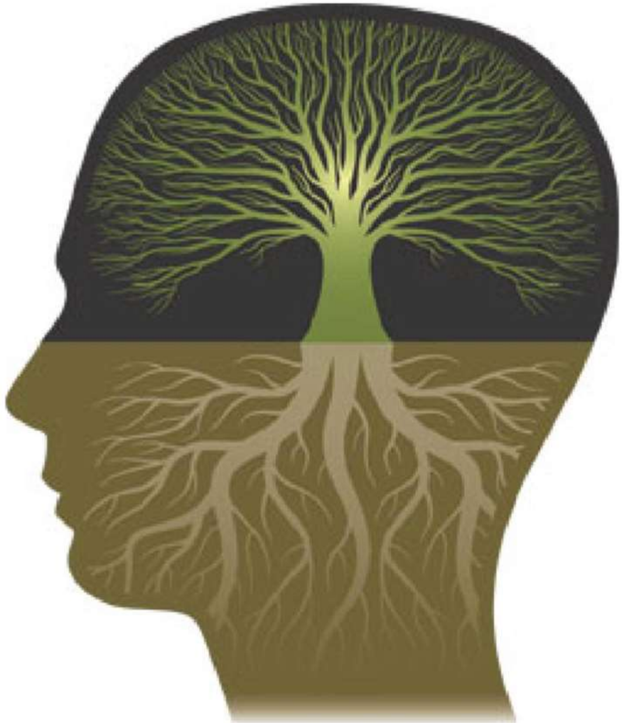
Neural Networks

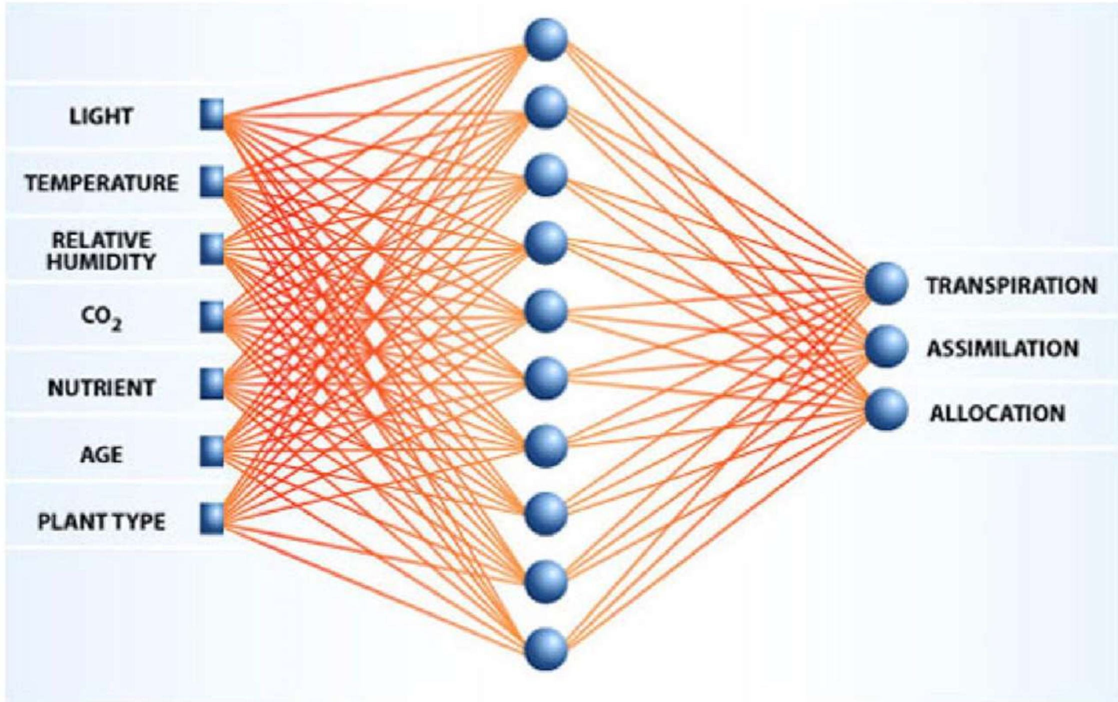
Perceptron

1943 M–P model
194* Hebbian learning model

- The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt.
- Perceptron is an algorithm for supervised classification.
- It is a type of linear classifier.
- It lays the foundation of artificial neural networks (ANN).

