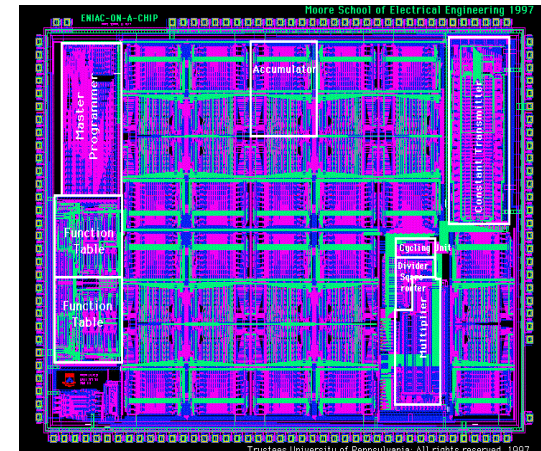
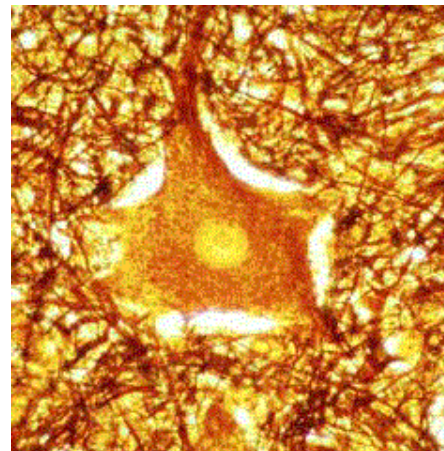
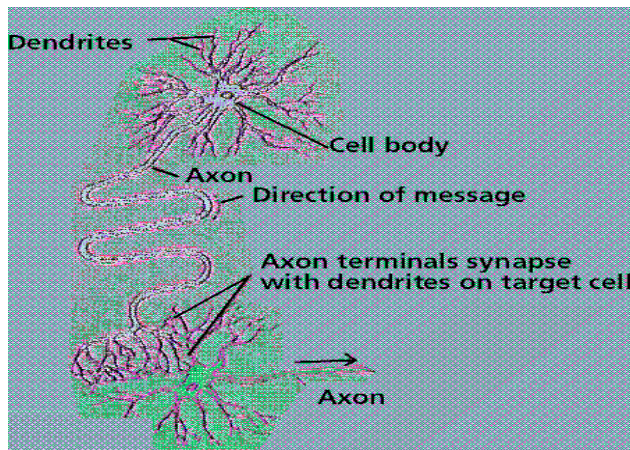


Review of Decision Tree Classifiers

- Basic algorithm is fine, but applications in the real world require further enhancements
 - Rare events
 - Concept drift
 - Sequential data
 - Continuous dependent variables
 - Multiple dependent variables

Biological Inspiration

- Brain consists of billions of switches called neurons, wired up in a complicated way
- Computers consists of many switches (transistors)



Why Model The Brain

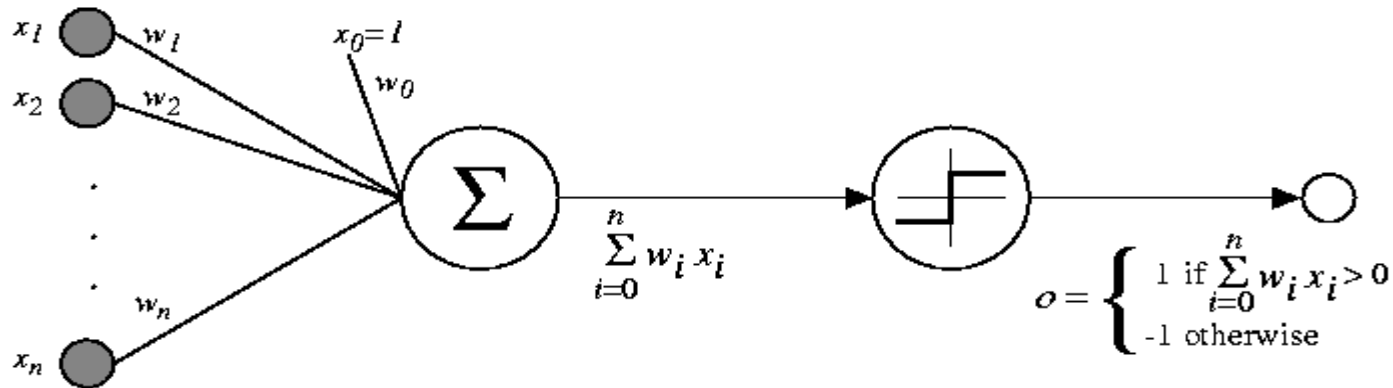
Consider humans:

