and x2
Two output classes 0 and 1

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

Example: Decision Tree for Continuous Valued Features

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

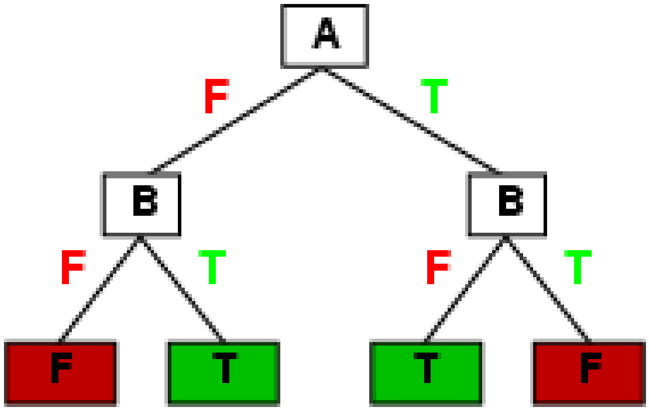


Expressiveness

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

Want to find more compact decision trees (to prevent overfitting and allow generalization)

Decision Tree Learning

Aim: find a small tree consistent with training examples

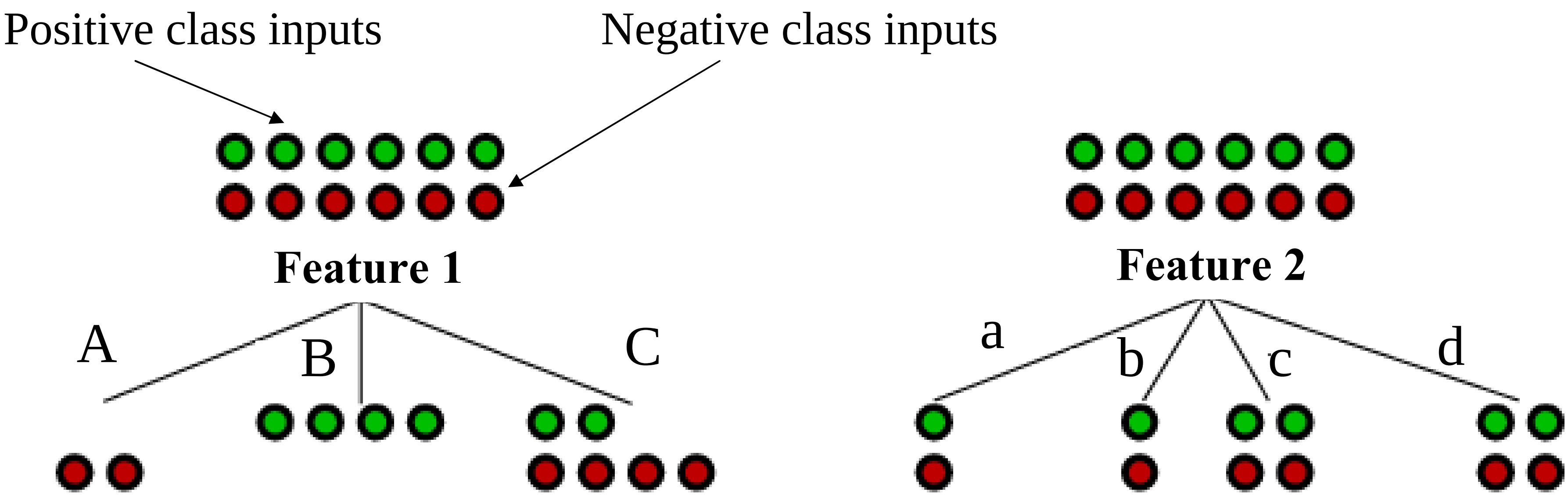
Idea: (recursively) choose "most significant" attribute (feature) as root of (sub)tree and expand

```
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/feature to split on

Idea: a good feature should reduce uncertainty

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



Feature 1 is a better choice

Output class probability
is still at 50%.

How do we quantify uncertainty?

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 $v_1, \ldots, v_n$:

$$I(P(v_1), \ldots, P(v_n)) = - \Sigma_i P(v_i) \log_2 P(v_i)$$

Using information theory

Imagine we have p examples with Feature1 = 1 or true, and n examples with Feature1 = 0 or false.

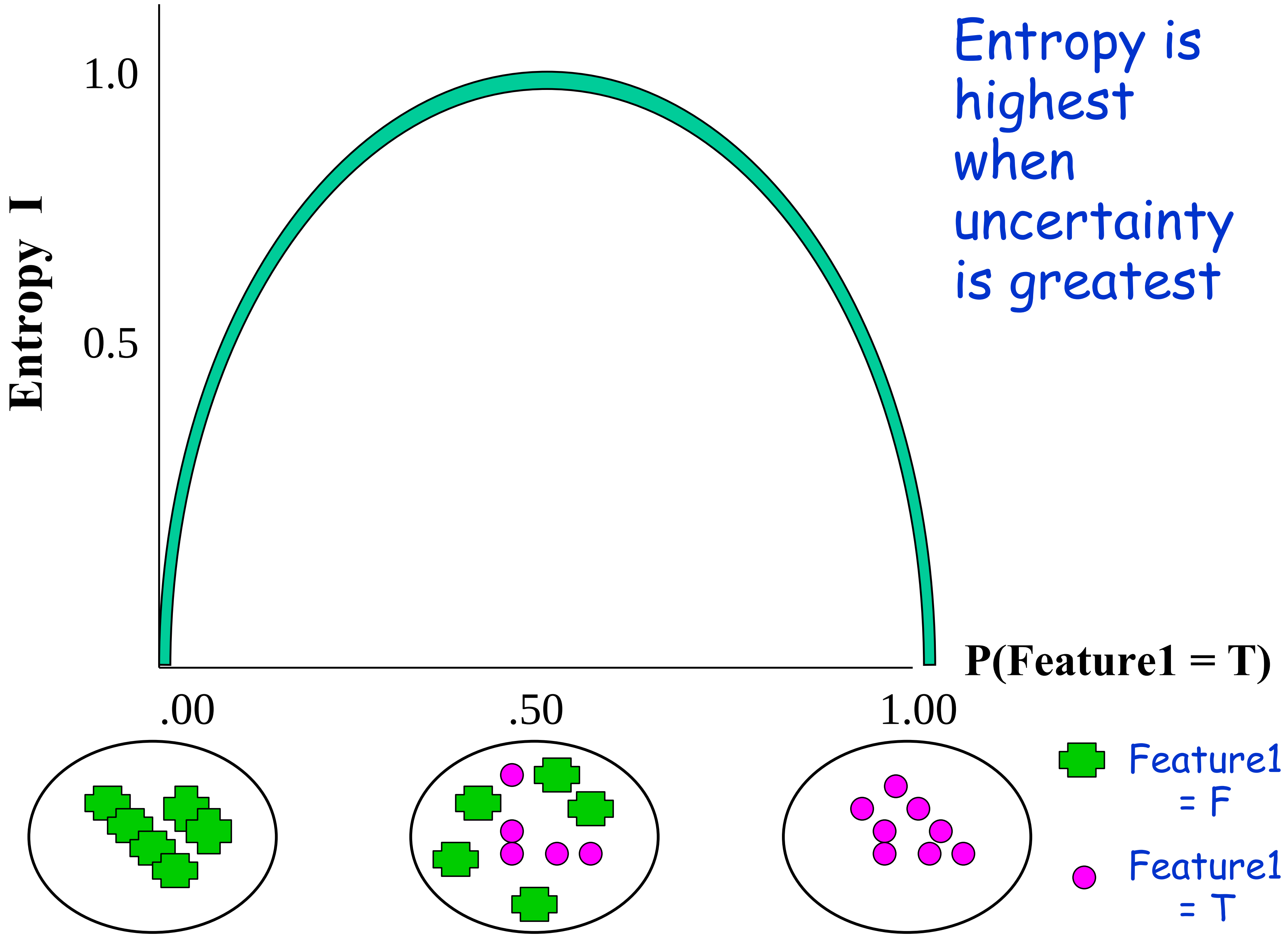
Our best estimate of the probabilities of Feature1 = true or false is given by:

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

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

Hence the entropy of Feature1 is given by:

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


Entropy is highest when uncertainty is greatest
Entropy I
1.0
0.5
P(Feature1 = T)
.00
.50
1.00
Feature1 = F
Feature1 = T

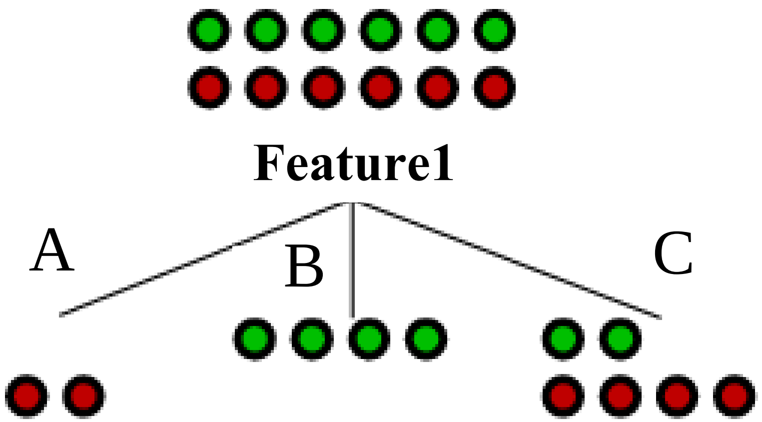
Choosing an attribute/feature to split on

Idea: a good feature should reduce uncertainty and result in “gain in information”

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

Answer: uncertainty before – uncertainty after

Example



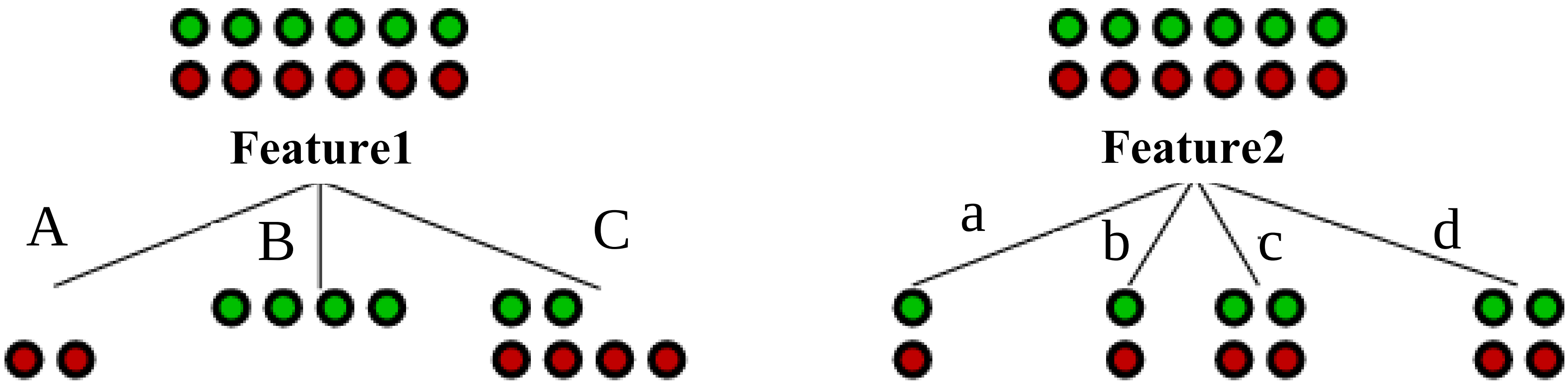
Before choosing any feature:

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"

Example



If we choose Feature2: Go along branch "a": we have entropy = 1 bit; similarly for the others.