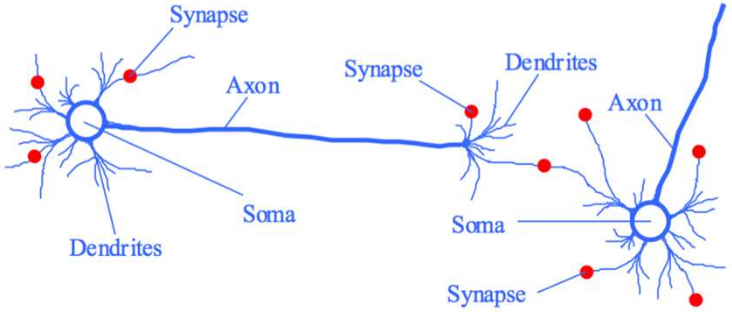
Inspired from Neural Networks

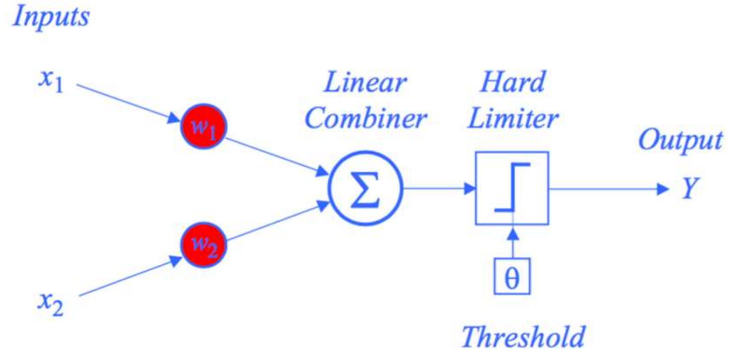




Perceptron and Neural Nets

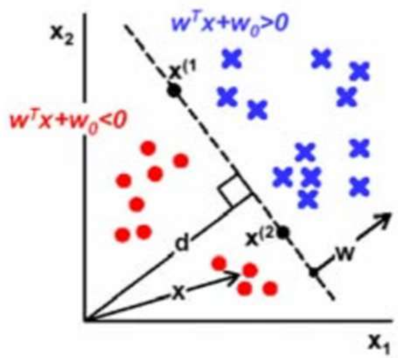
- From biological neuron to artificial neuron (perceptron)



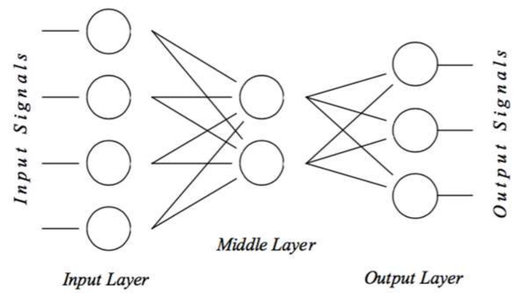


- Generative Model

$$X = \sum_{i=1}^{n} x_i w_i \qquad y = \begin{cases} +1, if X > w_0 \\ -1, if X < w_0 \end{cases}$$

