CSE/NB 528

Lecture 13: Supervised Learning

(Chapter 8)

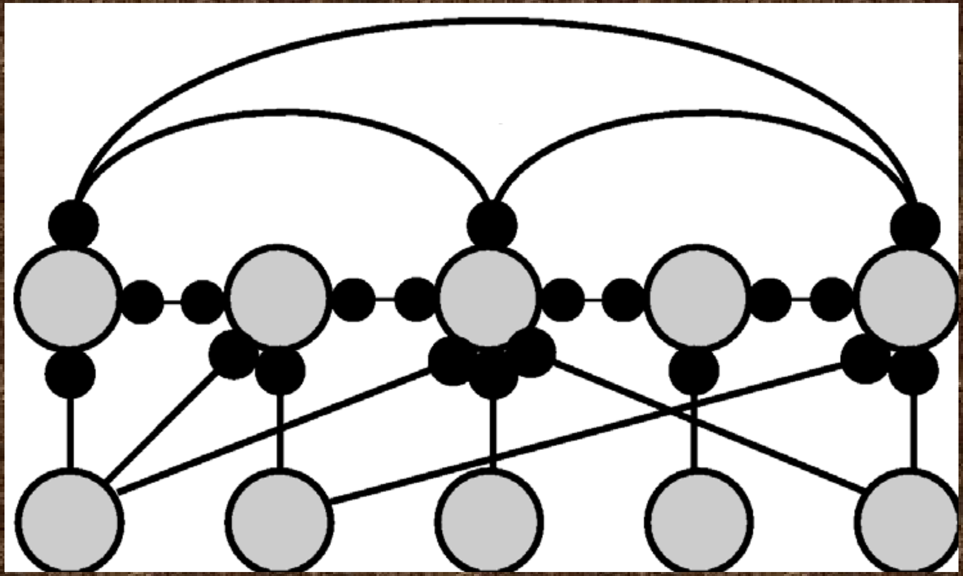
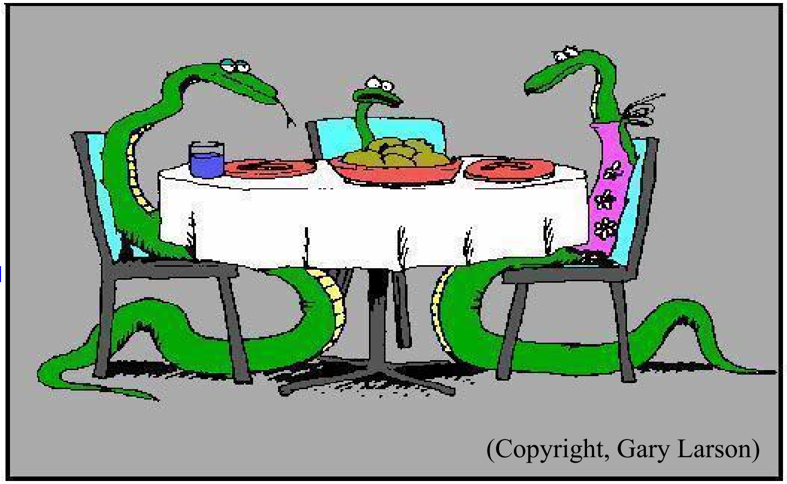


Image from http://clasdean.la.asu.edu/news/images/ubep2001/neuron3.jpg

Lecture figures are from Dayan & Abbott's book
http://people.brandeis.edu/~abbott/book/index.html

What's on the menu today?

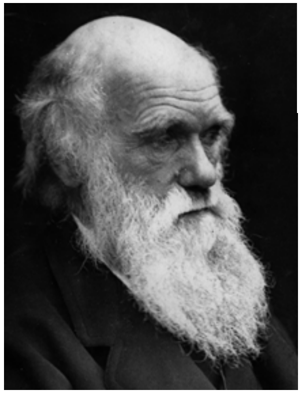
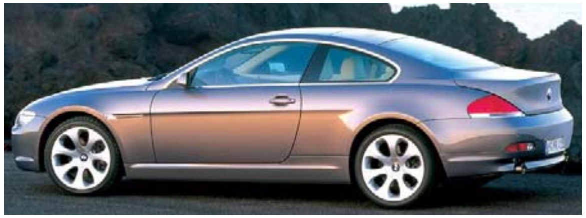


"Oh, brother! . . . Not hamsters again!"

- Supervised Learning
  - Why supervised learning?
    - Classification
    - Function Approximation
  - Perceptrons & Learning Rule
  - Linear Separability: Minsky-Papert deliver the bad news
  - Multilayer networks to the rescue
  - Function Approximation
  - Radial Basis Function Networks
  - Sigmoid Networks
  - Backpropagating (errors)

Example: Face Detection

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

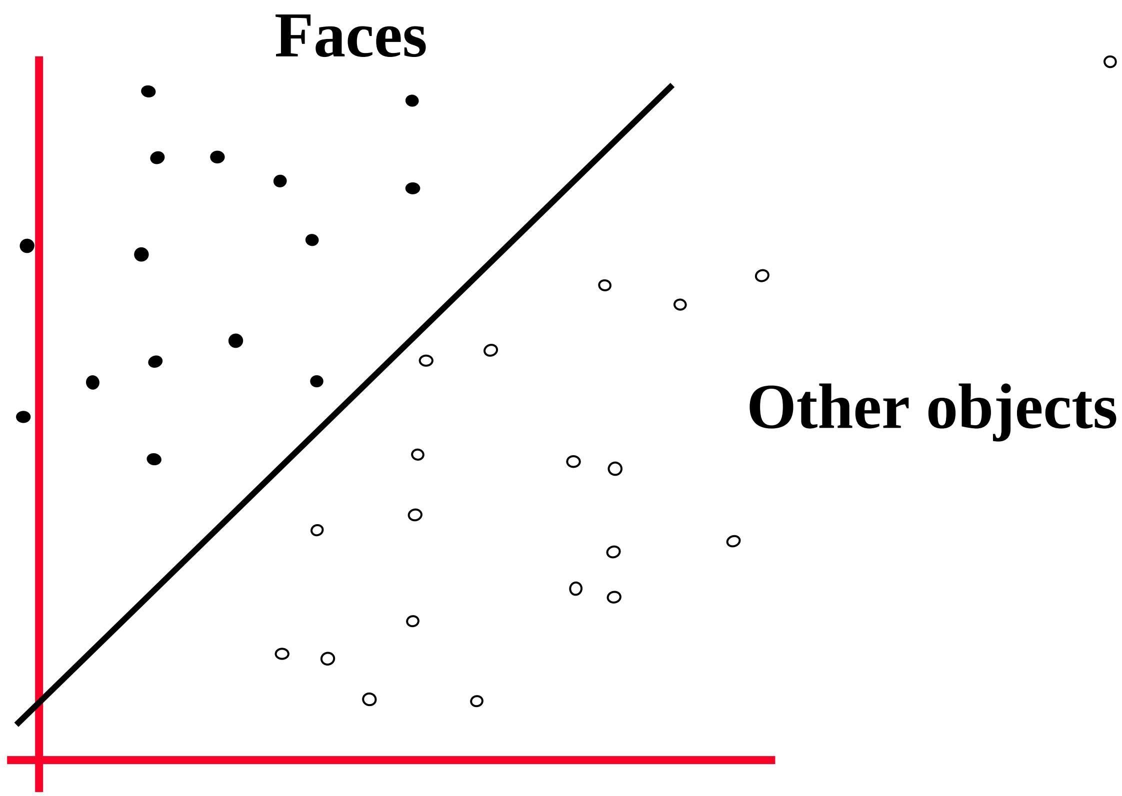







The Classification Problem

- • denotes +1 (faces)
- ∘ denotes -1 (other)



Idea: Find a separating hyperplane (line in this case)

Supervised Learning

- Two Primary Tasks

1. Classification

- Inputs u_1, u_2, … and discrete classes C_1, C_2, …, C_k
- Training examples: (u_1, C_2), (u_2, C_7), etc.
- Learn the mapping from an arbitrary input to its class
- Example: Inputs = images, output classes = face, not a face

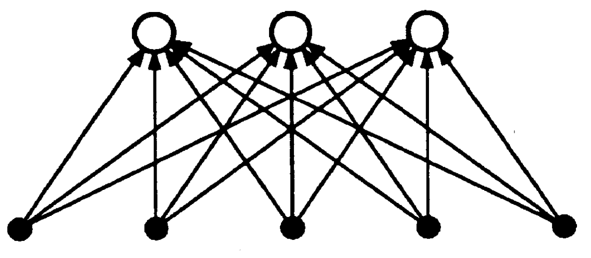
2. Function Approximation (regression)

- Inputs u_1, u_2, … and continuous outputs v_1, v_2, …
- Training examples: (input, desired output) pairs
- Learn to map an arbitrary input to its corresponding output
- Example: Highway driving
  Input = road image, output = steering angle

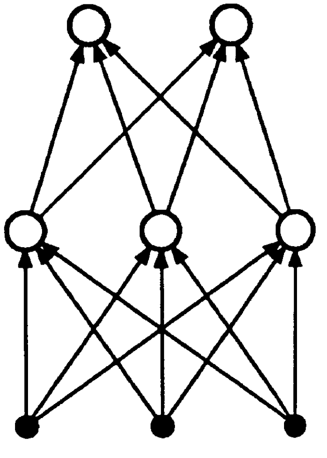
Classification using "Perceptrons"

- Fancy name for a type of layered feedforward networks
- Uses artificial neurons ("units") with binary inputs and outputs

Single-layer



Multilayer

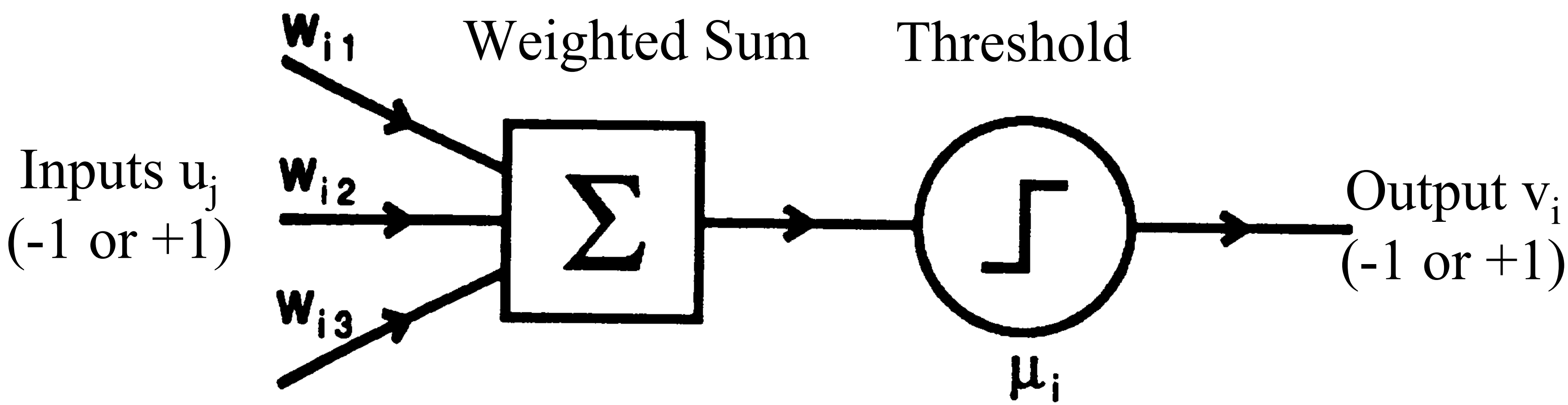


Perceptrons use "Threshold Units"

- Artificial neuron:
  - m binary inputs (-1 or 1) and 1 output (-1 or 1)
  - Synaptic weights w_{ij}
  - Threshold μ_i

$$v_i = \Theta(\sum_j w_{ij} u_j - \mu_i)$$

$\Theta(x) = 1$ if $x \geq 0$ and -1 if $x < 0$



What does a Perceptron compute?