Information gain = 1-1 = 0 along any branch

If we choose Feature1:

In branch "A" and "B", entropy = 0

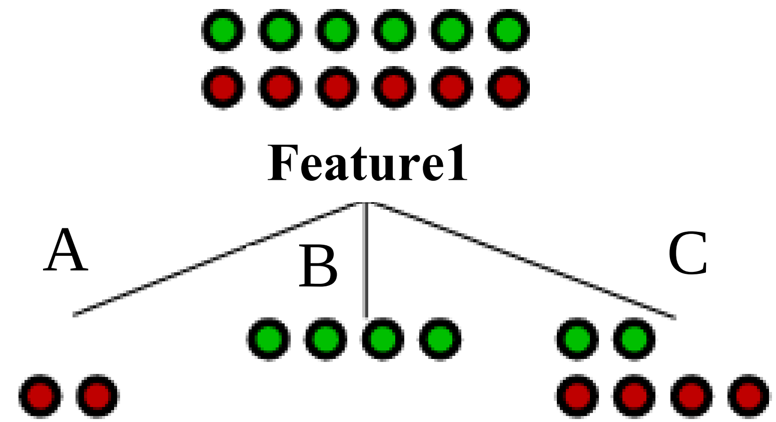
For "C", entropy = -2/6 log(2/6)-4/6 log(4/6) = 0.92

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

So choosing Feature1 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 "A", so weight for "A" = 1/6
  - "B" has weight: 4/12 = 1/3
  - "C" has weight 6/12 = ½



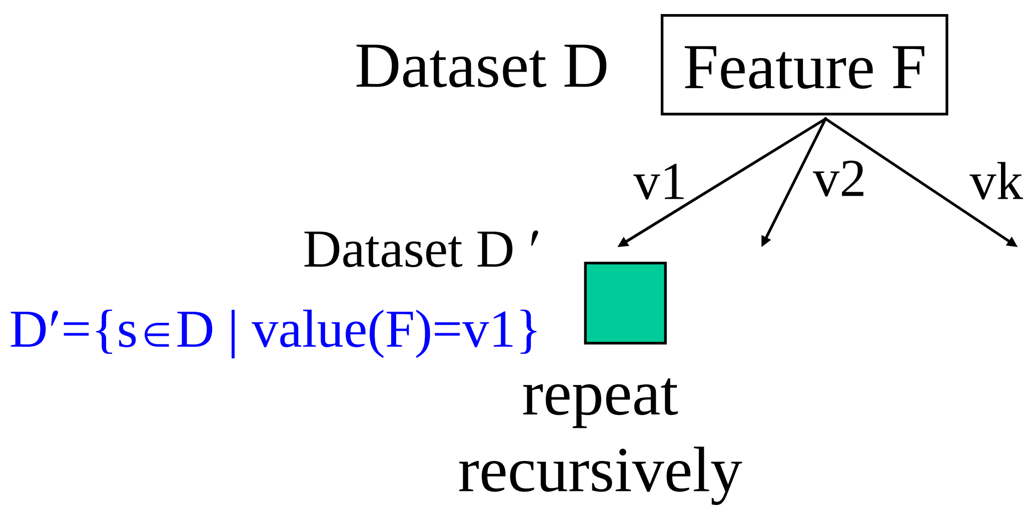
$$AvgEntropy = \sum_{i=1}^{m} \underbrace{\frac{p_i + n_i}{p + n}}_{\text{weight for each branch}} \underbrace{Entropy\left(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}\right)}_{\text{entropy for each branch}}$$

Information gain

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

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

1. Choose the feature/attribute with the largest IG
2. Create (sub)tree with this feature as root
3. Recursively call the algorithm for each value of feature

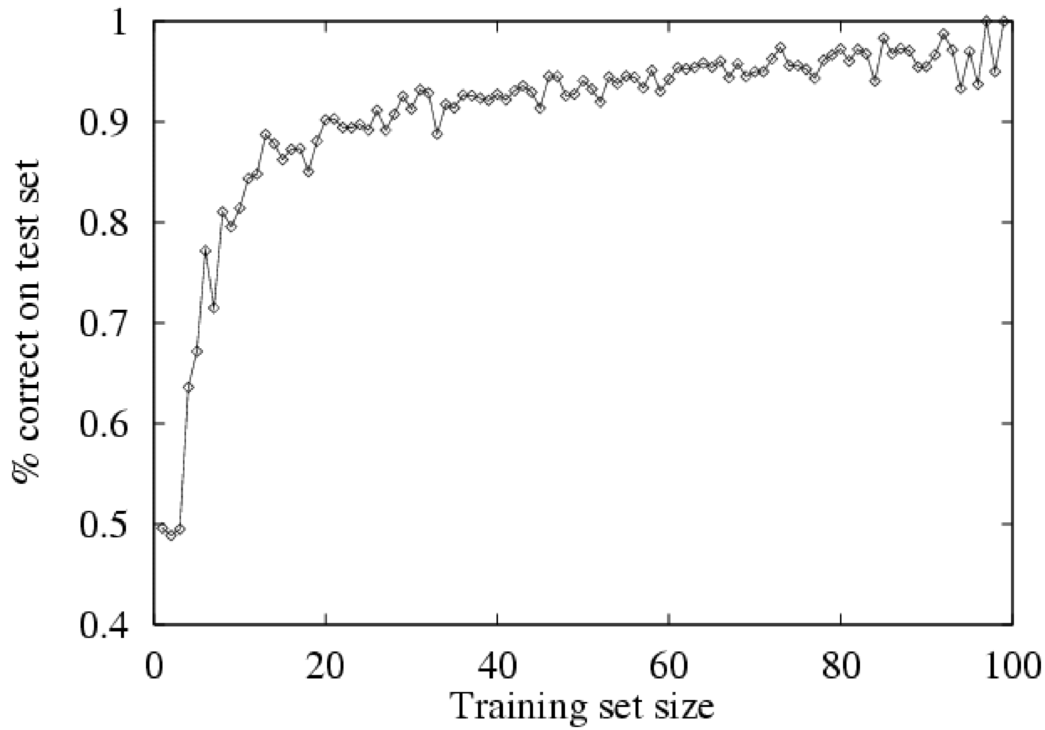


Performance Measurement

How do we test the performance of the learned tree?

Answer: Try it on a test set of examples not used in training

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



Cross-validation

Instead of only 1 subset held out as the test set, better to use 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

Confusion matrix

class j output by the pattern recognition system

		’0’	’1’	’2’	’3’	’4’	’5’	’6’	’7’	’8’	’9’	’R’
	’0’	97	0	0	0	0	0	1	0	0	1	1
	’1’	0	98	0	0	1	0	0	1	0	0	0
true	’2’	0	0	96	1	0	1	0	1	0	0	1
object	’3’	0	0	2	95	0	1	0	0	1	0	1
class	’4’	0	0	0	0	98	0	0	0	0	2	0
	’5’	0	0	0	1	0	97	0	0	0	0	2
i	’6’	1	0	0	0	0	1	98	0	0	0	0
	’7’	0	0	1	0	0	0	0	98	0	0	1
	’8’	0	0	0	1	0	0	1	0	96	1	1
	’9’	1	0	0	0	3	0	0	0	1	95	0

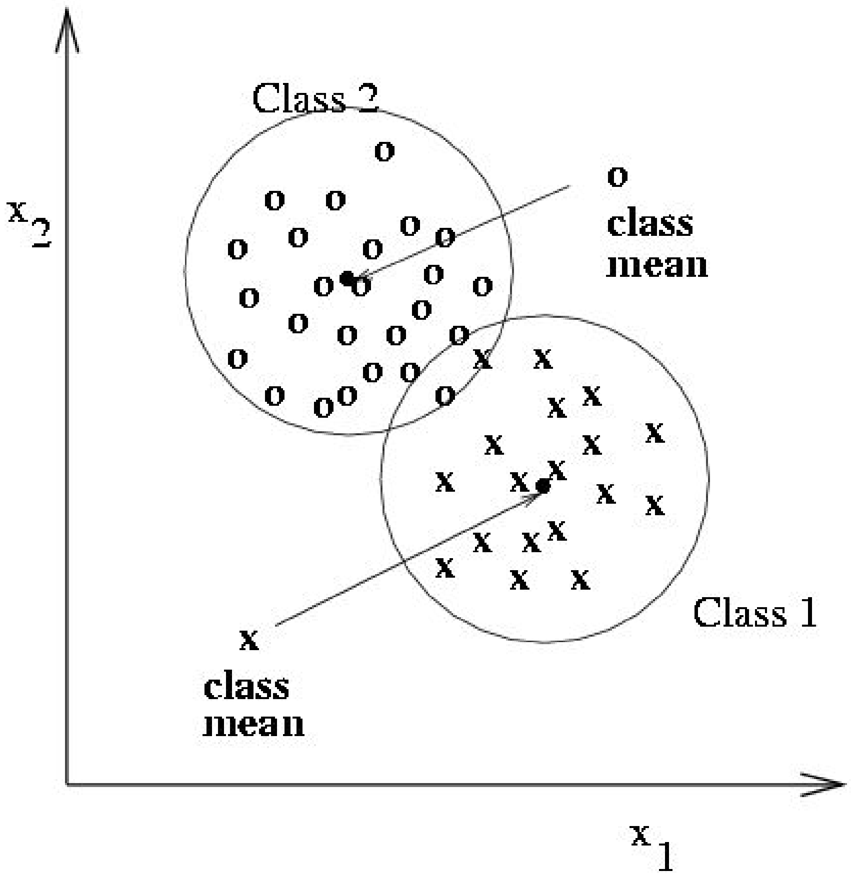
Useful for characterizing recognition performace

Quantifies amount of “confusion” between similar classes

Other classification methods

These utilize the full feature vector for each input

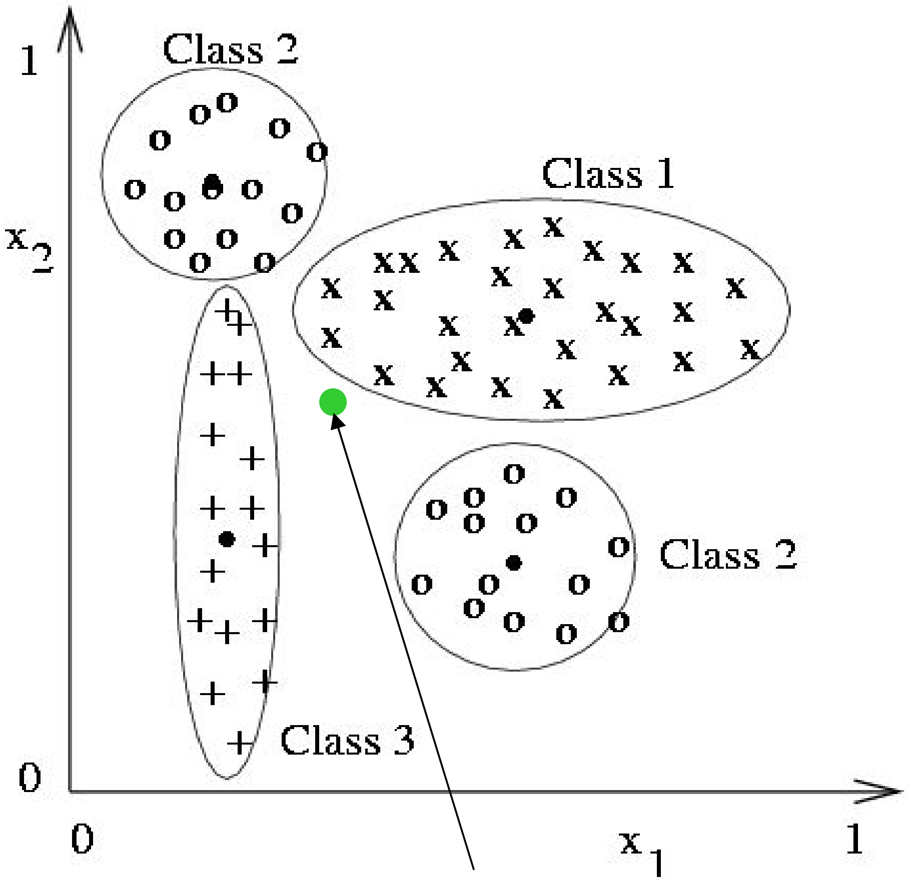
Classification using nearest class mean



Given new input image **I**, compute the distance (e.g., Euclidean distance) between feature vector $\mathbf{F_I}$ and the mean of each class

Choose closest class, if close enough (reject otherwise)

If the class distributions are complex…



New input point
What is its class?