- Neuron switching time $\sim .001$ second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{4-5}$
- Scene recognition time $\sim .1$ second
- 100 inference steps doesn't seem like enough

→ much parallel computation

- Computer switch at speeds of 10^{-11}
- Sub-symbolic learning
- Koch, January 16, 1997 Nature.

Simplest Type of Unit - Perceptron

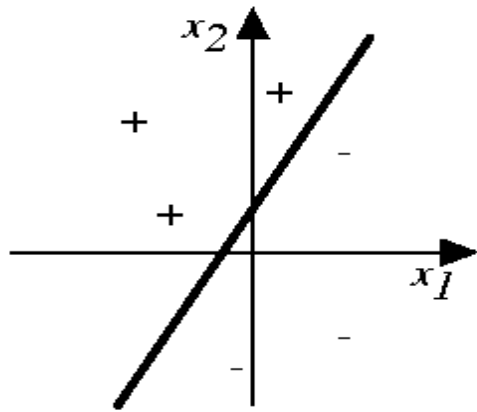


$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

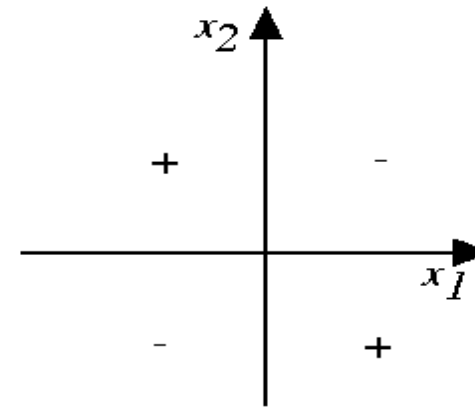
Sometimes we'll use simpler vector notation:

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

Perceptron Decision Boundaries



(a)



(b)

Represents some useful functions

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

What function?
Minsky and Papert

But some functions not representable

- e.g., not linearly separable

Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

Can prove it will converge

- If training data is linearly separable
- and η sufficiently small

Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta(t - o)x_i$$

- Learning the AND function, rate = 0.05

w0	w1	w2	x0	x1	x2	t	w0,x0	w1,x1	w2,x2	o	delta_w0	delta_w1	delta_w2
0	0.2	0.3	1	1	1	1	0	0.2	0.3	1	0	0	0
0	0.2	0.3	1	1	0	-1	0	0.2	0	1	-0.1	-0.1	0
-0.1	0.1	0.3	1	0	1	-1	-0.1	0	0.3	1	-0.1	0	-0.1
-0.2	0.1	0.2	1	0	0	-1	-0.2	0	0	-1	0	0	0
-0.2	0.1	0.2	1	1	1	1	-0.2	0.1	0.2	1	0	0	0
-0.2	0.1	0.2	1	1	0	-1	-0.2	0.1	0	-1	0	0	0
-0.2	0.1	0.2	1	0	1	-1	-0.2	0	0.2	-1	0	0	0
-0.2	0.1	0.2	1	0	0	-1	-0.2	0	0	-1	0	0	0

Linear Units – No Threshold

- Gradient Descent (Delta Rule) (update weights after looking at all training data)

- * For each linear unit weight w_i , Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