- Consider a single-layer perceptron
  - Weighted sum forms a *linear hyperplane*

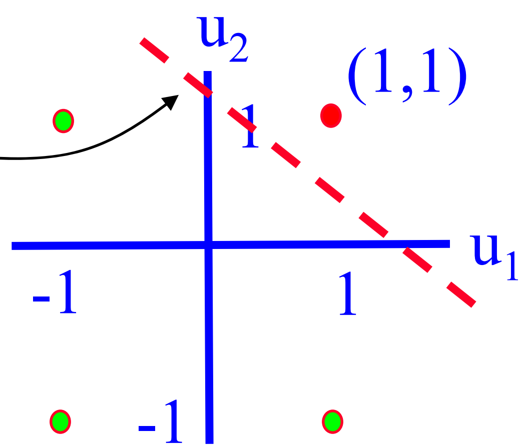
$$\sum_j w_{ij} u_j - \mu_i = 0$$

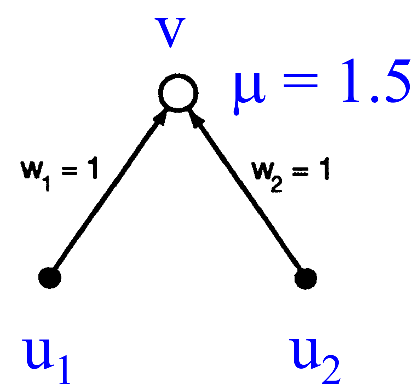
  - Everything *on one side* of hyperplane is in class 1 (output = +1) and everything *on other side* is class 2 (output = -1)
  - Any function that is linearly separable can be computed by a perceptron

Linear Separability

- Example: AND is linearly separable
  - a AND b = 1 if and only if a = 1 and b = 1

Linear hyperplane





Perceptron for AND

Perceptron Learning Rule

- Given inputs **u** and desired output v^d, adjust **w** as follows:

1. Compute error signal $e = (v^d - v)$ where v is the current output

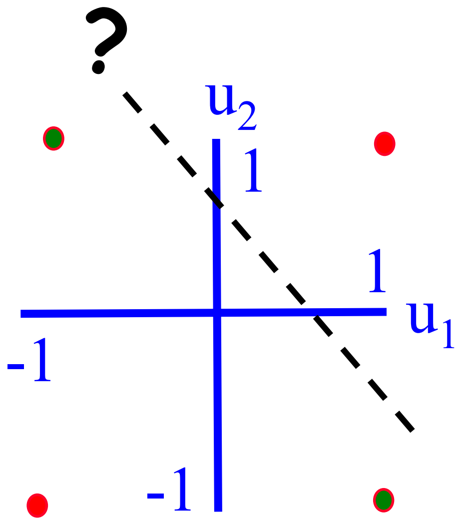
2. Change weights according to the error e
   $\Rightarrow$ For positive inputs, increase weights if error is positive and decrease if error is negative (opposite for negative inputs)

$$\mathbf{w} \rightarrow \mathbf{w} + \varepsilon(v^d - v)\mathbf{u}$$

$A \rightarrow B$ means replace A with B

What about the XOR function?

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

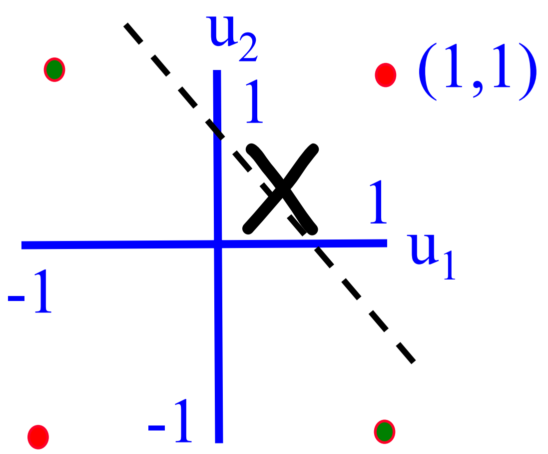


Can a straight line separate the +1 outputs from the -1 outputs?

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!

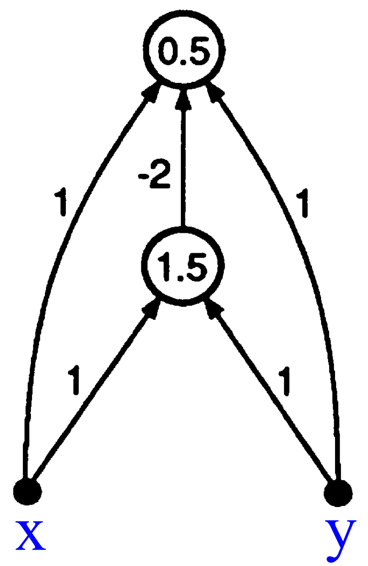
- Minsky and Papert's book showing such negative results put a damper on neural networks research for over a decade!



How do we deal with linear inseparability?

Solution in 1980s: Multilayer perceptrons

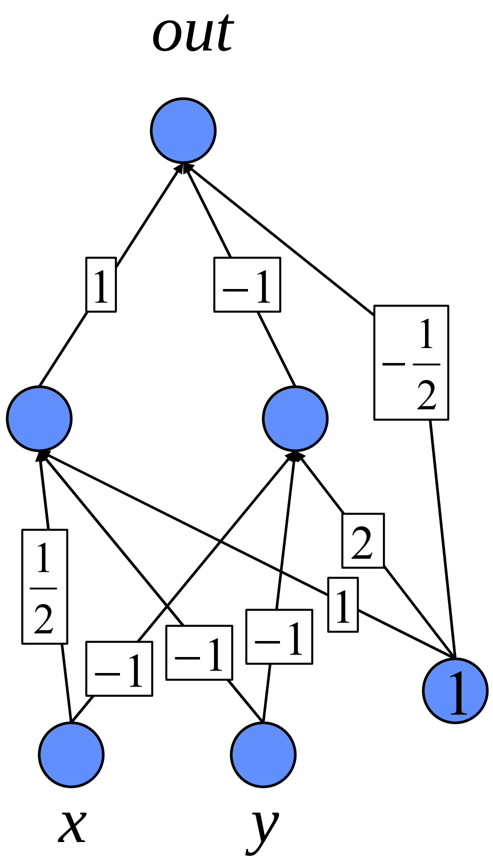
- Removes limitations of single-layer networks
  - Can solve XOR
- An example of a two-layer perceptron that computes XOR

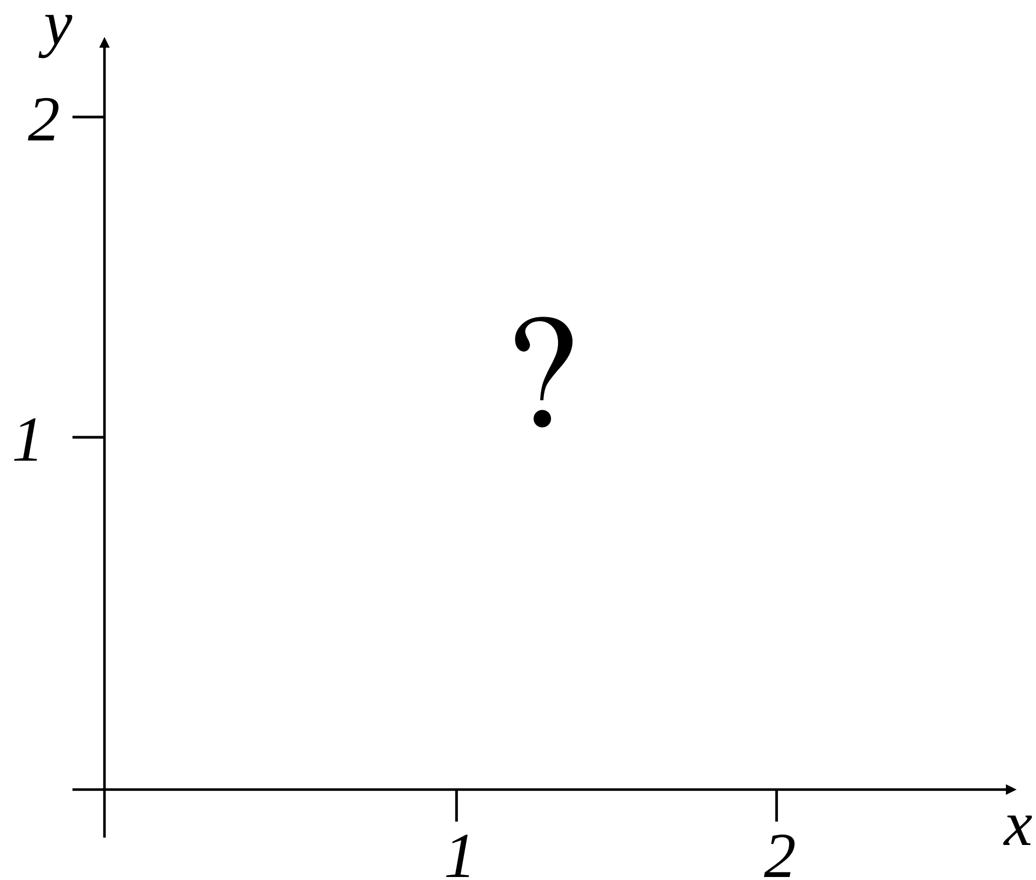


- Output is +1 if and only if $x + y - 2\Theta(x + y - 1.5) - 0.5 > 0$

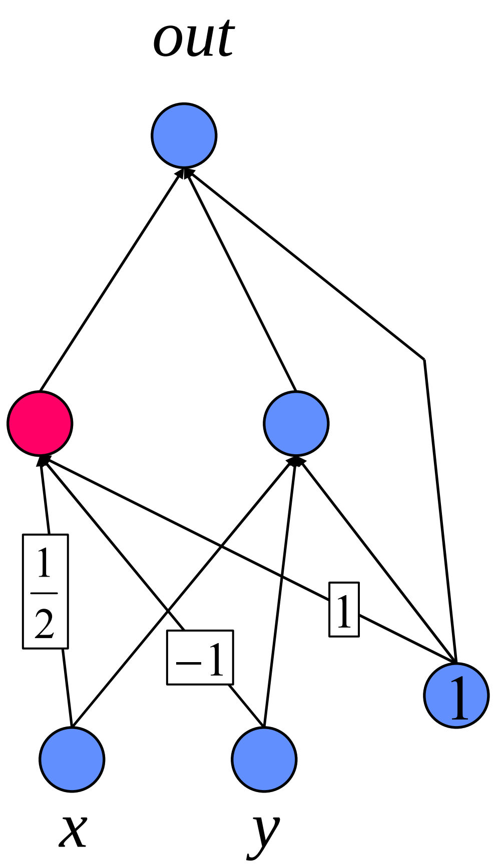
(Here, inputs x, y are assumed to be 0 or 1)