Class 2 has two clusters

Where is its mean?

Nearest class mean method will likely fail badly in this case

Nearest Neighbor Classification

- Keep all the training samples in some efficient look-up structure
- Find the nearest neighbor of the feature vector to be classified and assign the class of the neighbor
- Can be extended to K nearest neighbors

K-Nearest Neighbors

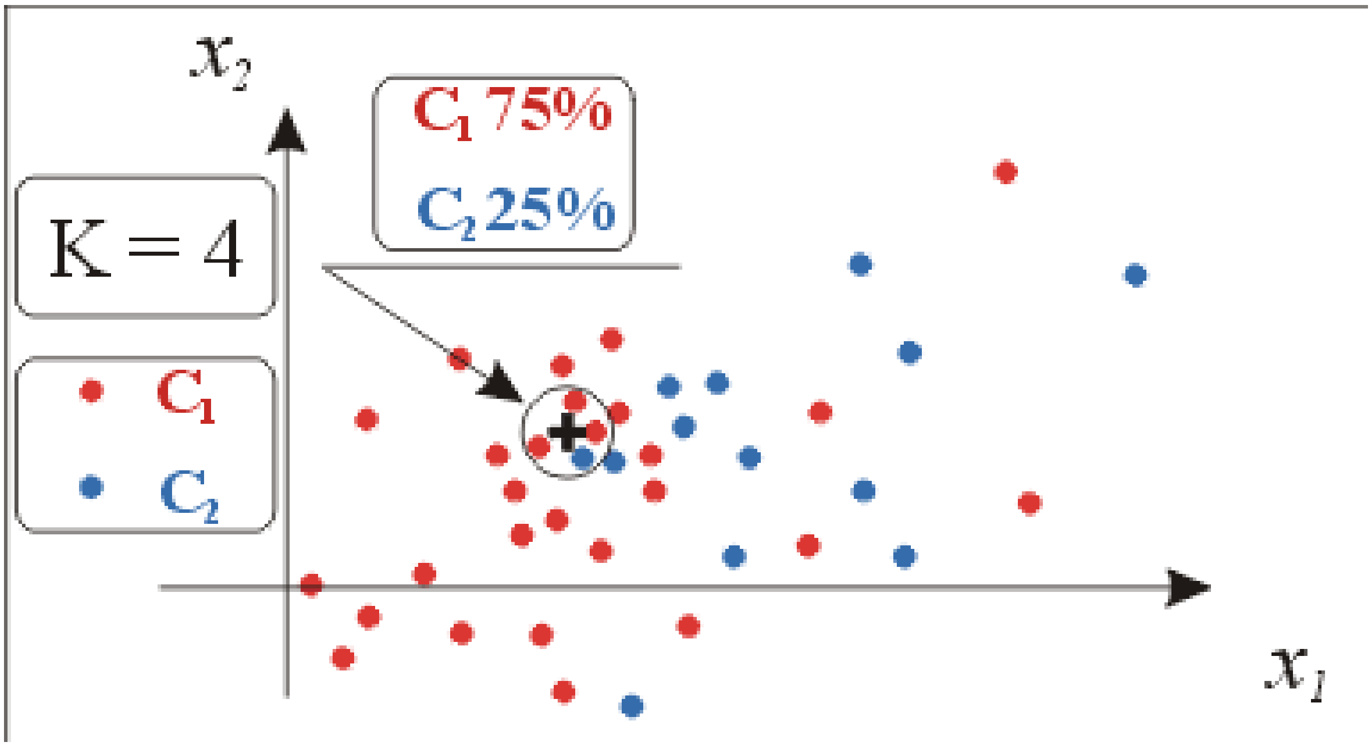
Idea:

- 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

Example

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

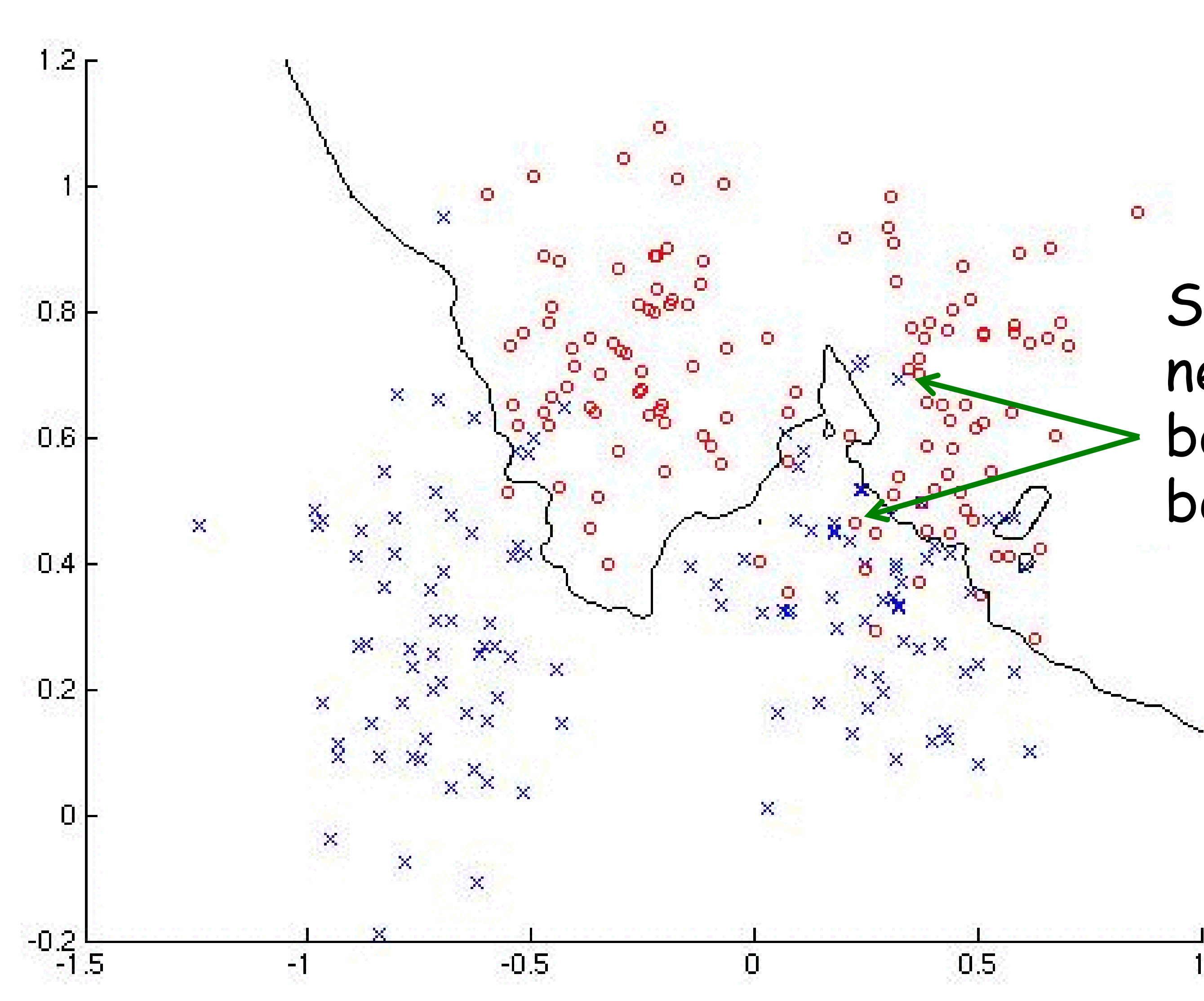
Two classes: C_1 and C_2. New Data Point +



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

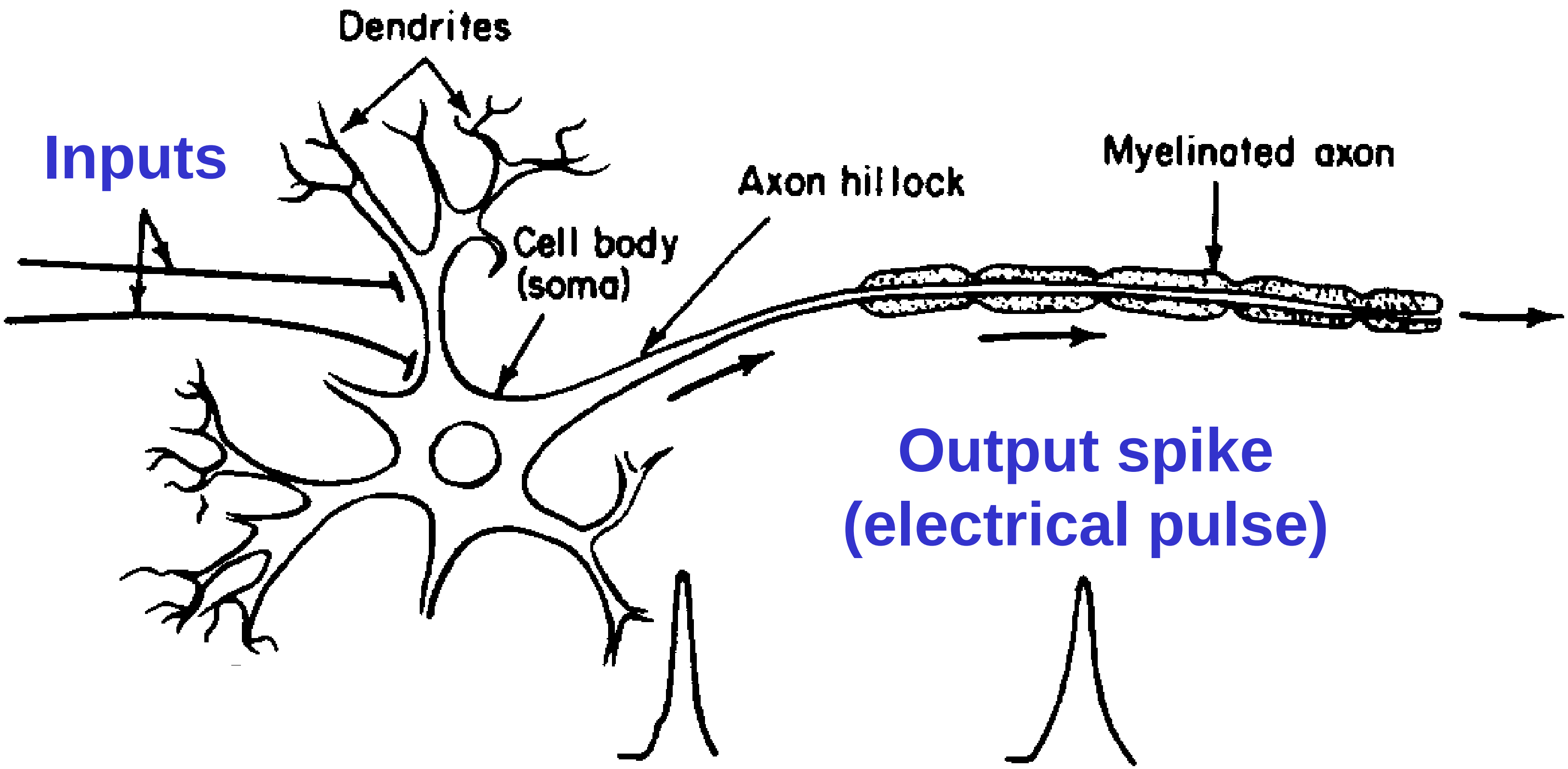
K-NN is for girlie men – what about something stronger?