- For each linear unit weight w_i , Do

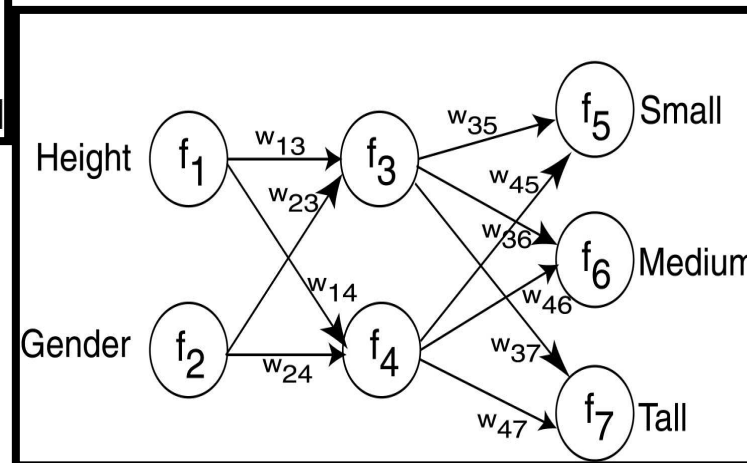
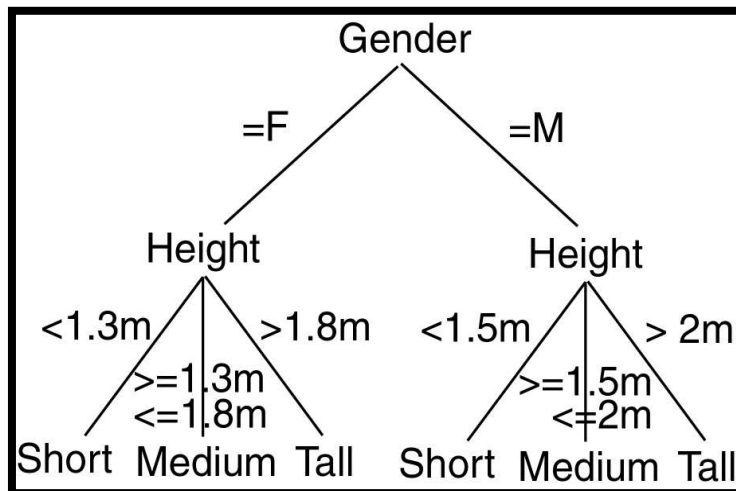
$$w_i \leftarrow w_i + \Delta w_i$$

- Stochastic Gradient Descent (update weights after looking at each instance)

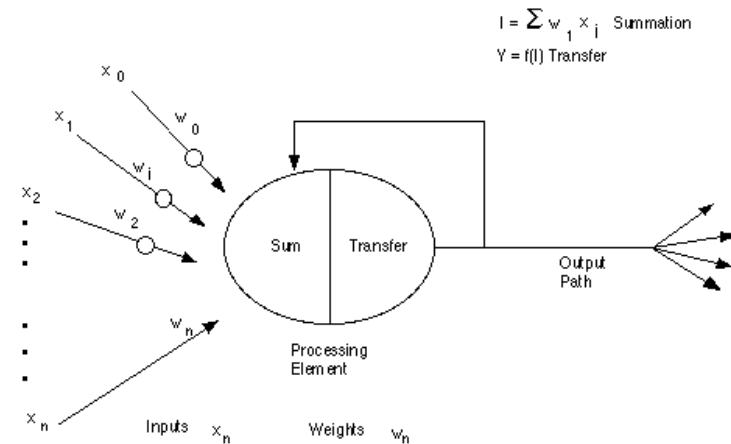
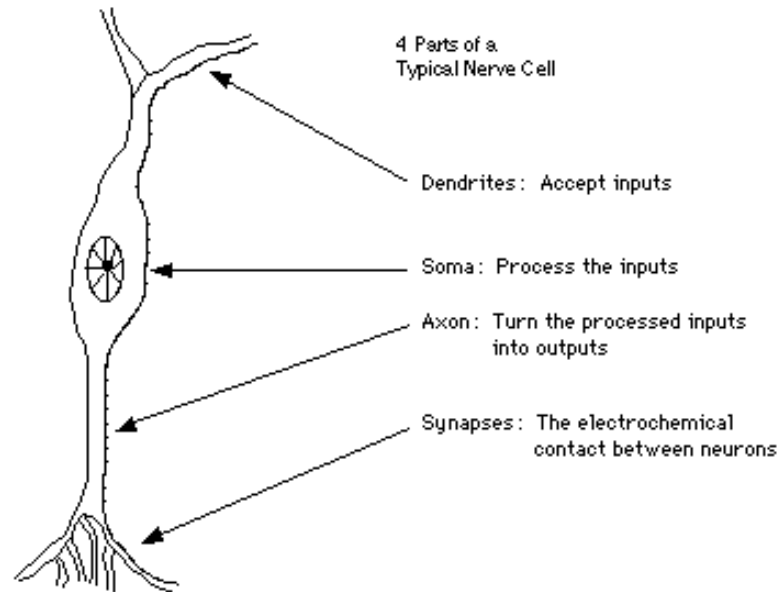
Classification Using Neural Networks