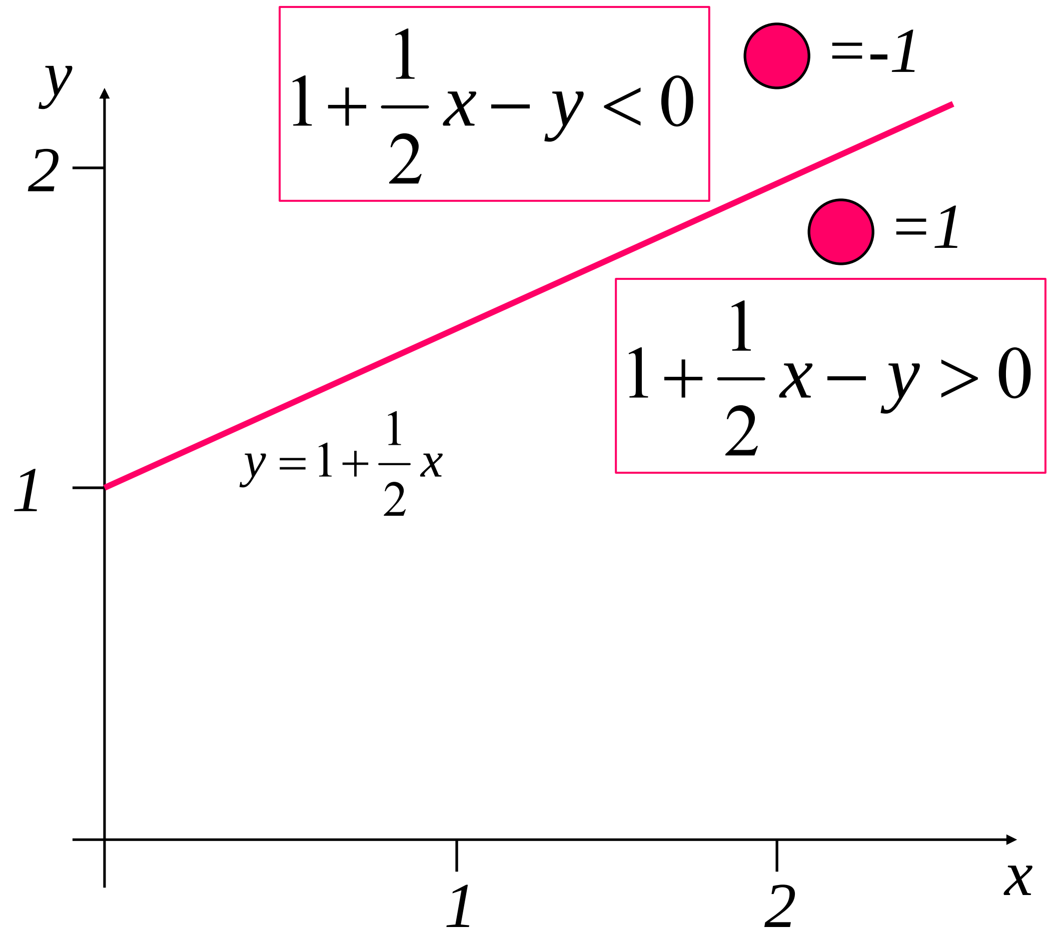
Multilayer Perceptron: What does it do?



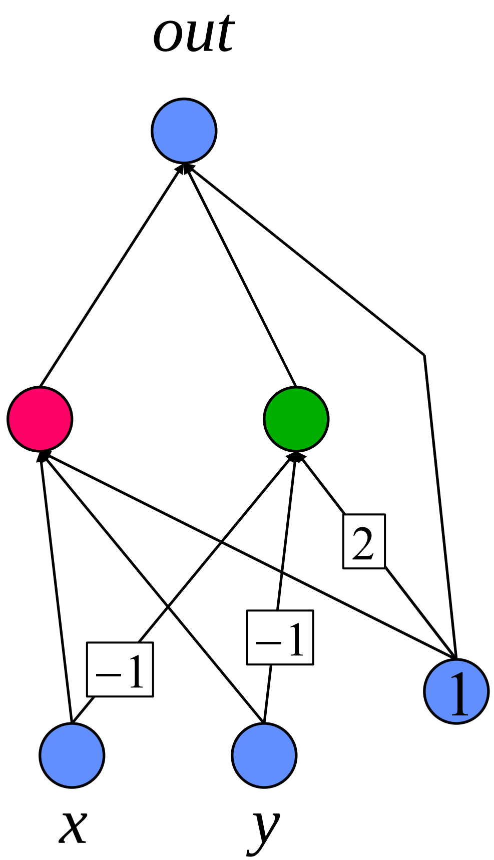


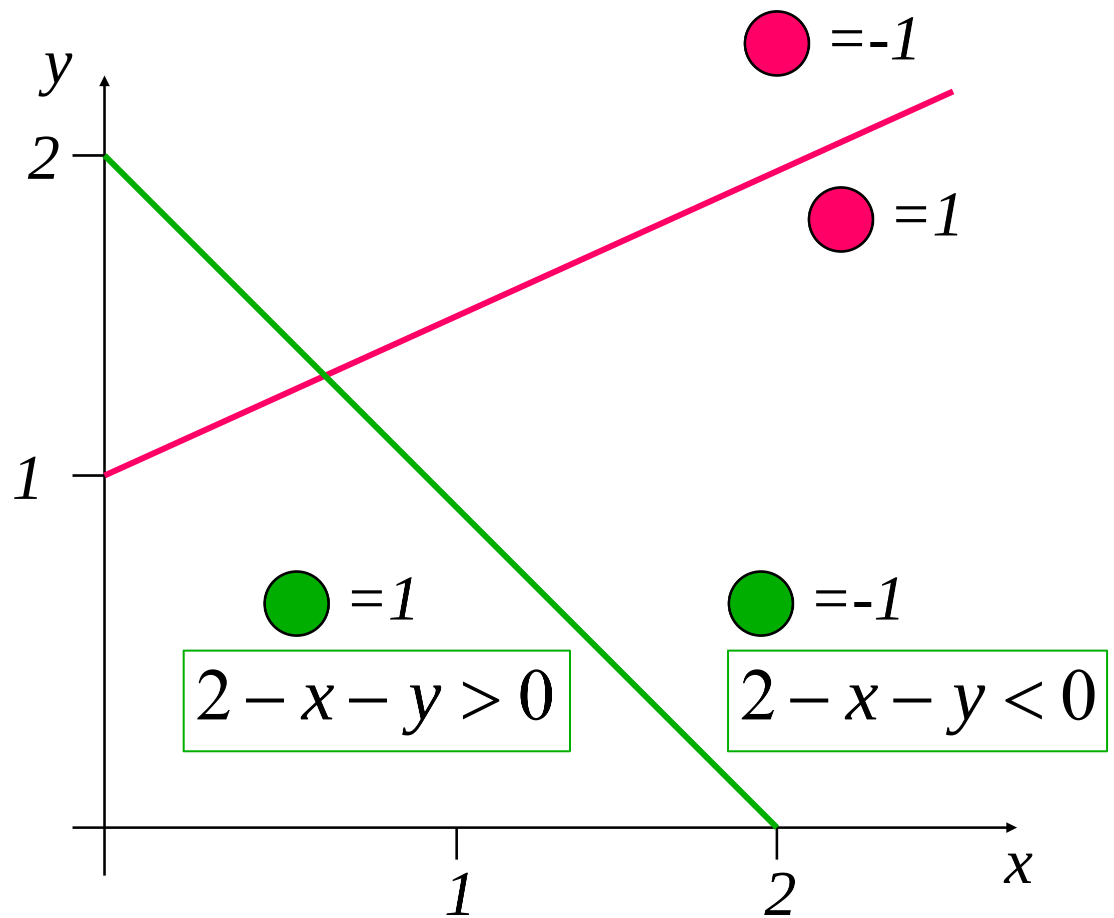
Example: Perceptrons as Constraint Satisfaction Networks