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

The human brain is extremely good at classifying objects in images

Can we develop classification methods by emulating the brain?

Neurons compute using spikes



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

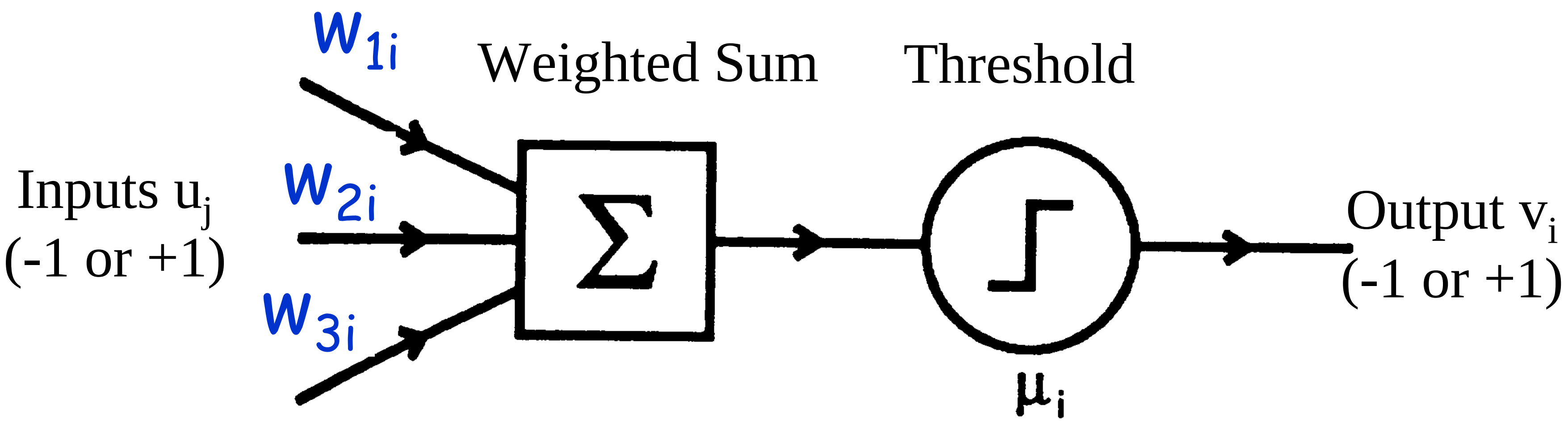
Neurons as "Threshold Units"

Artificial neuron:

- m binary inputs (-1 or 1) and 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 $x \leq 0$



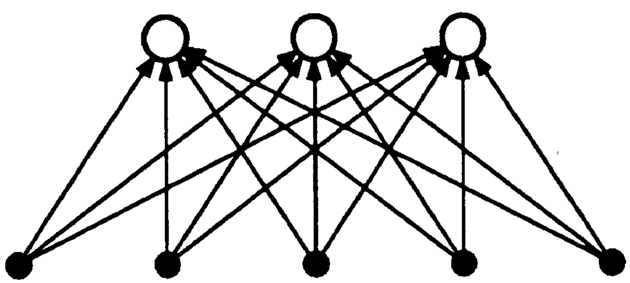
“Perceptrons” for Classification

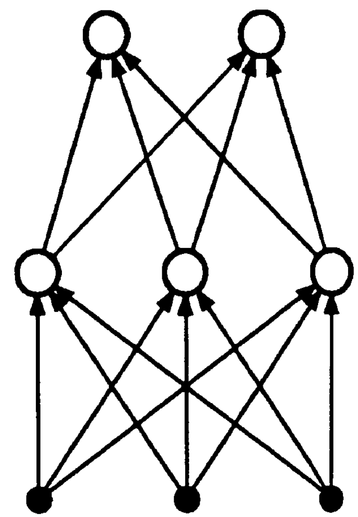
Fancy name for a type of layered “feed-forward” networks (no loops)

Uses artificial neurons (“units”) with binary inputs and outputs

Multilayer

Single-layer





Perceptrons and Classification

Consider a single-layer perceptron

- Weighted sum forms a *linear hyperplane*

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

- 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)

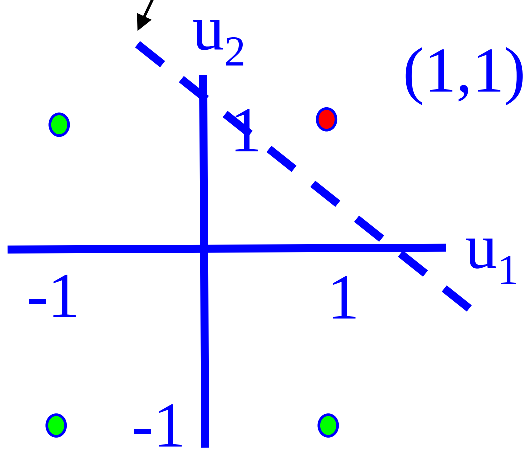
Any function that is linearly separable can be computed by a perceptron

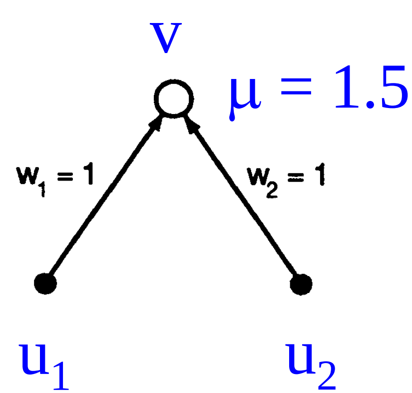
Linear Separability

Example: AND is linearly separable

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

A linear hyperplane
(a line in 2D)



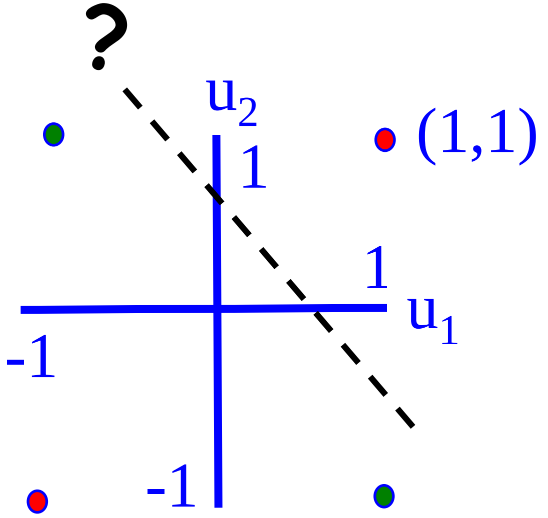


$v = 1$ iff $u_1 + u_2 - 1.5 > 0$

Similarly for OR and NOT

What about the XOR function?

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



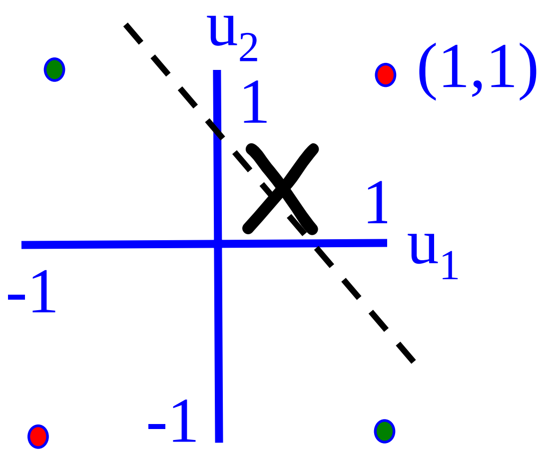
Can a straight line separate the +1 outputs from the -1 outputs?

Linear Inseparability

Single-layer 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!



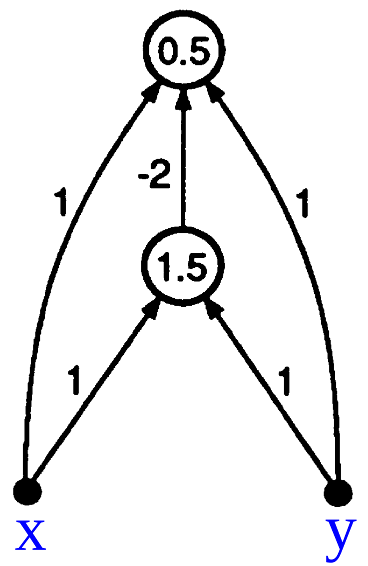
How do we deal with linear inseparability?