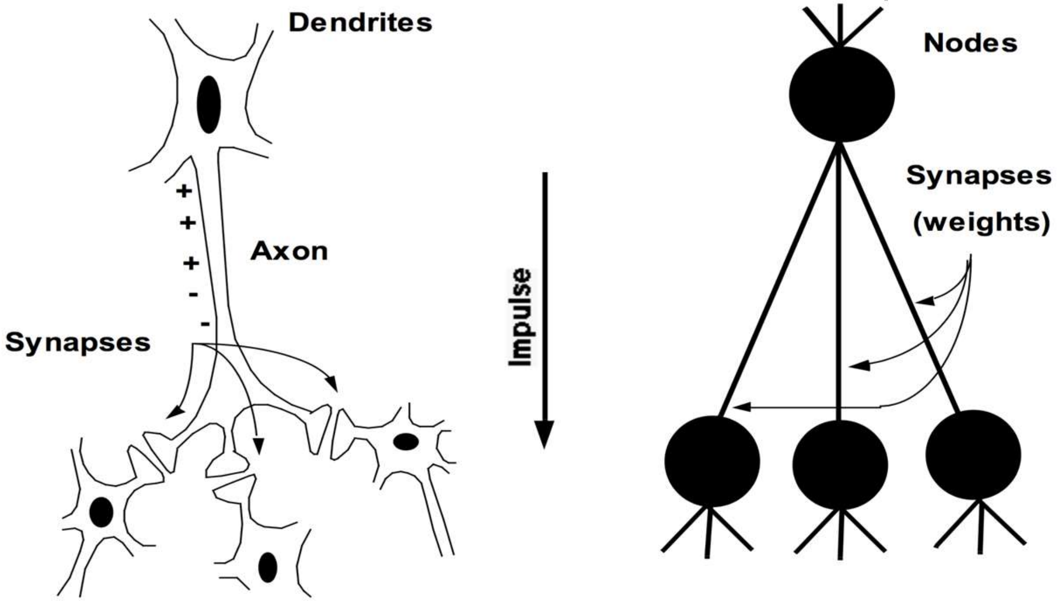


- Artificial neuron networks
  - supervised learning
  - gradient descent



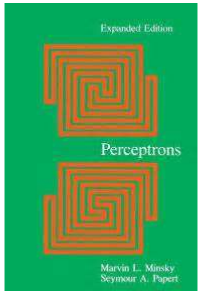
Connectionist Models

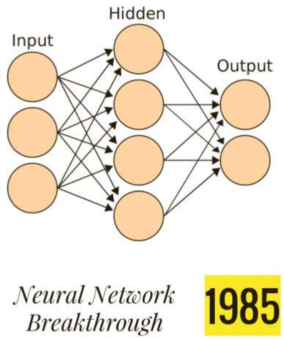
- Consider humans:
  - Neuron switching time ~ 0.001 second
  - Number of neurons ~ 10^{10}
  - Connections per neuron ~ 10^{4-5}
  - Scene recognition time ~ 0.1 second
  - 100 inference steps doesn't seem like enough
    → much parallel computation
- Properties of artificial neural nets (ANN)
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed processes



History of Neural Networks

- The Beginnings (1940s): M-P model, Hebbian learning theory,…
- Golden Years (1958~1969): The perceptron algorithm, Adaline,…
- Winter (1969~1980): Minsky's devastating criticism of perceptrons