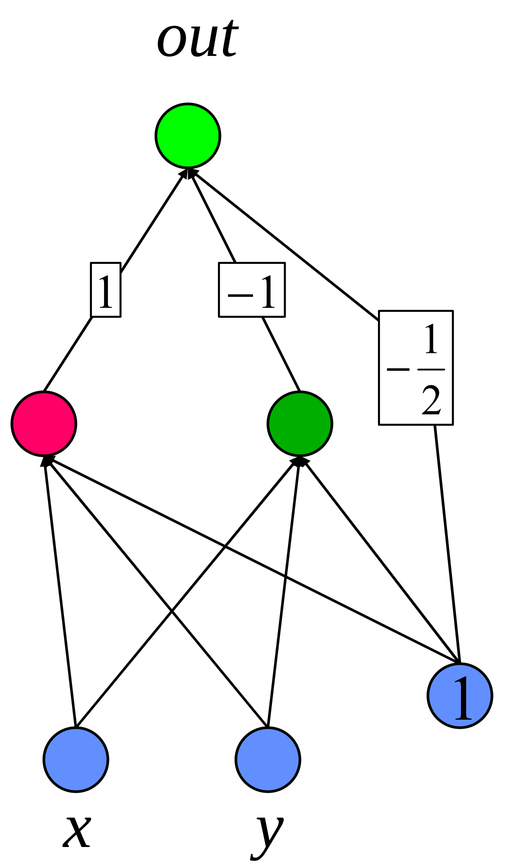


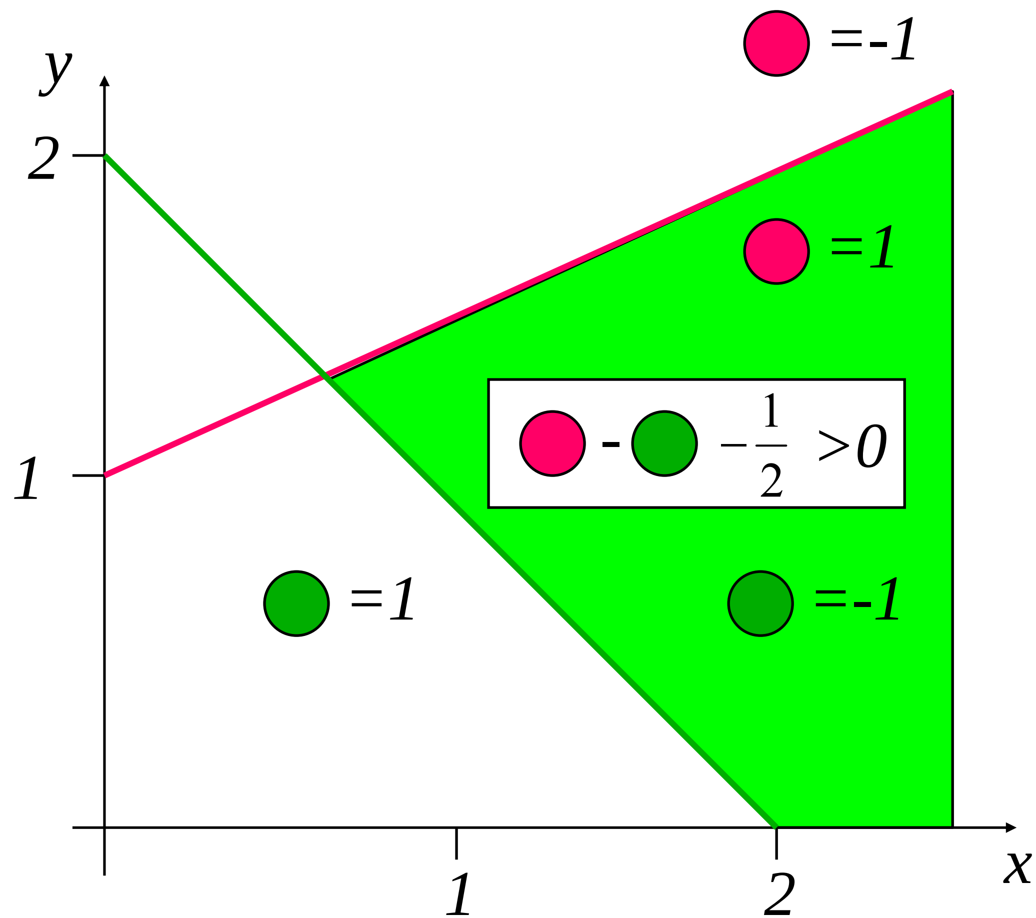
Example: Perceptrons as Constraint Satisfaction Networks



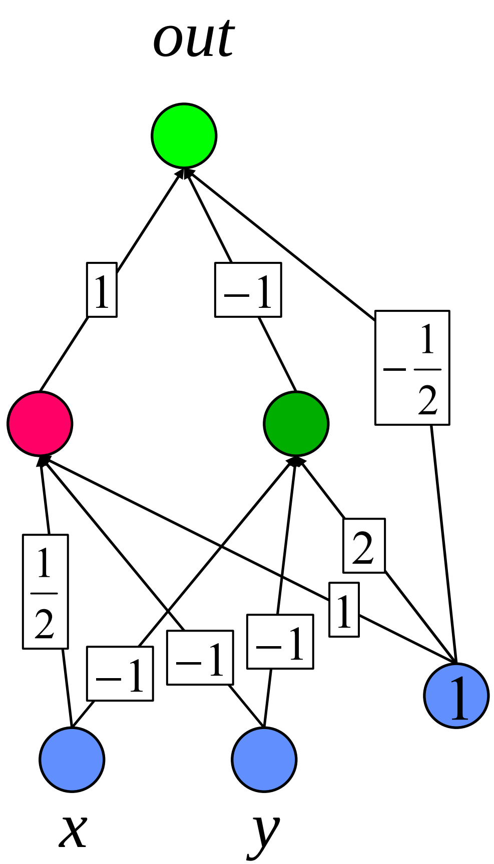


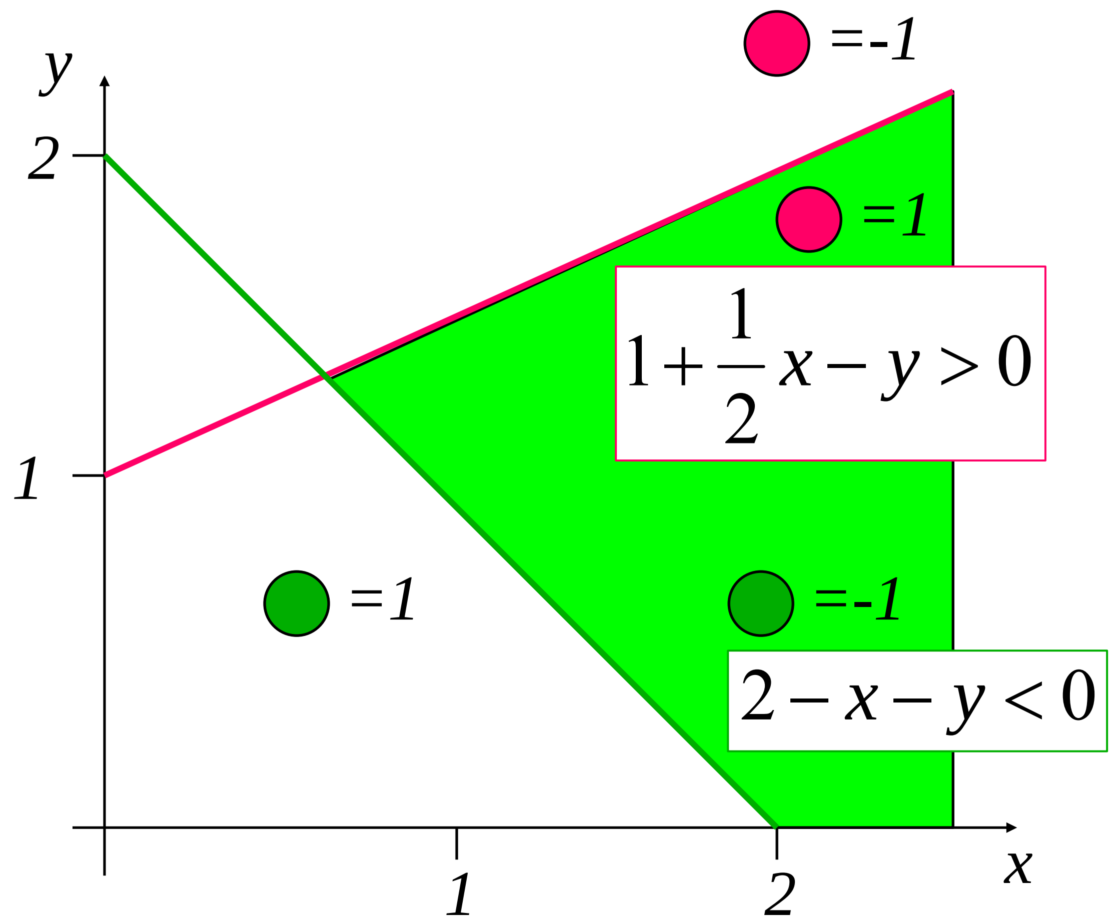
Example: Perceptrons as Constraint Satisfaction Networks





Perceptrons as Constraint Satisfaction Networks





What if you want to approximate a continuous function?

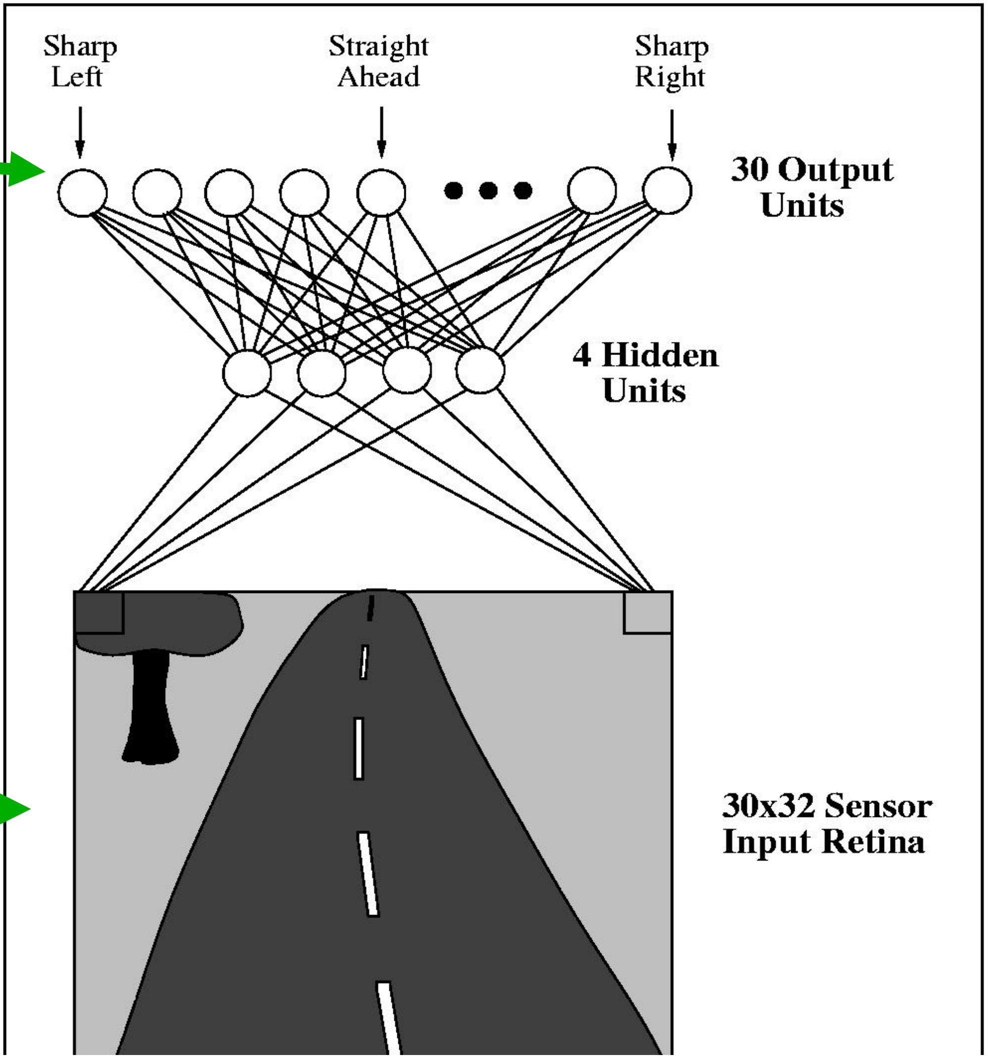
Can a network learn to drive?