Multilayer Perceptrons

Removes limitations of single-layer networks

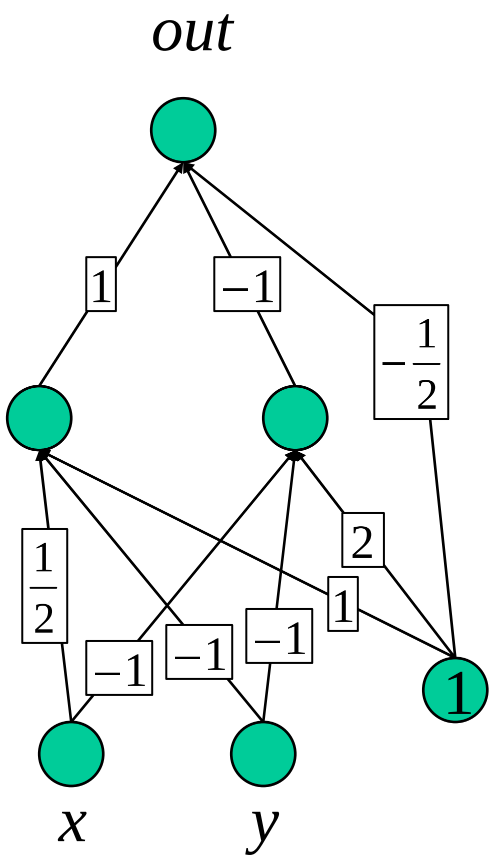
- Can solve XOR

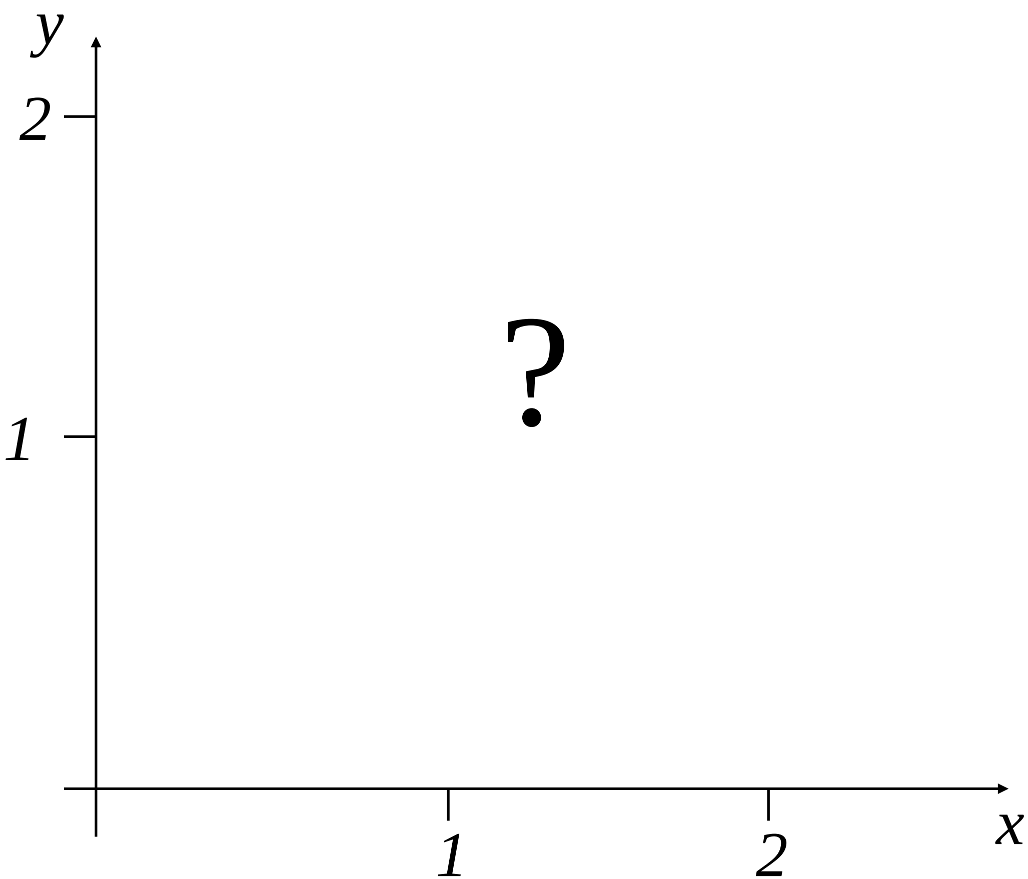
Example: Two-layer perceptron that computes XOR



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

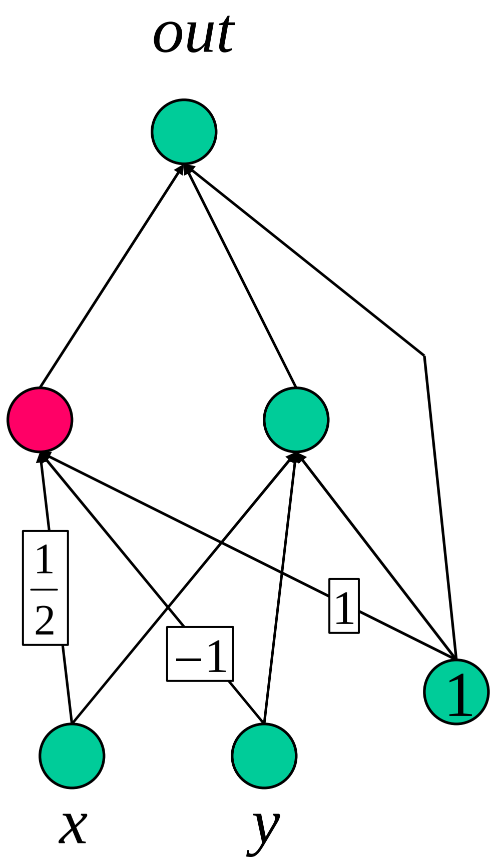
Multilayer Perceptron: What does it do?

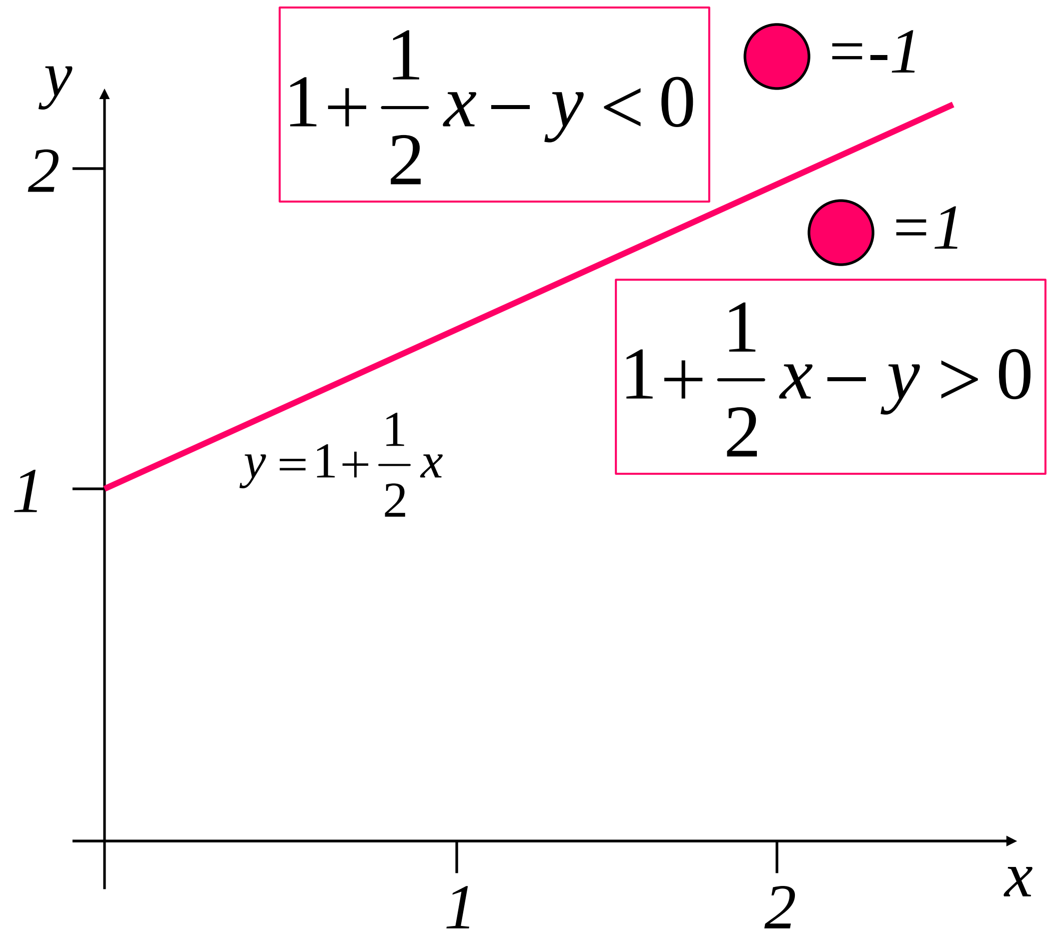




Example: Perceptrons as Constraint Satisfaction Networks

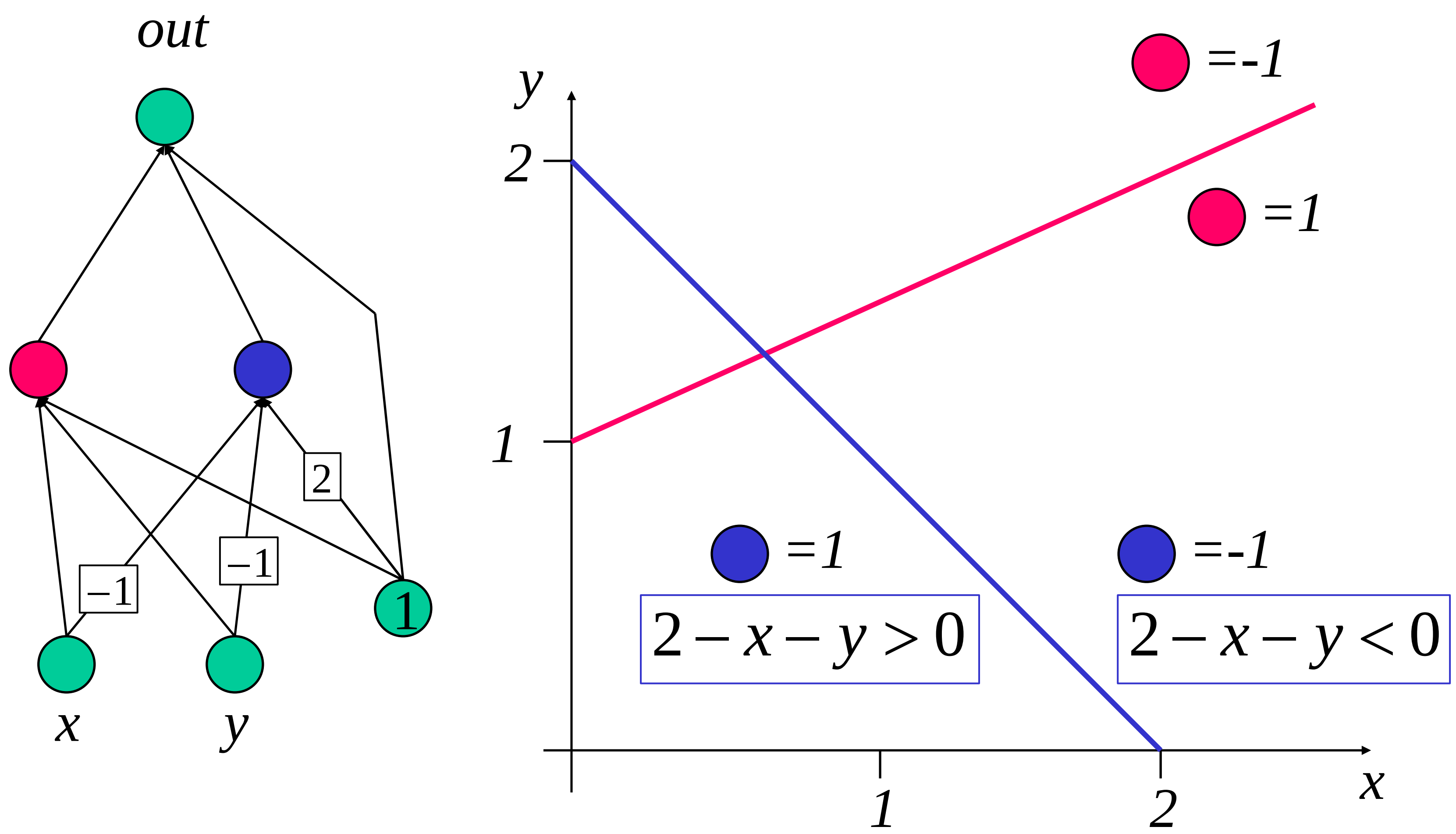
Line defined by first hidden unit





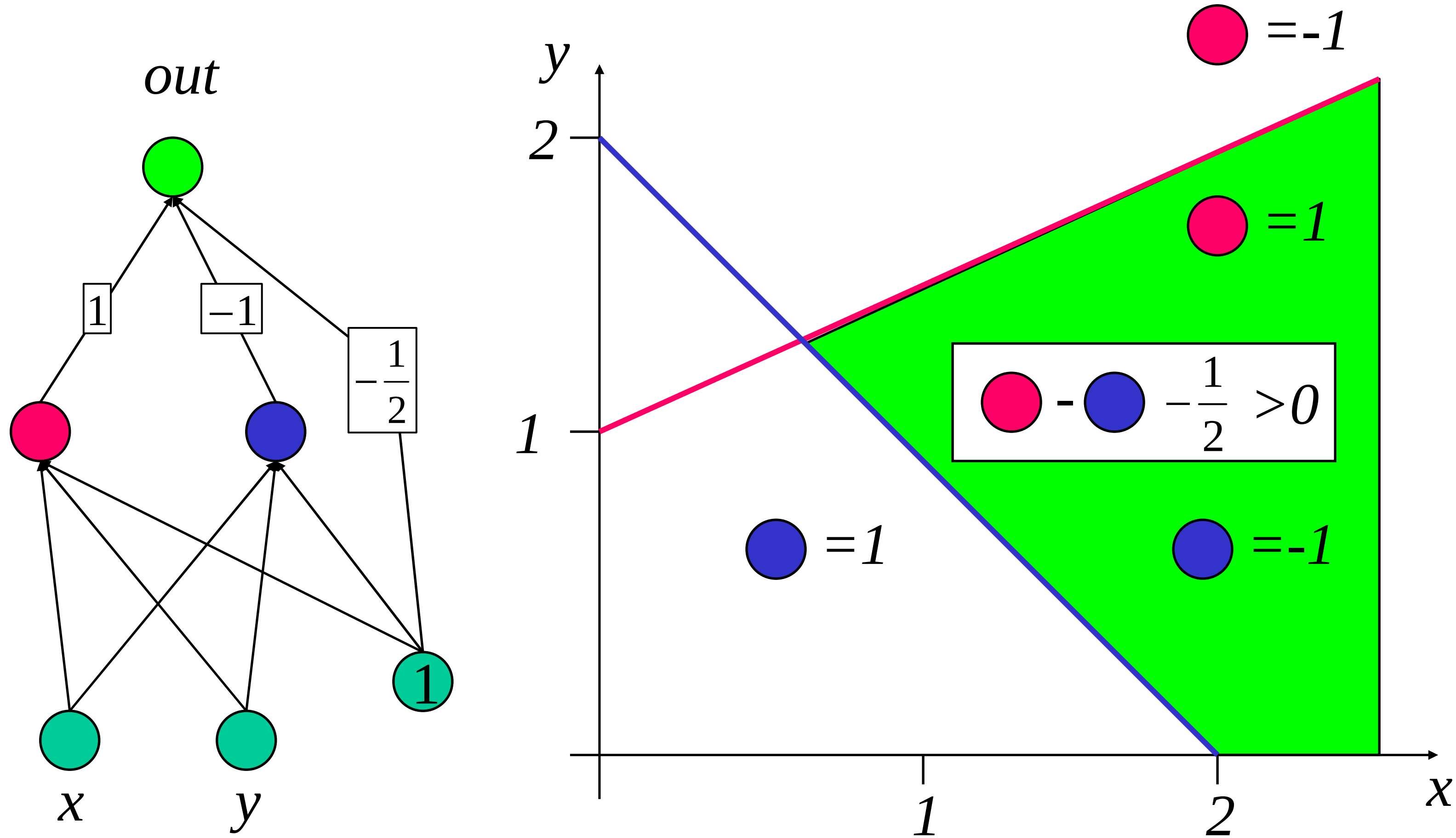
Example: Perceptrons as Constraint Satisfaction Networks

Line defined by second hidden unit



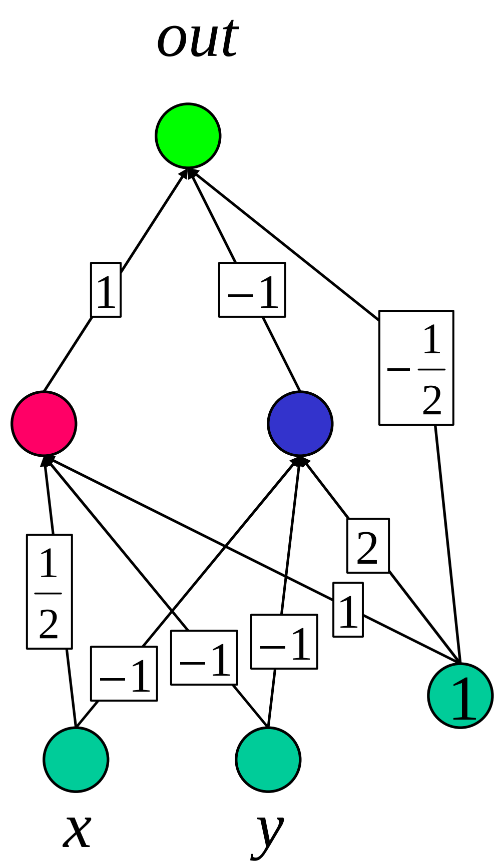
Example: Perceptrons as Constraint Satisfaction Networks

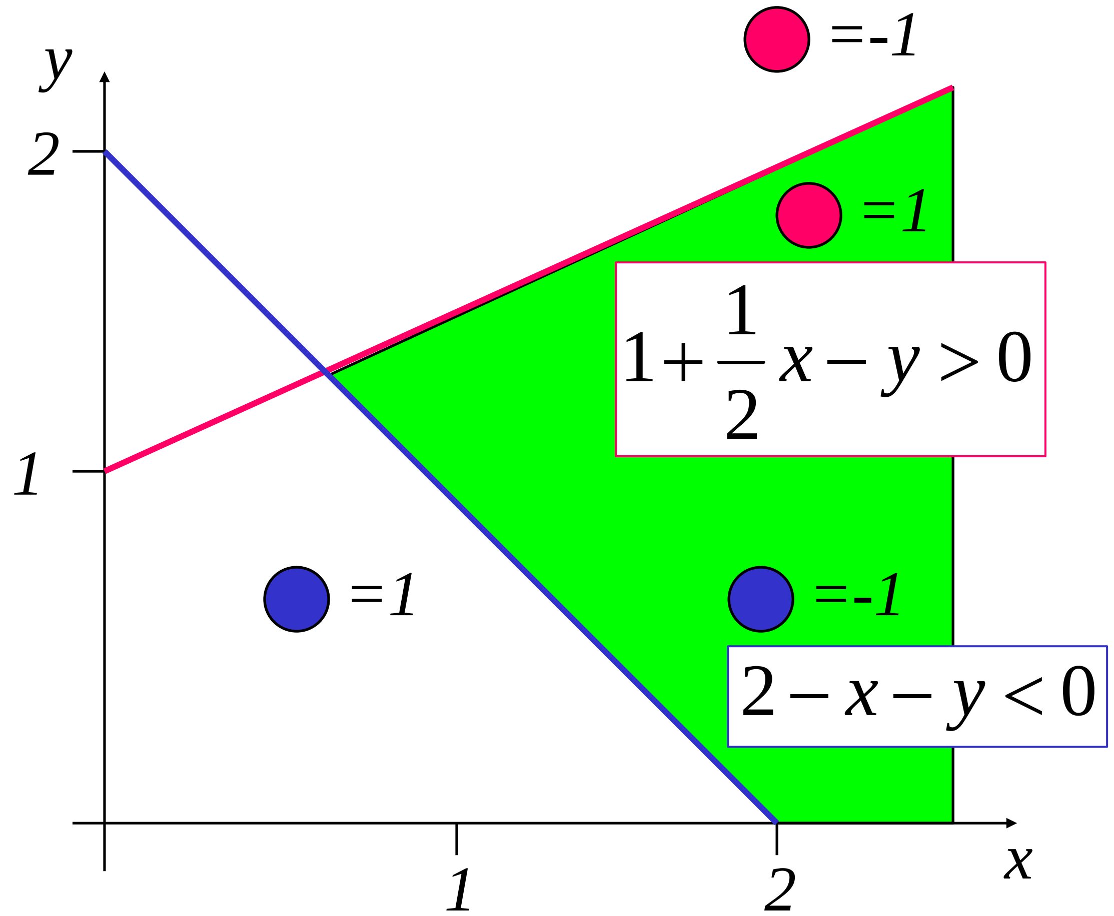
Output region defined by combining hidden unit outputs



Example: Perceptrons as Constraint Satisfaction Networks

Output is 1 if and only if inputs satisfy the two constraints





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

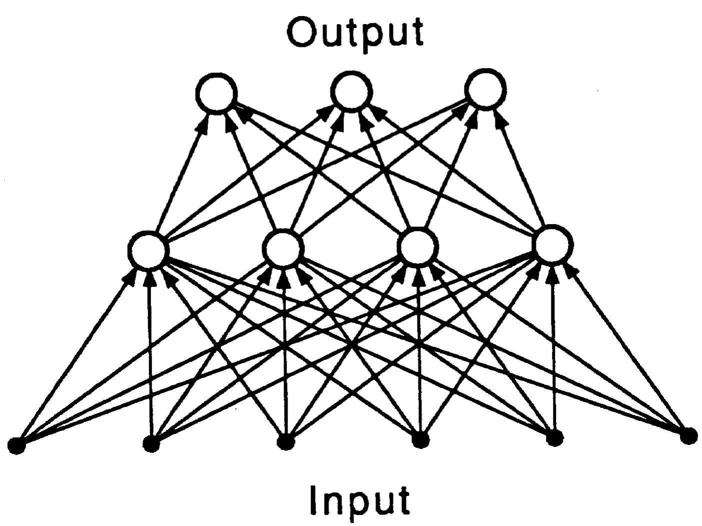
Idea: Change the weights to decrease the error in ouput

Learning Multilayer Networks

We want networks that can learn to map inputs to outputs

- Assume outputs are real-valued between 0 and 1 (instead of only 0 and 1, or -1 and 1)