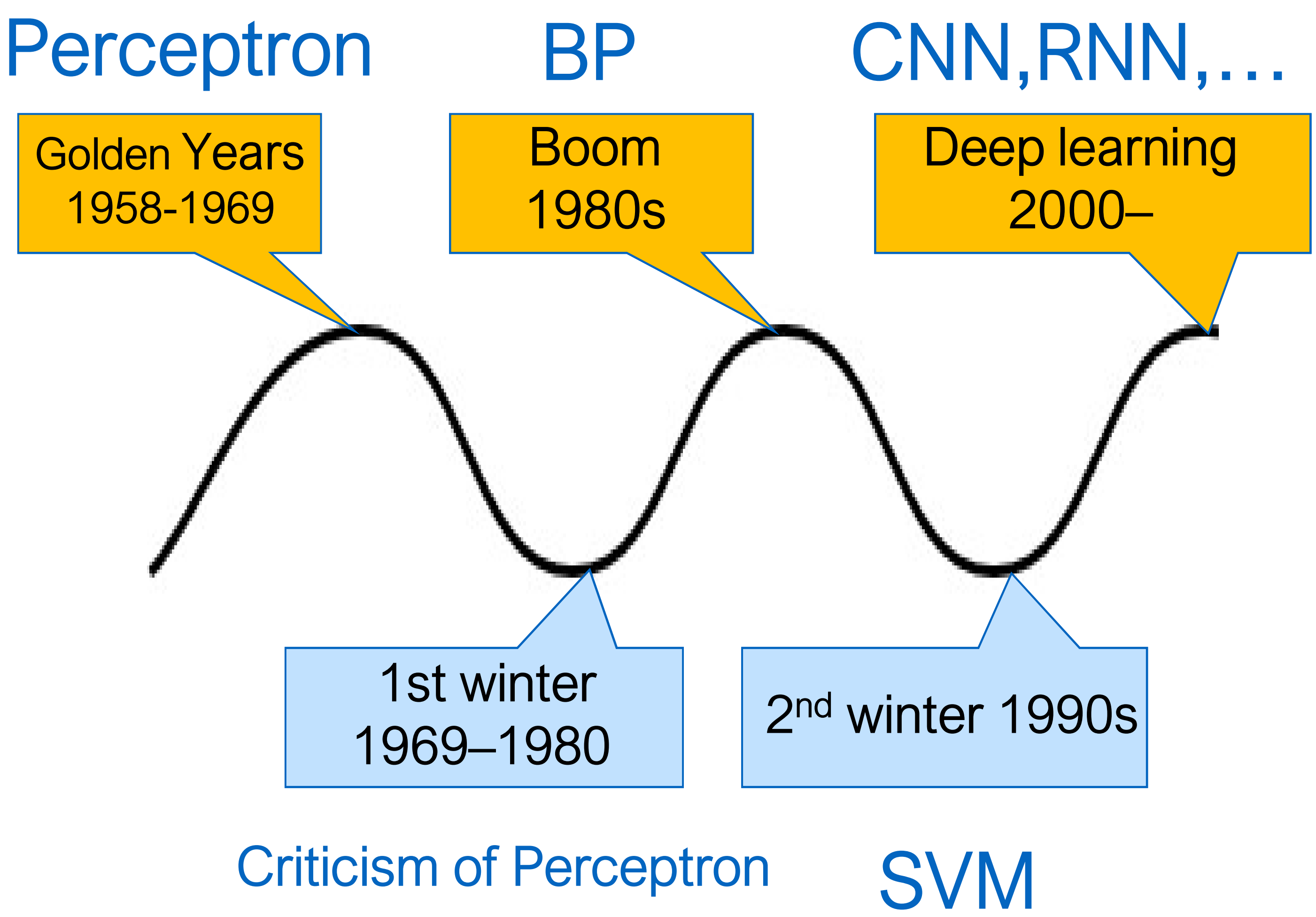




- Boom (1980s): Hopfield net, Back Propagation algorithm
- Winter (1990s): Statistical learning theory, SVM