Example Network

Steering angle

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

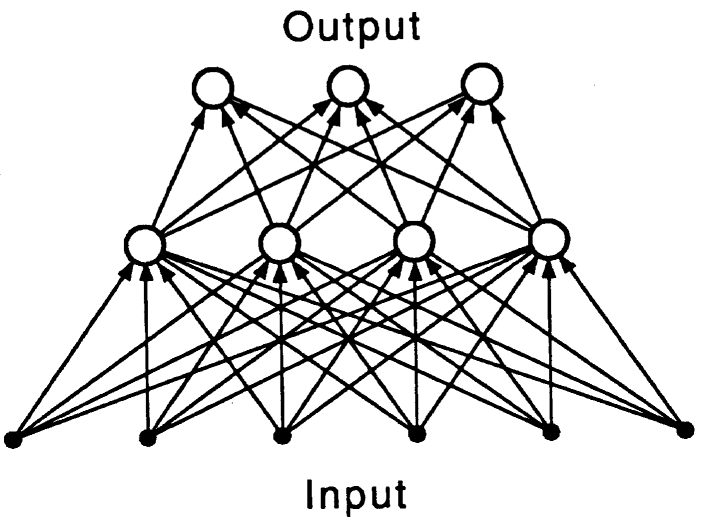
Current image



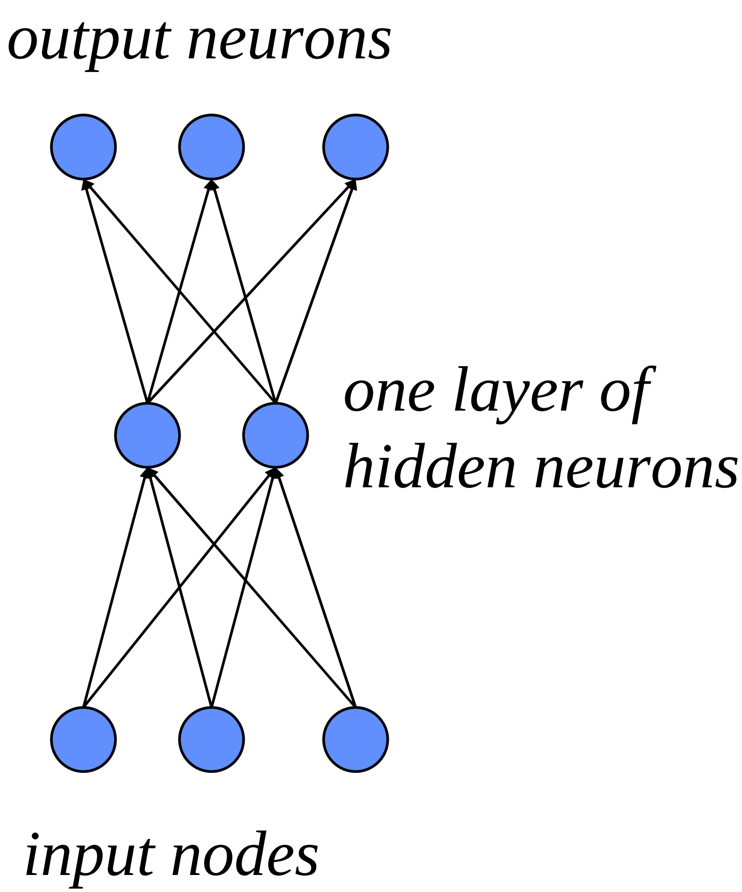
Input $\mathbf{u} = (u_1 \ u_2 \ \ldots \ u_{960})$ = image pixels

Function Approximation

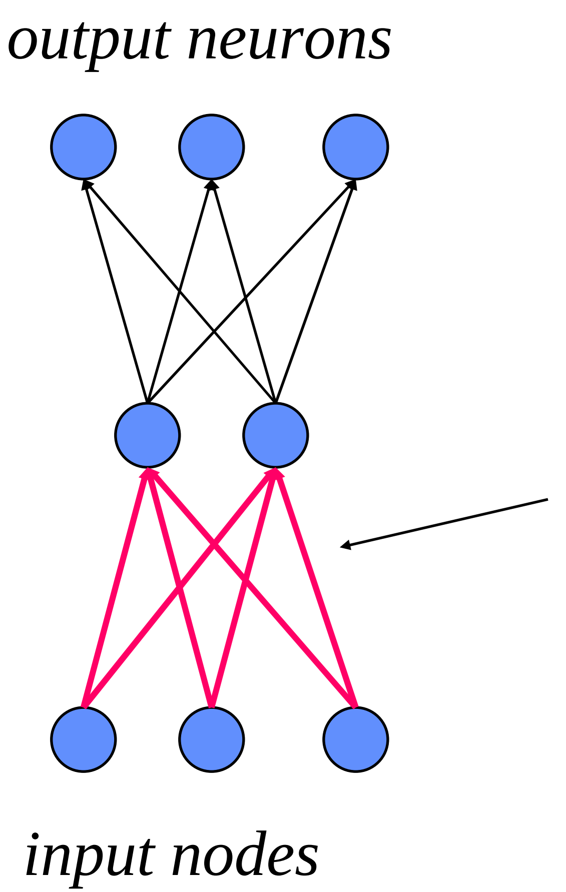
- We want networks that can <u>learn a function</u>
  - Network maps real-valued inputs to real-valued outputs
  - Want to generalize to predict outputs for new inputs
  - <u>Idea</u>: Given input data, map input to desired output by *adapting weights*



Radial Basis Function (RBF) Networks



Radial Basis Function Networks

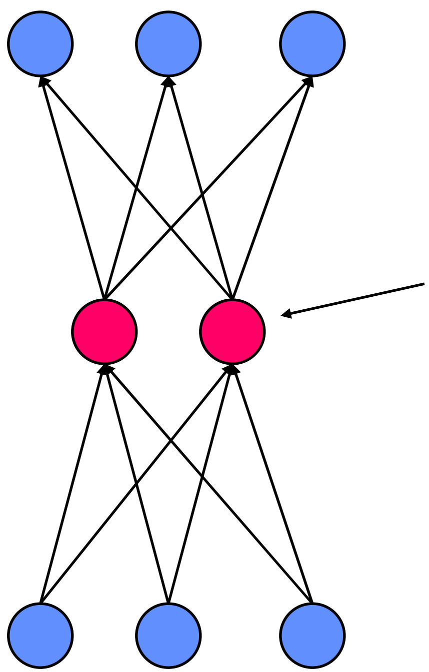


"activation" function:

$$a_j = \sqrt{\sum_{i=1}^{n} (x_i - \mu_{i,j})^2}$$

Radial Basis Function Networks

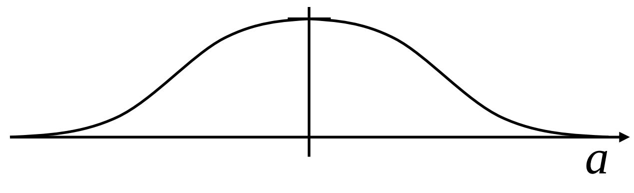
output neurons



Hidden layer:
(Gaussian bell-shaped function)

input nodes

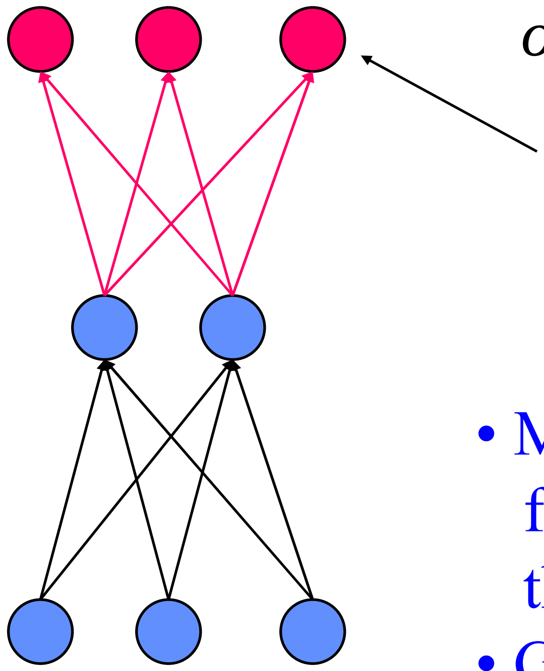
h(a)



$$h(a) = e^{-\frac{a^2}{2\sigma^2}}$$

Radial Basis Function Networks

output neurons



input nodes

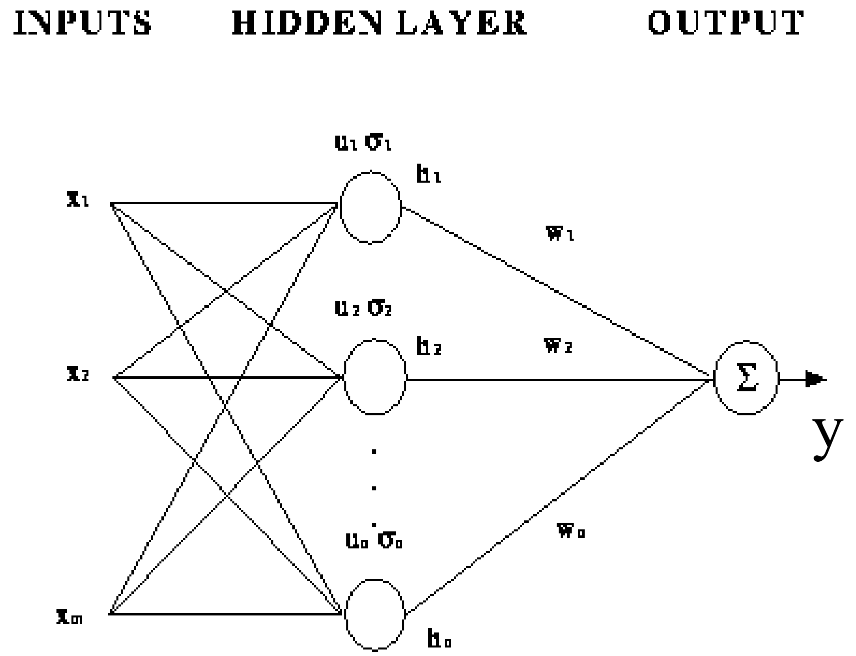
output of network:

$$\text{out}_j = \sum_i w_{i,j} h_i$$

- Main Idea: Use a mixture of Gaussian functions of the input to approximate the output
- Gaussians are called "basis functions"