  - Can threshold output to decide if class 0, class 1, or Reject

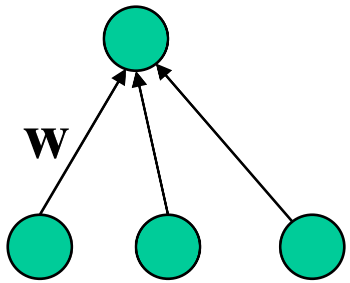
- Idea: Given data, *minimize errors* between network's output and desired output by changing weights



To minimize errors, a *differentiable* output function is desirable (threshold function won't do)

Sigmoidal Networks

$$Output\;\; v = g(\mathbf{w}^T\mathbf{u}) = g(\sum_i w_i u_i)$$



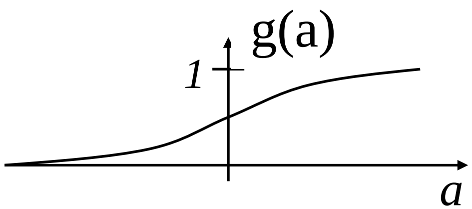
Input nodes

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

The most commonly used differentiable function:

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 ($\mathbf{u}^m$,d^m) (m = 1, …, N), define a sum of squared output errors function (also called a 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 $E(\mathbf{w})$ is minimized

- Gradient Descent: Change **w** in proportion to $-dE/d\mathbf{w}$ (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:

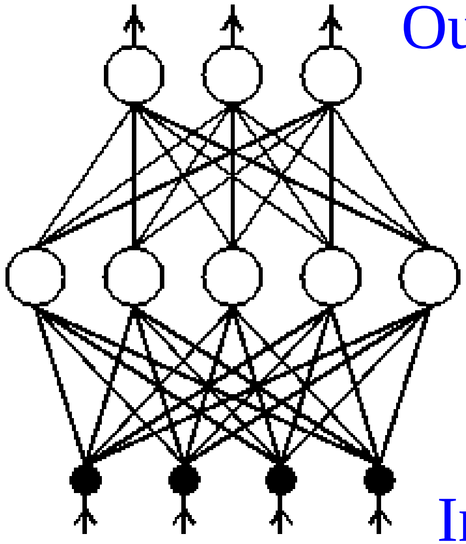
$$\mathbf{w} \rightarrow \mathbf{w} - \varepsilon \frac{dE_1}{d\mathbf{w}}$$

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

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

But wait….

What if we have multiple layers?



Output $\mathbf{v} = (v_1\ v_2 \ldots v_J)^T$; Desired = $\mathbf{d}$

Delta rule can be used to adapt these weights

How do we adapt these?