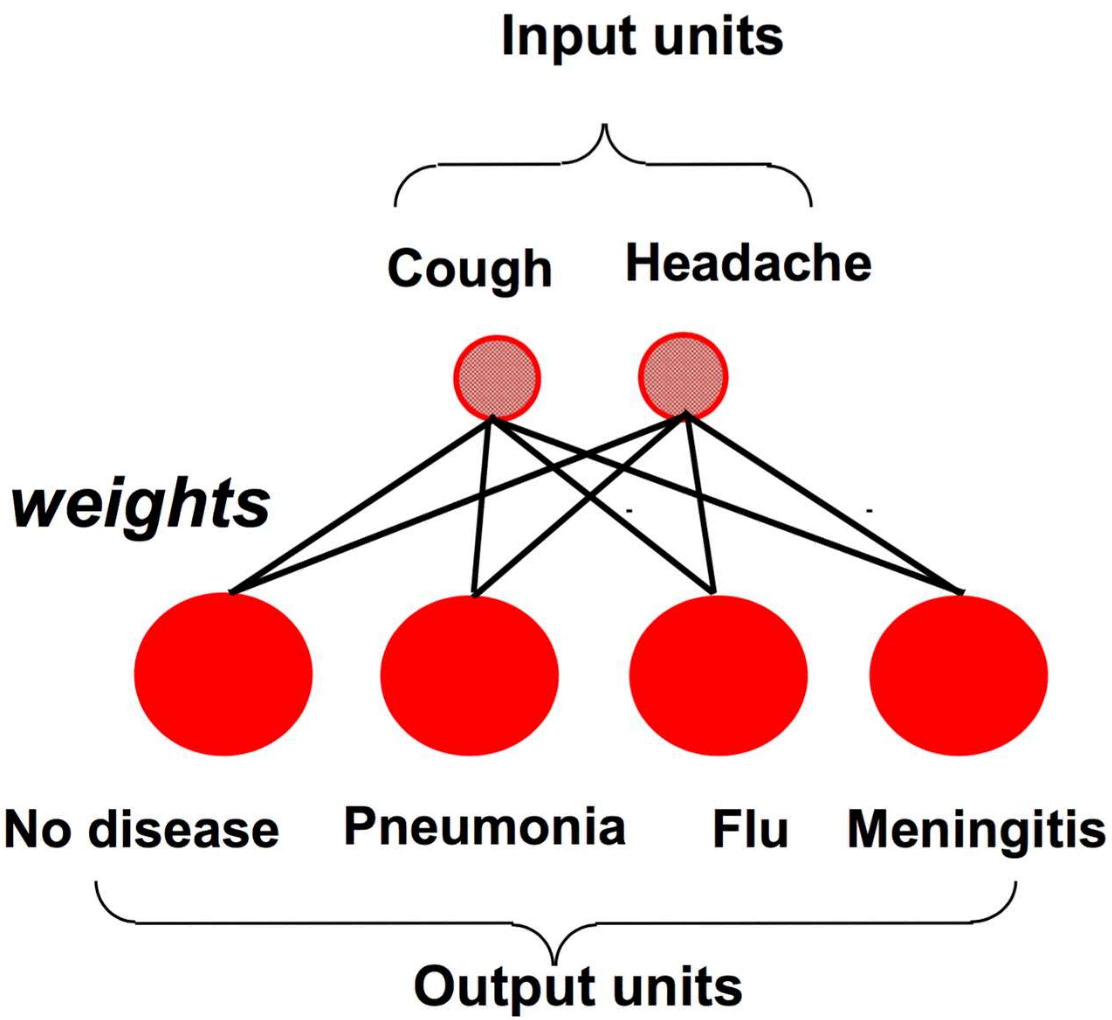
  In the 1990’s, many researchers abandoned neural because SVMs worked better, and there was no successful attempts to train deep networks.
- The 3rd rise of NN (2000-present): Deep learning

Time Line

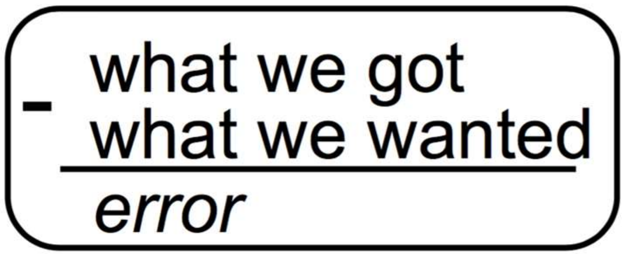


Perceptron
BP
CNN,RNN,…
Golden Years 1958-1969
Boom 1980s
Deep learning 2000–
1st winter 1969–1980
2nd winter 1990s
Criticism of Perceptron
SVM