RBF networks

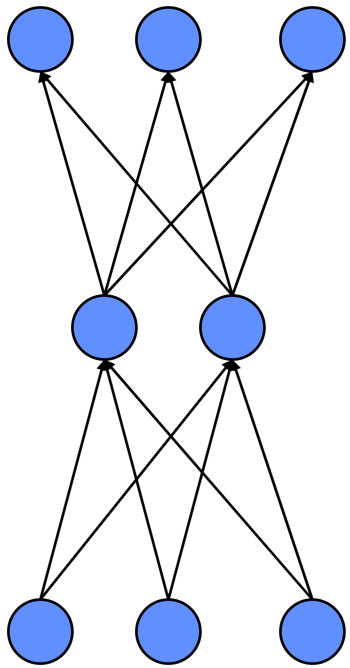
- Radial basis functions
  - Hidden units store means and variances
  - Hidden units compute a Gaussian function of inputs $x_1, \ldots x_n$
- Learn weights w_i, means μ_i, and variances σ_i by minimizing squared error function (gradient descent learning)



$$h_i = exp[-\frac{(\mathbf{x} - \mathbf{u}_i)^T (\mathbf{x} - \mathbf{u}_i)}{2\sigma^2}],\ y = \sum_i h_i w_i$$

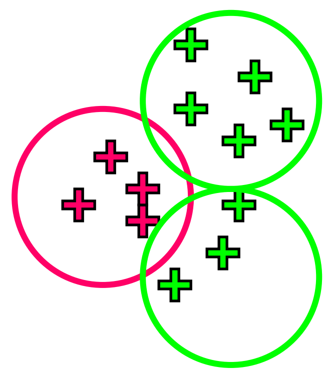
RBF Networks versus Multilayer Perceptrons

output neurons

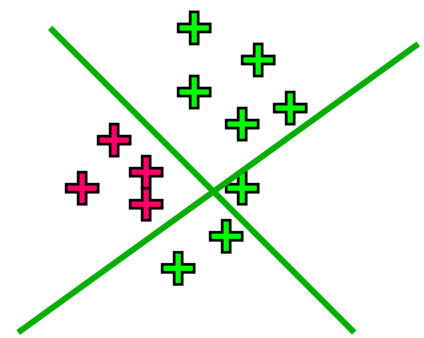


input nodes

RBF:

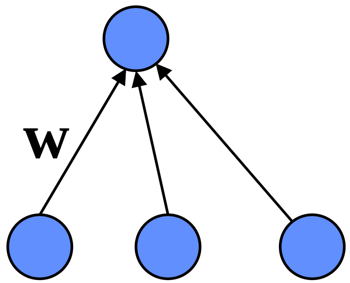


MLP:



Another Model: Sigmoidal Networks

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

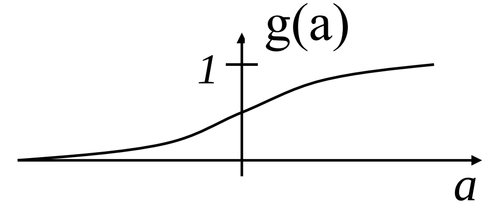


Input nodes

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

Sigmoid function:

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



Sigmoid is 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)

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

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

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:

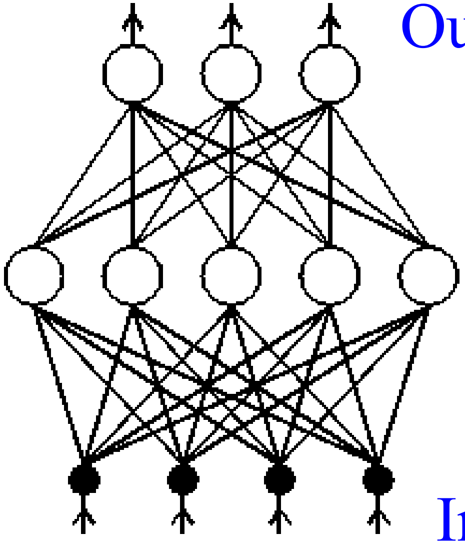
$$\mathbf{w} \rightarrow \mathbf{w} - \varepsilon \frac{dE_1}{d\mathbf{w}}$$

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

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

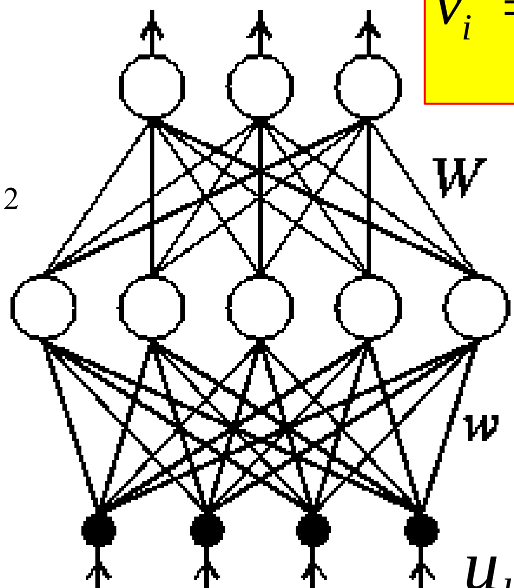
Enter…the backpropagation algorithm

(Actually, nothing but the chain rule from calculus)

Backpropagation: Uppermost layer (delta rule)

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

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



x_j

u_k

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

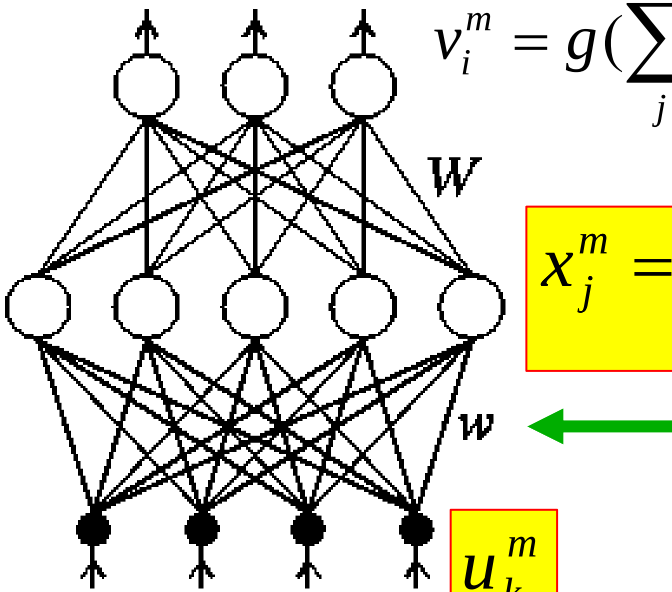
$$W_{ji} \rightarrow W_{ji} - \varepsilon \frac{dE}{dW_{ji}}$$ {*gradient descent*}

$$\frac{dE}{dW_{ji}} = -(d_i - v_i)\, g'\left(\sum_j W_{ji} x_j\right) x_j$$ {*delta rule*}

Backpropagation: Inner layer (chain rule)

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

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



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

u_k^m

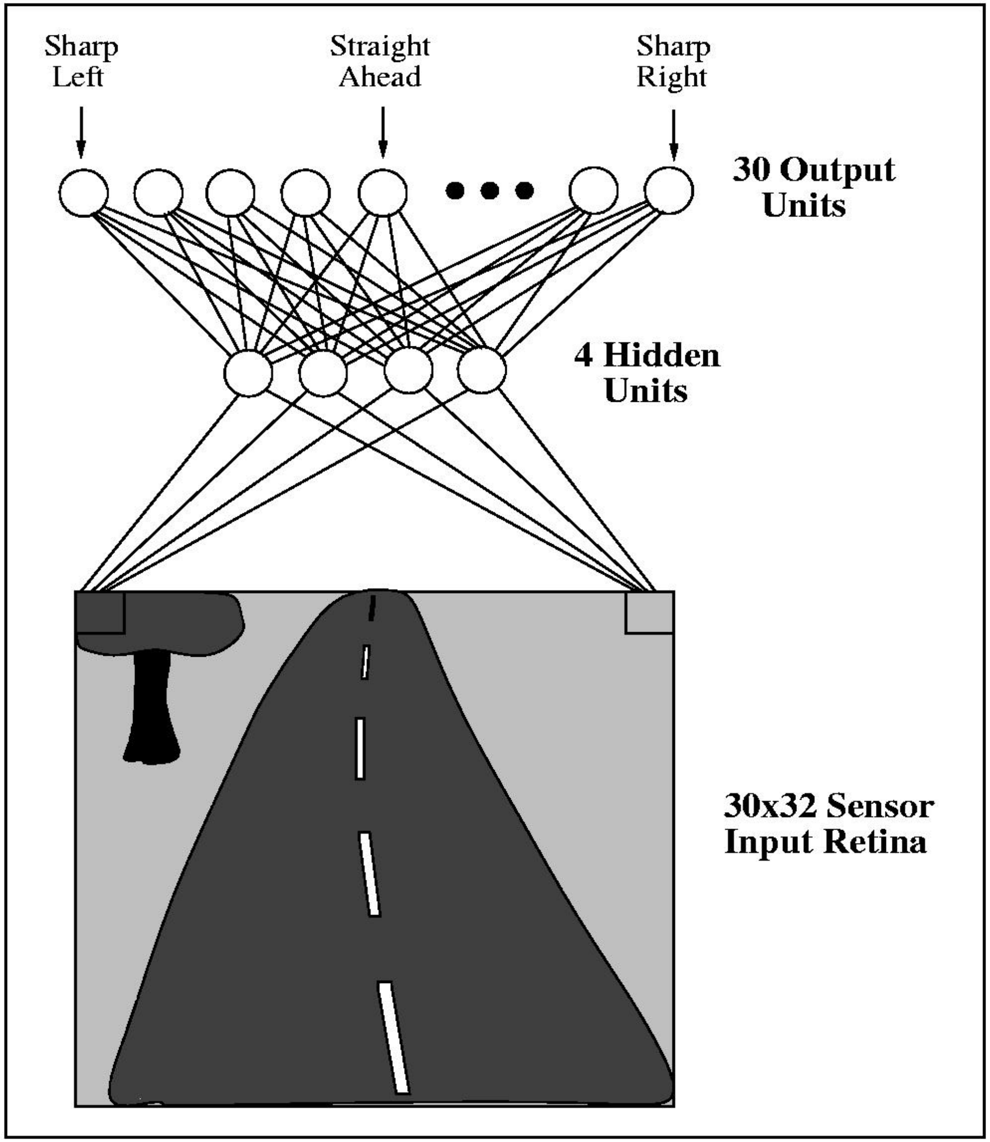
Learning rule for <u>input-hidden weights **w</u>:**