- Typical NN structure for classification:
 - One output node per class
 - Output value is class membership function value
- Supervised learning
- For each tuple in training set, propagate it through NN. Adjust weights on edges to improve future classification.
- Algorithms: Propagation, Backpropagation, Gradient Descent

Decision Tree vs. Neural Network



Network of Neurons



Four Key Decisions To Make

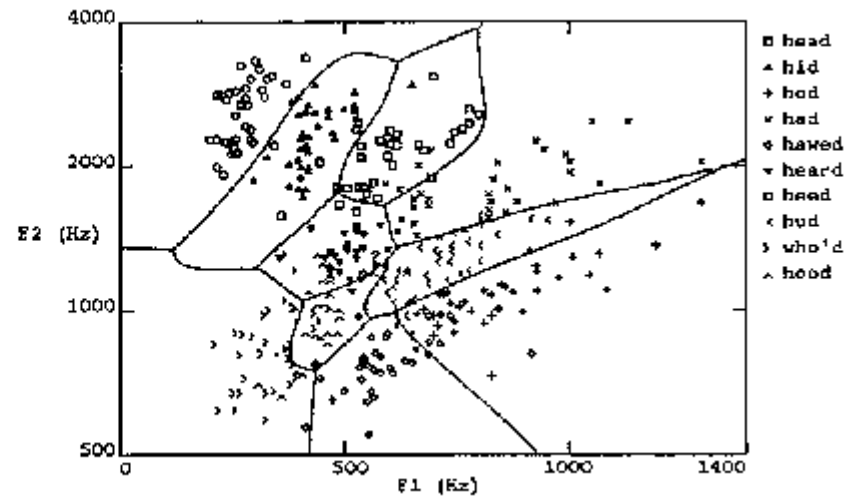
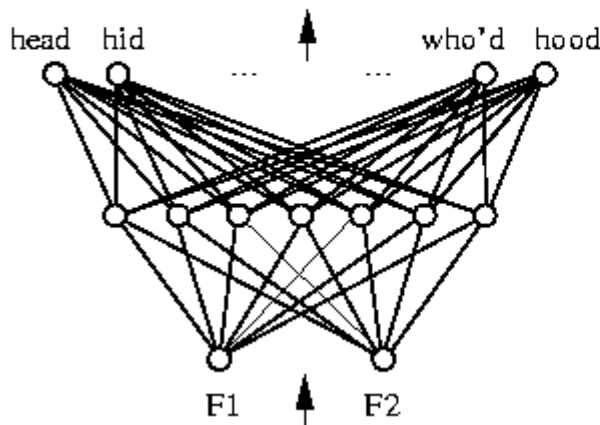
- Arrange neurons in various layers.
- Deciding the type of connections among neurons for different layers, as well as among the neurons within a layer.
- Deciding the way a neuron receives input and produces output.
- Determining the strength of connection within the network

Layers and Connections?