Input $\mathbf{u} = (u_1\ u_2 \ldots u_K)^T$

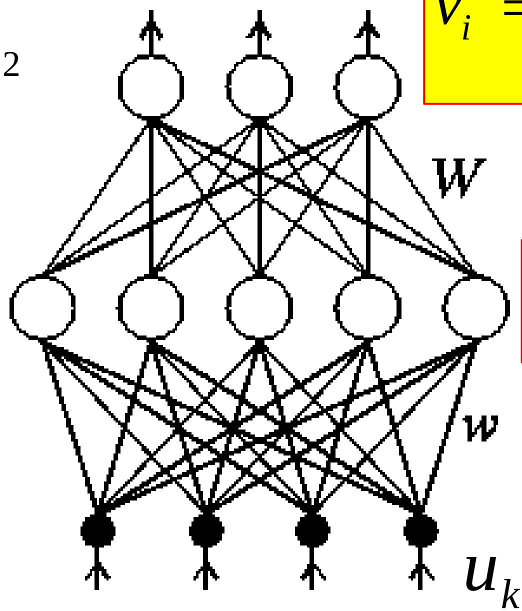
Enter…the backpropagation algorithm

(Actually, nothing but the chain rule from calculus)

Backpropagation: Uppermost layer (delta rule)

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

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



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

$$W_{ji} \rightarrow W_{ji} - \varepsilon \frac{dE}{dW_{ji}}$$

{***gradient descent***}

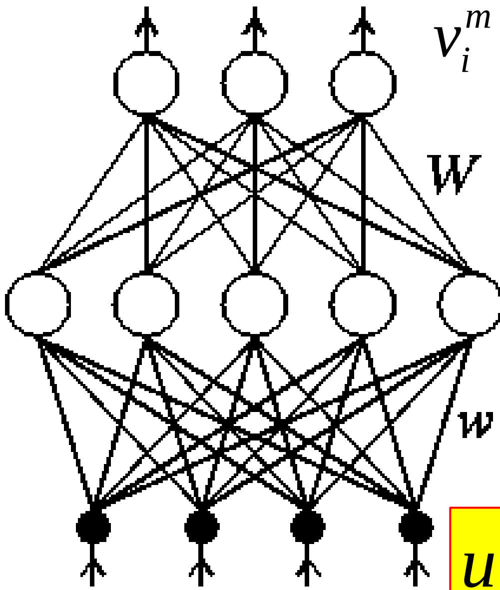
$$\frac{dE}{dW_{ji}} = -(d_i - v_i)g'(\sum_j W_{ji} x_j)x_j$$

{***delta rule***}

Backpropagation: Inner layer (chain rule)

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

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



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

$$\boxed{u_k^m}$$

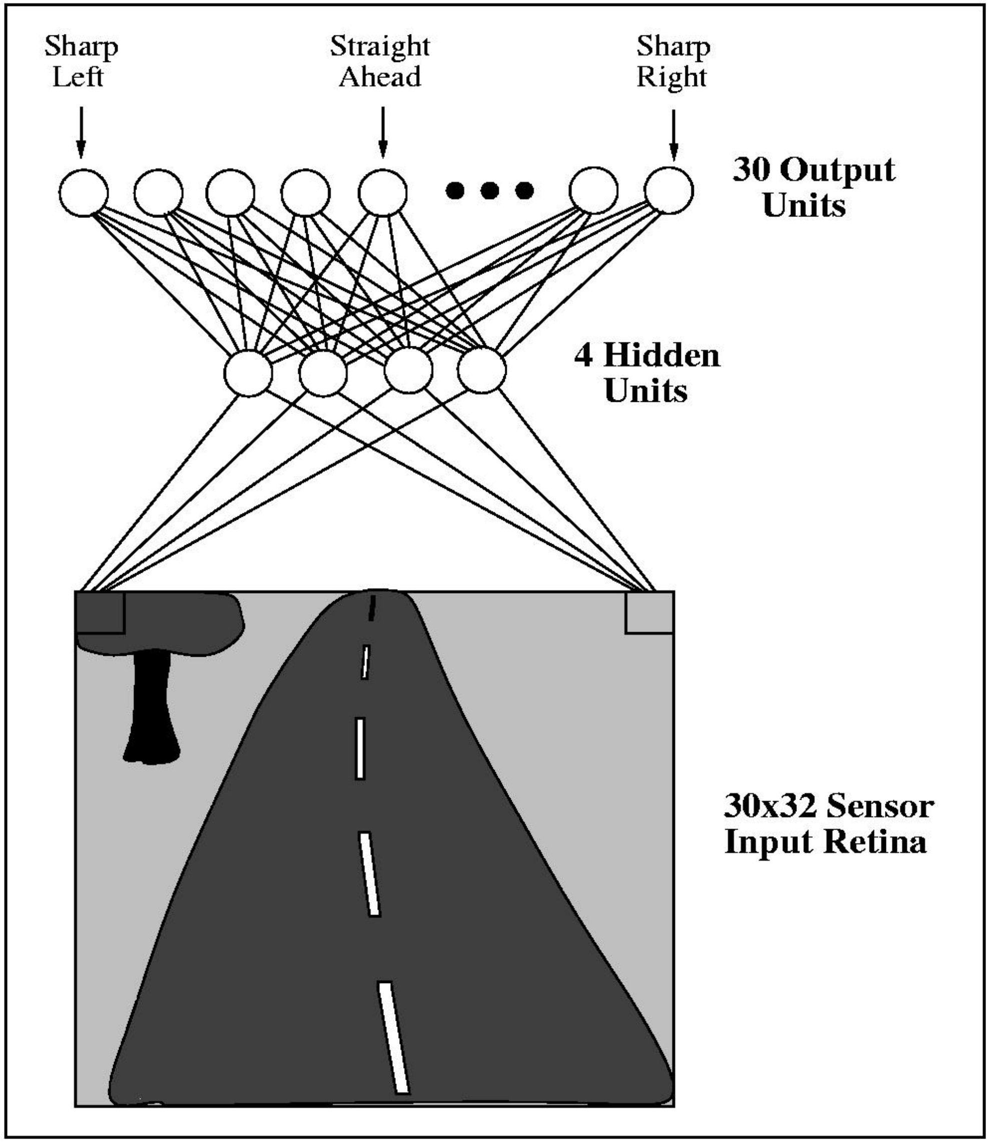
Learning rule for <u>input-hidden weights **w**</u>:

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

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

Example: Learning to Drive

Example Network



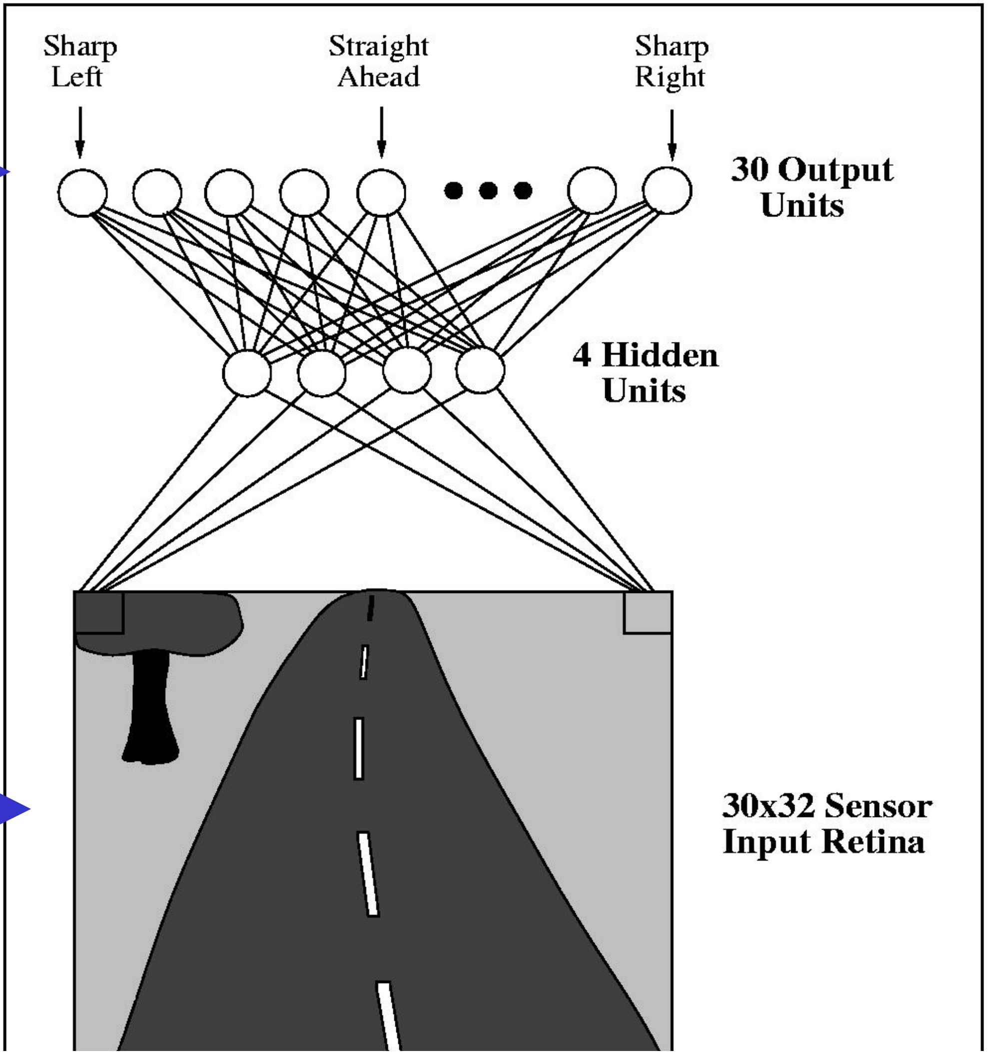
(Pomerleau, 1992)

Example Network

Get steering angle from a human driver

Training Output:

$\mathbf{d} = (d_1 \ d_2 \ \ldots \ d_{30})$