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

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

Example: Learning to Drive

Example Network

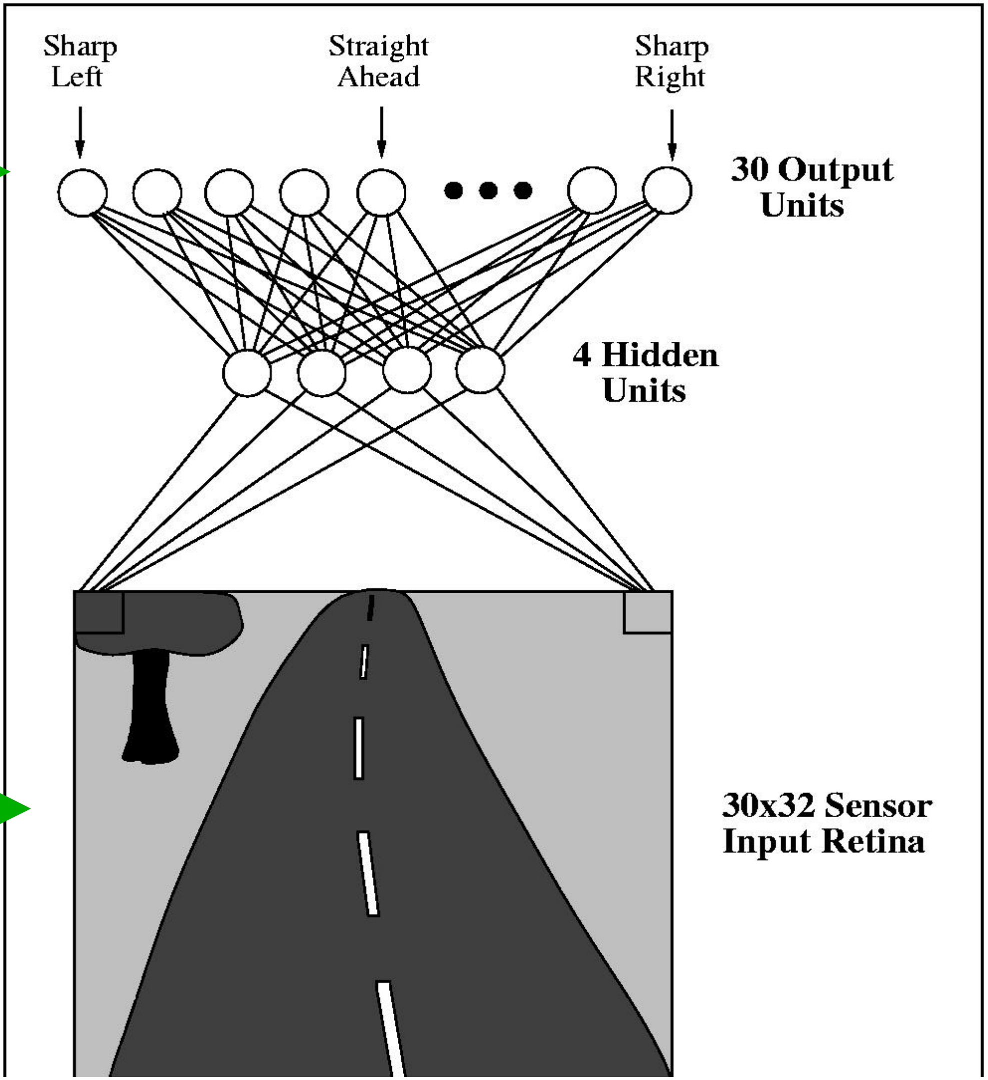


(Pomerleau, 1992)

Example Network

Get steering angle

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

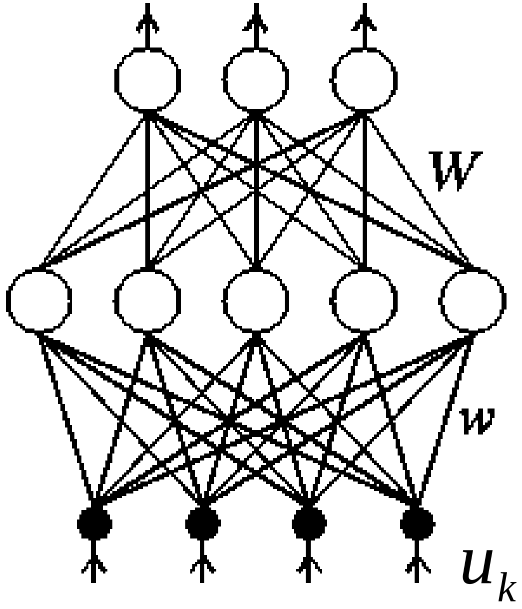


Get current camera image

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

Training the network using backprop

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



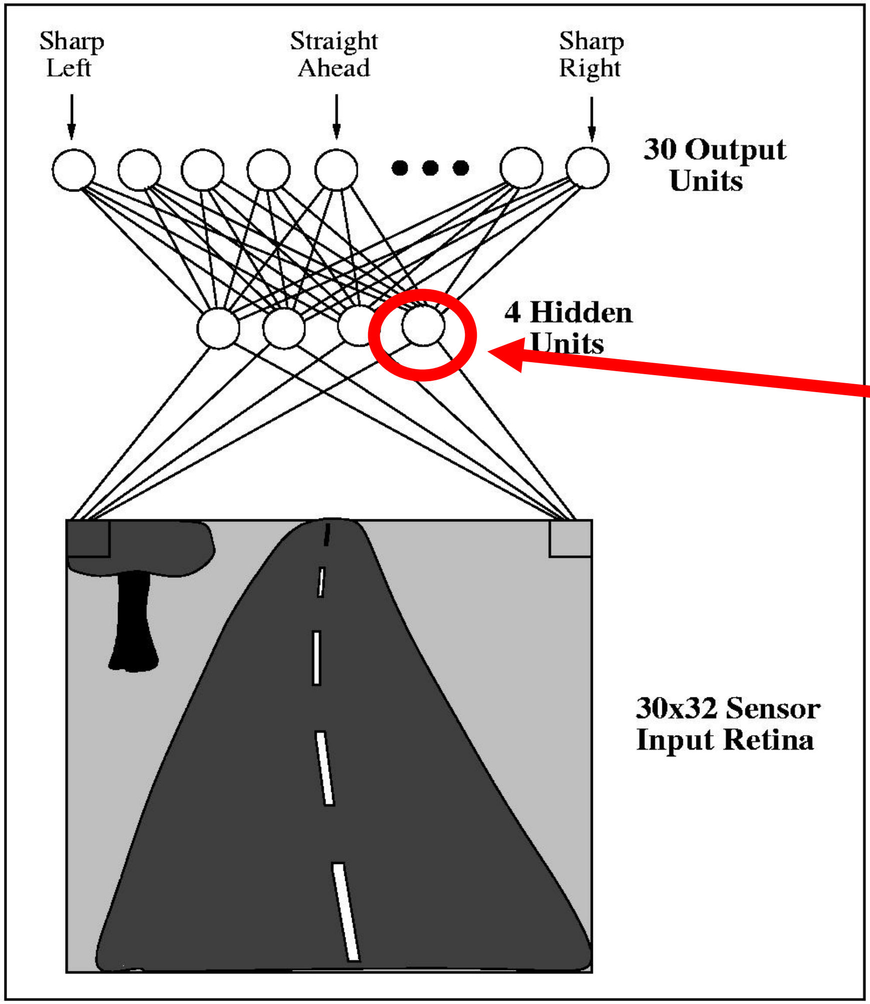
Start with random weights **W**, **w**

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

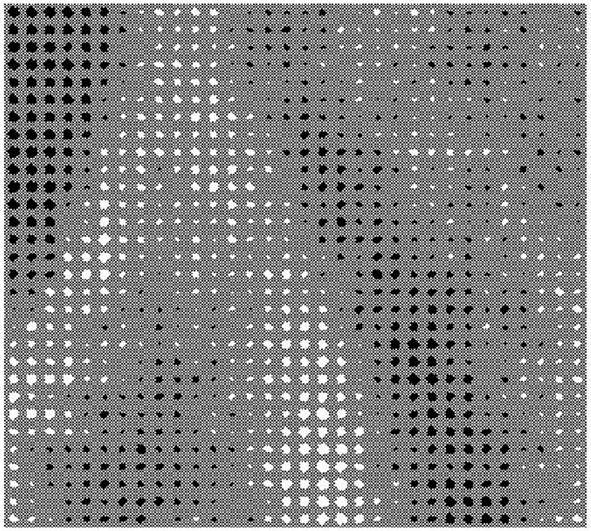
Use backprop to learn **W** and **w** that minimize total error over all output units (labeled *i*):

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

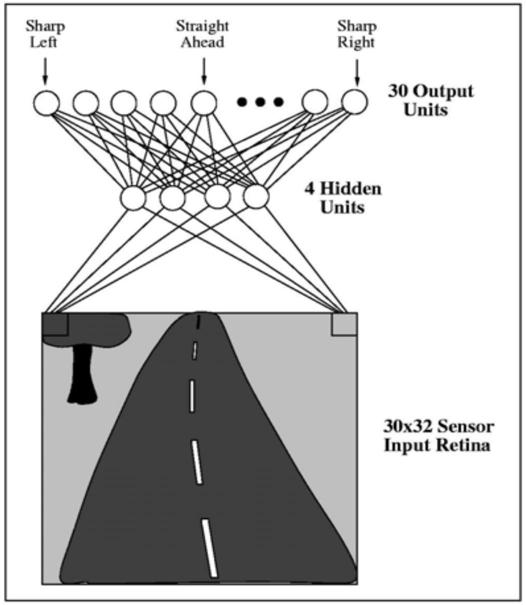
Learning to Drive using Backprop



One of the learned "road features" w_i



ALVINN (Autonomous Land Vehicle in a Neural Network)