- Layers
 - How many input nodes, hidden units, hidden layers, output units.
 - What happens if you have too many hidden units?
- Connections
 - Uni (Hierarchical) or bi-directional (resonance) between neurons
 - Connect to units in other layers or within a layer (re-current: form cliques)
 - Full or partial connections between layers

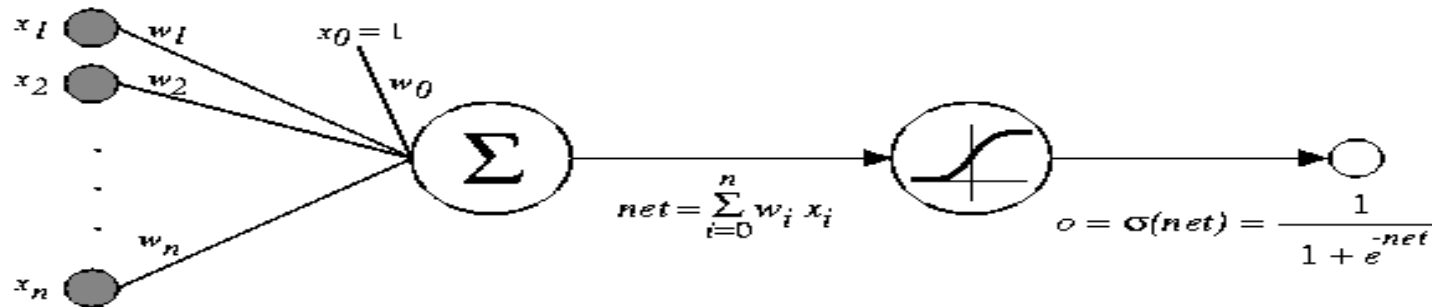
Training a Network of Neurons

- Use the backpropagation algorithm
 - Gradient descent (can get stuck in local minima)
 - Error is summed over all outputs
 - Network of neurons allows complex decision boundaries. Input layer not neurons.



What Type of Neuron to Use?

- Linear units? Perceptrons? Use sigmoid, tanh



$\sigma(x)$ is the sigmoid function

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

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

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units →
Backpropagation

Error Gradient For Sigmoid Function

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\ &= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}\end{aligned}$$

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Backpropagation Algorithm

Initialize all weights to small random numbers.
Until satisfied, Do

• For each training example, Do

1. Input the training example to the network
and compute the network outputs

2. For each output unit k

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

3. For each hidden unit h

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

4. Update each network weight $w_{i,j}$

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

where

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

What's in a name?

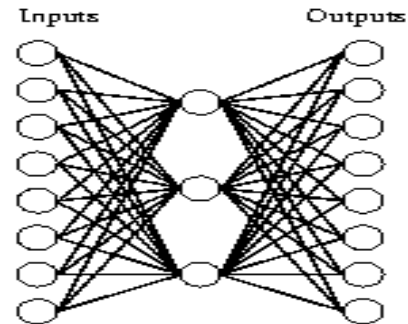
Feedforward,
Two layers of sigmoid
Units

Stochastic gradient
descent

Insights into Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
 - In practice, often works well (can run multiple times)
- Often include weight *momentum* α
$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n - 1)$$
- Minimizes error over *training* examples
 - Will it generalize well to subsequent examples?
- Training can take thousands of iterations → slow!
- Using network after training is very fast

Hidden Layer and Latent Concepts - 1



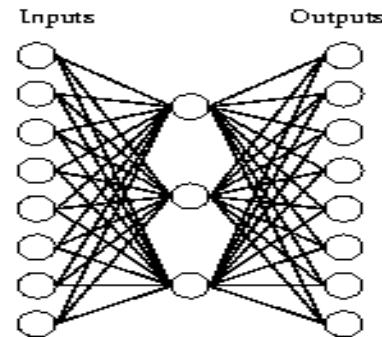
A target function:

Input	→	Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned??

Hidden Layer and Latent Concepts - 2

A network:



Learned hidden layer representation:

Input		Hidden Values		Output
10000000	→	.89 .04 .08	→	10000000
01000000	→	.01 .11 .88	→	01000000
00100000	→	.01 .97 .27	→	00100000
00010000	→	.99 .97 .71	→	00010000
00001000	→	.03 .05 .02	→	00001000
00000100	→	.22 .99 .99	→	00000100
00000010	→	.80 .01 .98	→	00000010
00000001	→	.60 .94 .01	→	00000001