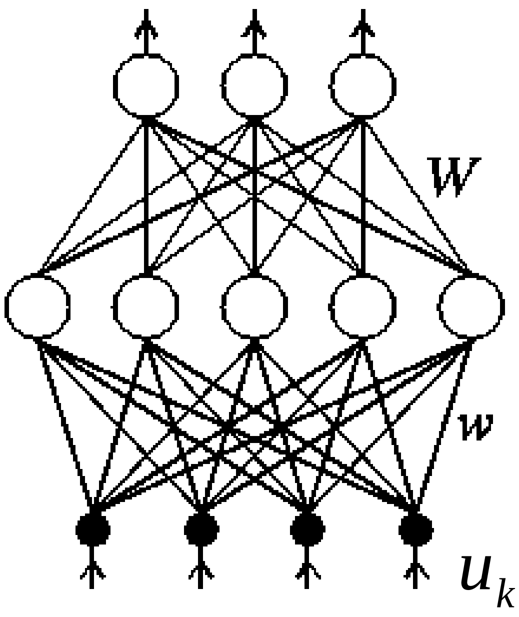


Get current camera image

Training Input $\mathbf{u} = (u_1 \ u_2 \ \ldots \ u_{960})$ = image pixels

Training the network using backprop

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



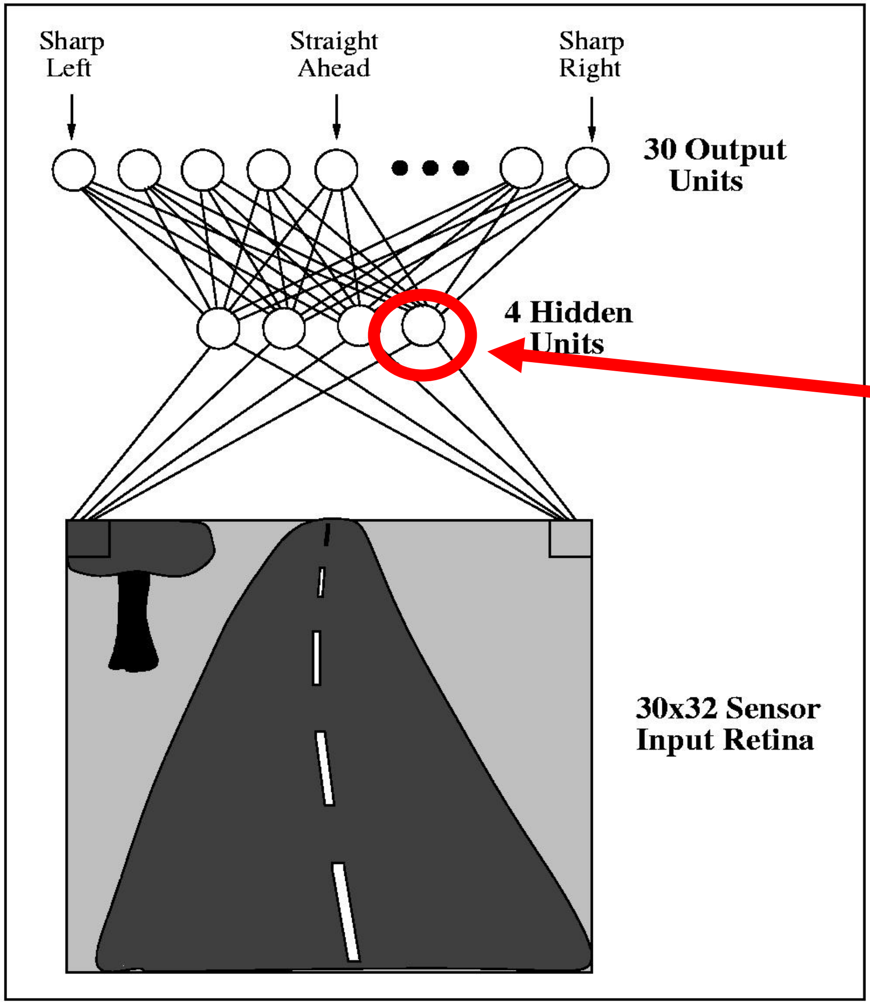
Start with random weights **W**, **w**

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

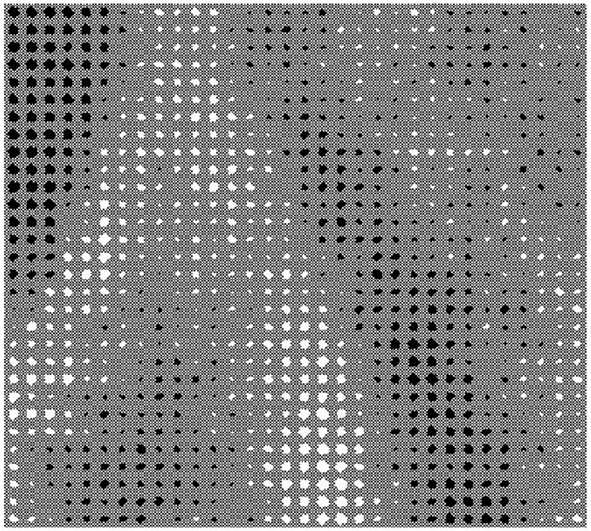
Use backprop to learn **W** and **w** that minimize total error over all output units (labeled *i*):

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

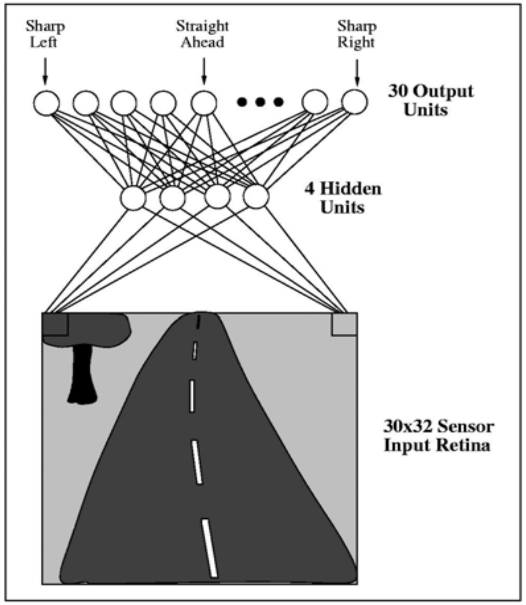
Learning to Drive using Backprop



One of the learned "road features" w_i



ALVINN (Autonomous Land Vehicle in a Neural Network)



CMU Navlab

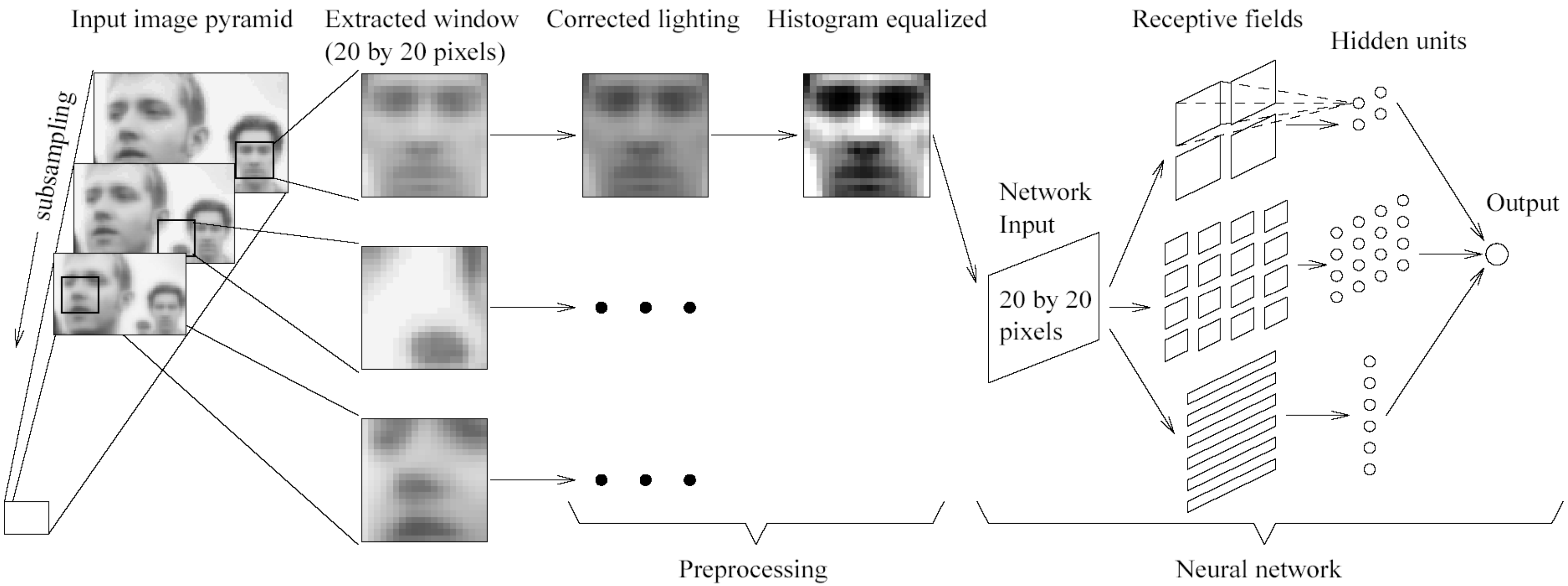
Trained using human driver + camera images

After learning:

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

(Pomerleau, 1992)

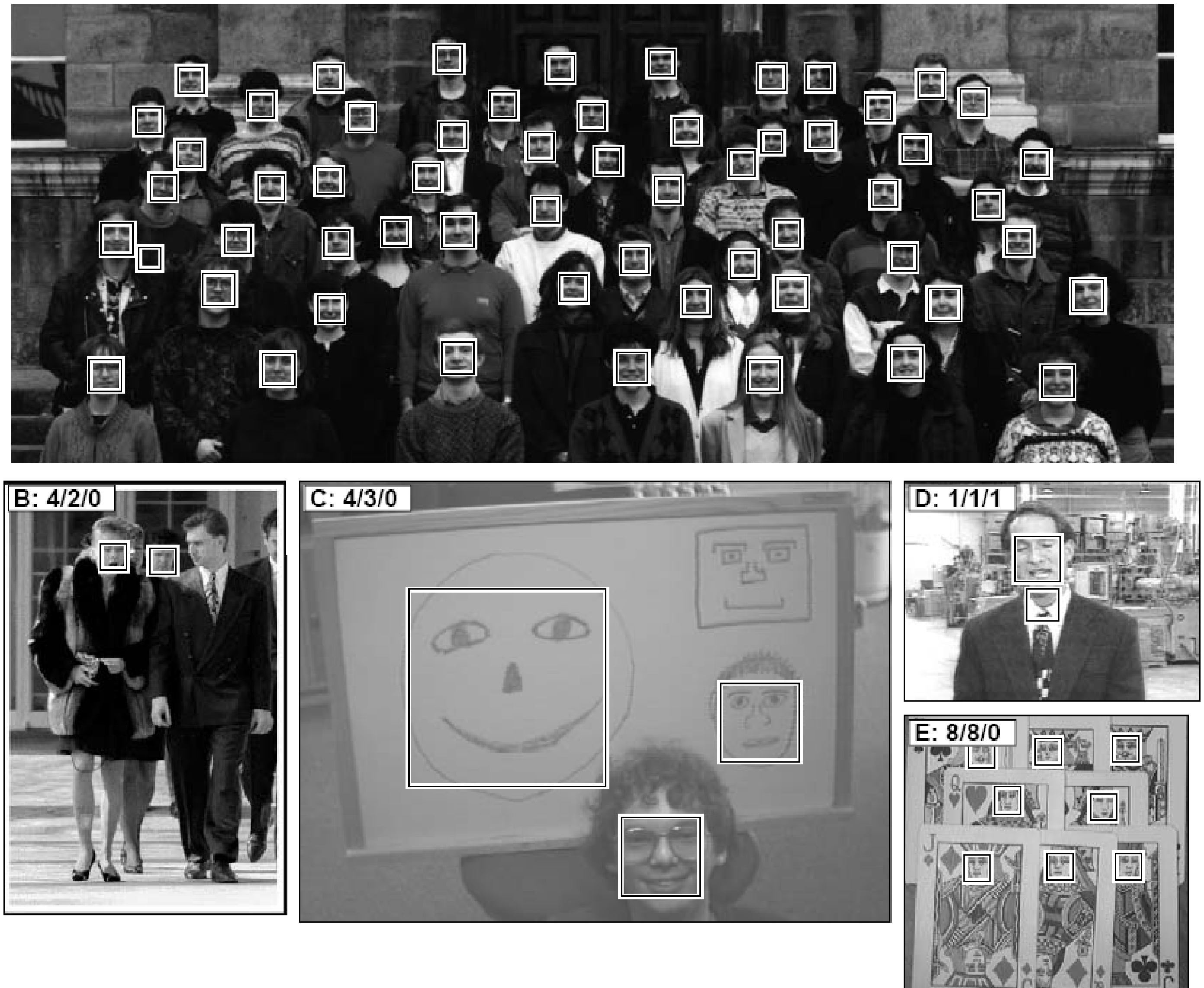
Another Example: Face Detection



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

(Rowley, Baluja & Kanade, 1998)

Face Detection Results



(Rowley, Baluja & Kanade, 1998)

Next Time: More Pattern Recognition & Learning

Things to do:

- Work on Project 2
- Vote on Project 1 Artifacts
- Read Chap. 4

Have a good wekened!