CMU Navlab

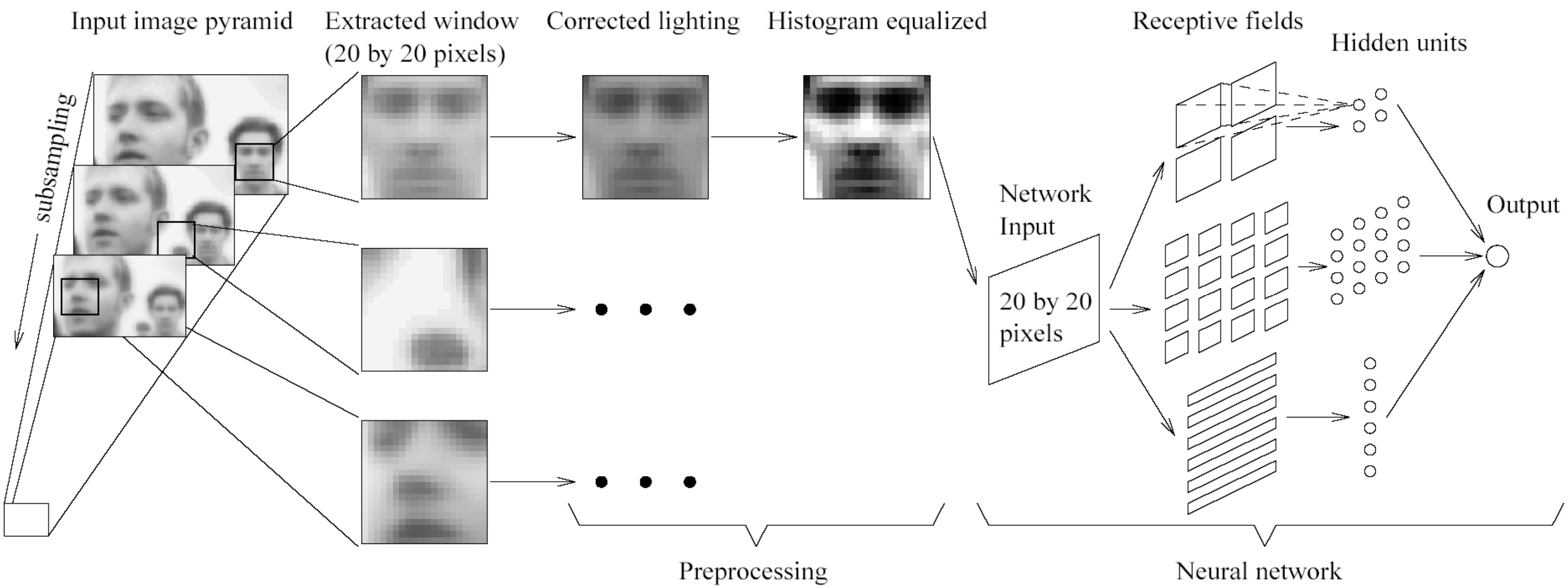
Trained using human driver + camera images

After learning:

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

(Pomerleau, 1992)

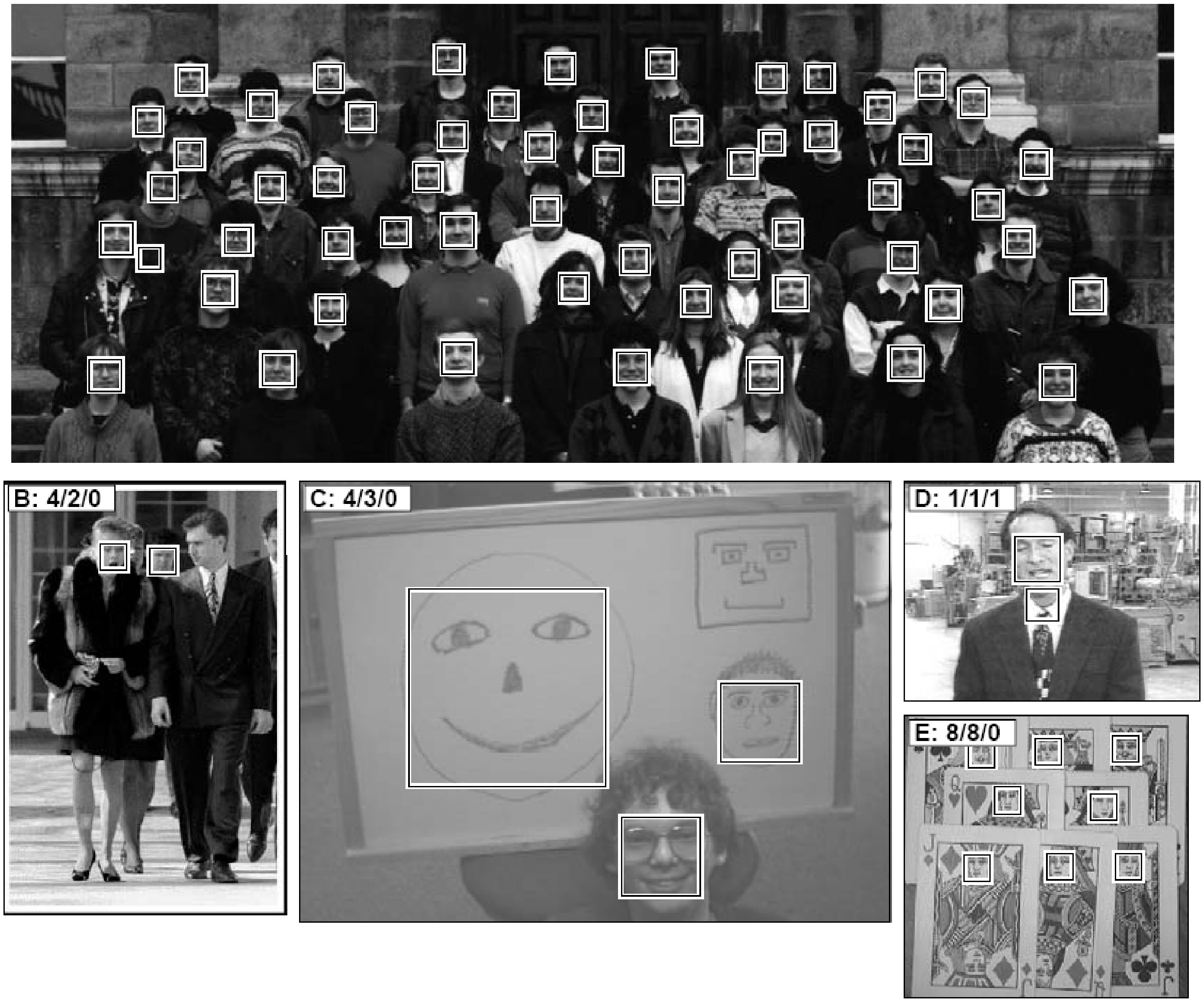
Another Example: Face Detection



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

(Rowley, Baluja & Kanade, 1998)

Face Detection by a Neural Network



(Rowley, Baluja & Kanade, 1998)

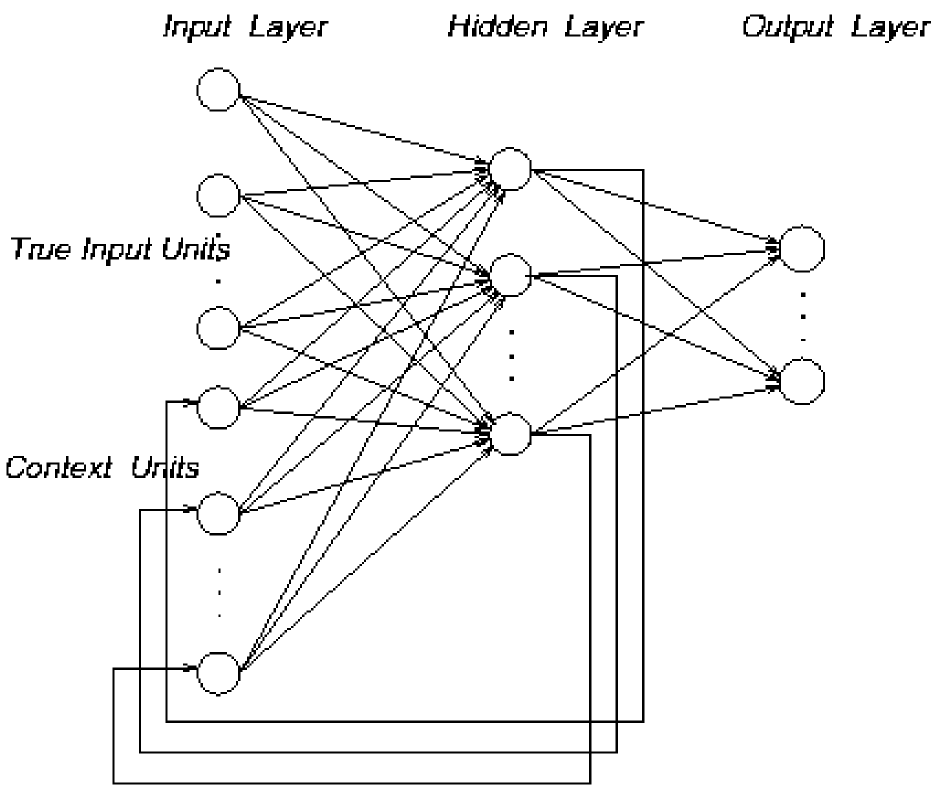
Recurrent Supervised Networks

- Why use recurrent networks?
  - To keep track of recent history and context
  - Can learn temporal patterns (time series or oscillations)
- Examples
  - Recurrent backpropagation networks: for small sequences, *unfold network in time dimension* to get multi-layered network and use backpropagation learning
  - Partially recurrent networks E.g. Elman net

Partially Recurrent Networks

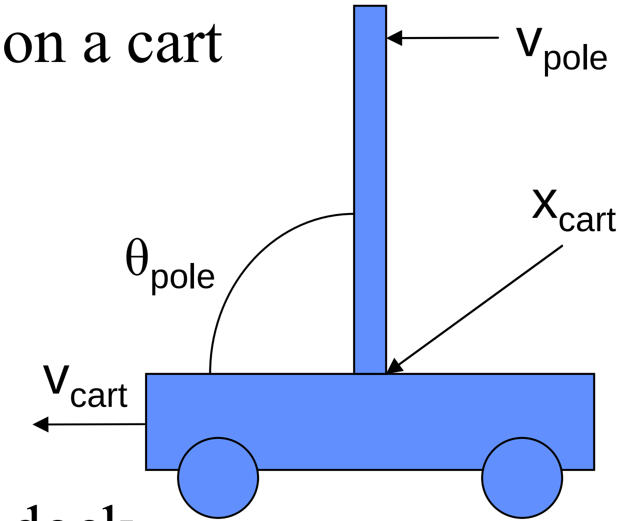
- Example
  - Elman net
    - Partially recurrent
    - Context units keep *internal memory of past inputs*
    - *Fixed* context weights
    - Backpropagation for learning
    - E.g. Can disambiguate A→B→C and C→B→A

Elman network

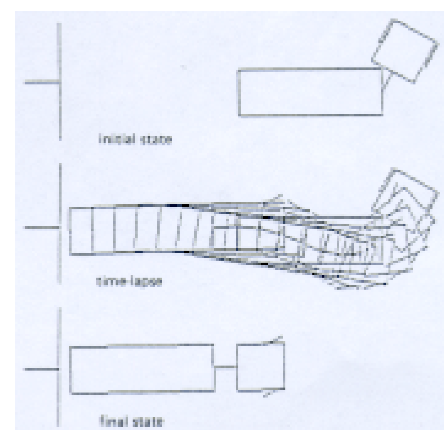


Demos (by Keith Grochow, CSE 599, 2001)

- 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 $\theta_{steering}$



Next Class: Guest lecture by Mike Shadlen

- Things to do:
  - Read Chapter 9
  - Finish Last Homework (due Wed, June 3)
  - Work on mini-project