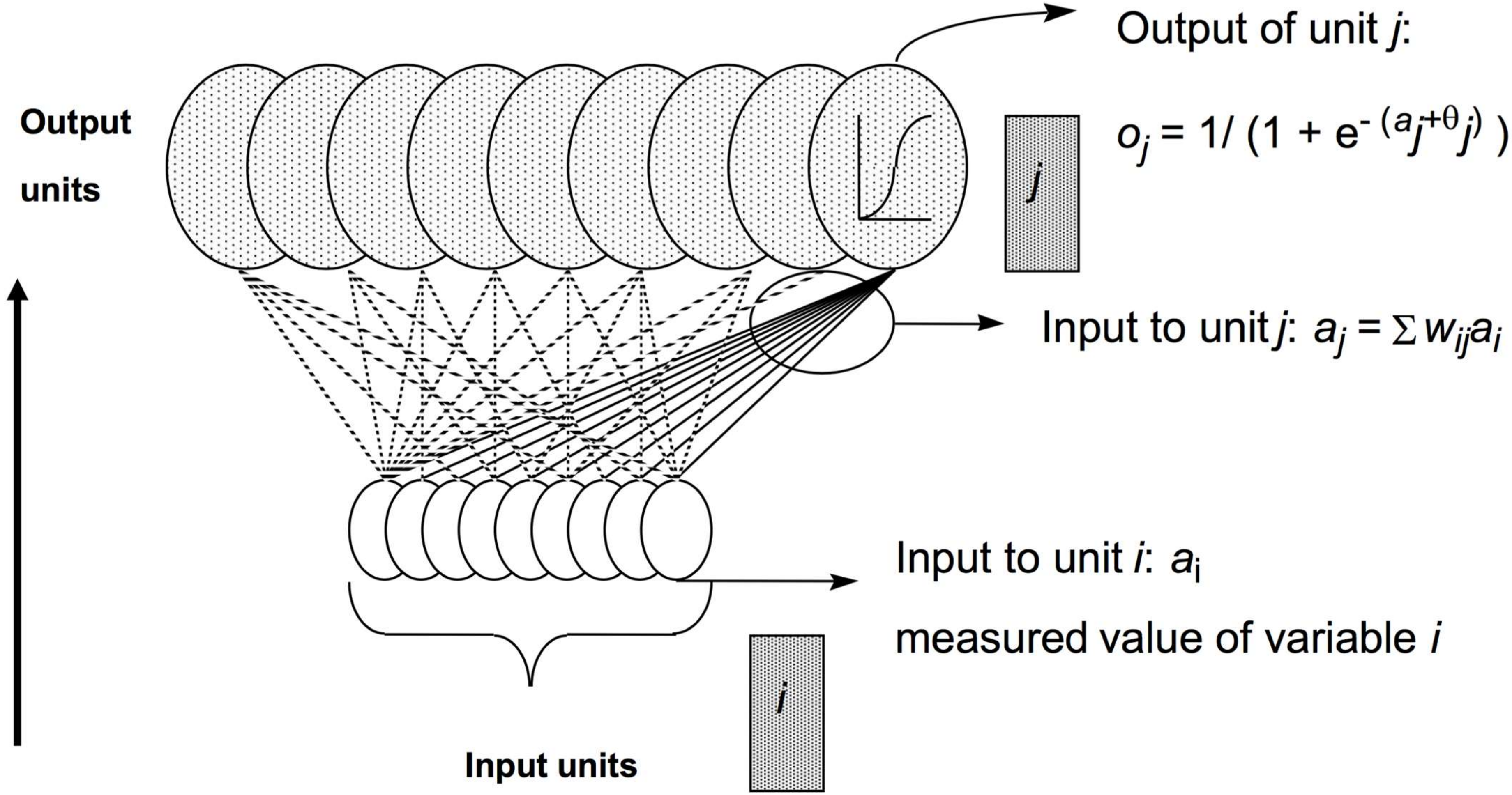
Perceptrons



Δ rule
change weights to decrease the error



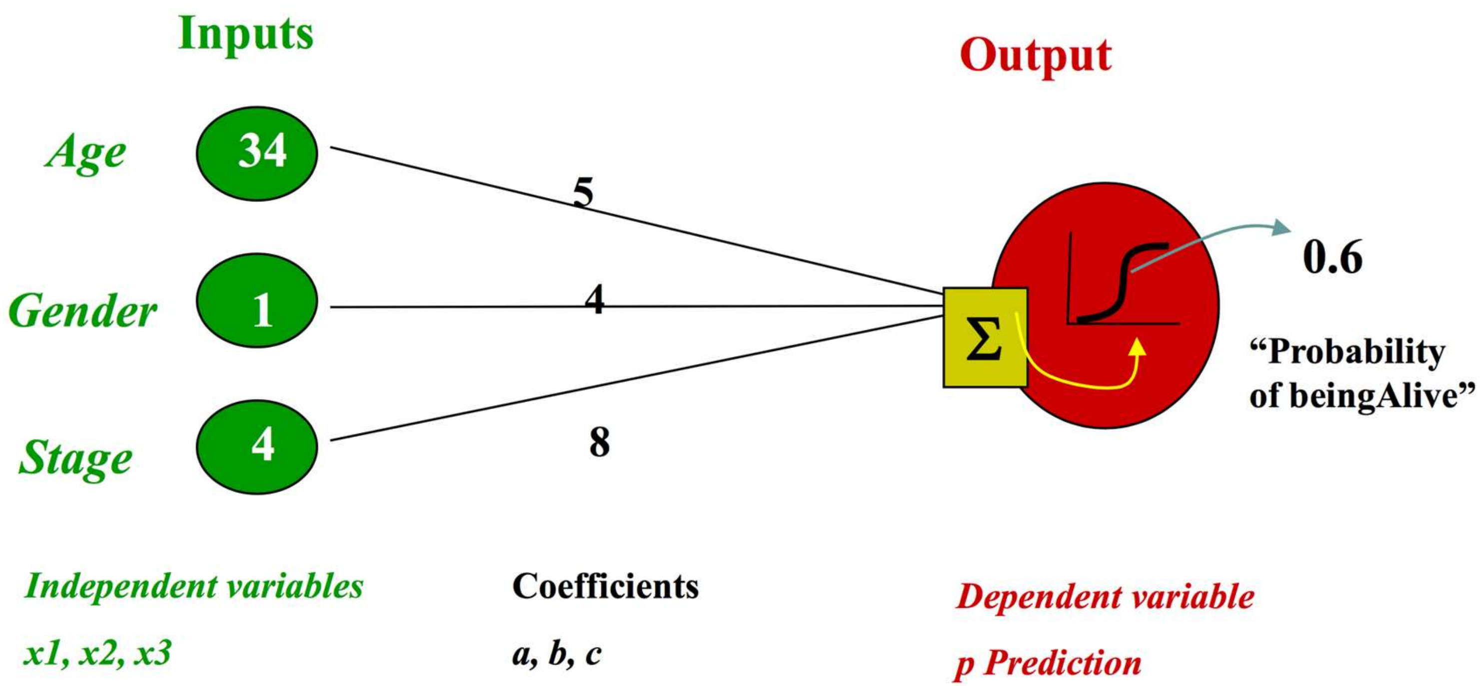
Perceptrons



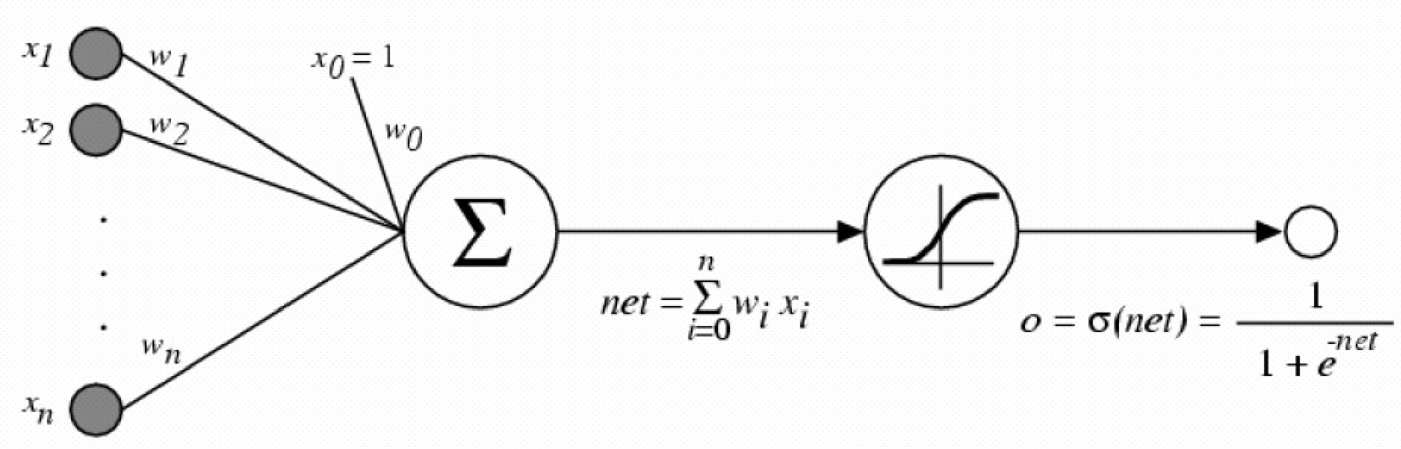
Jargon Pseudo-Correspondence

- Independent variable = input variable
- Dependent variable = output variable
- Coefficients = “weights”
- Estimates = “targets”

Logistic Regression Model (the sigmoid unit)



The perceptron learning algorithm

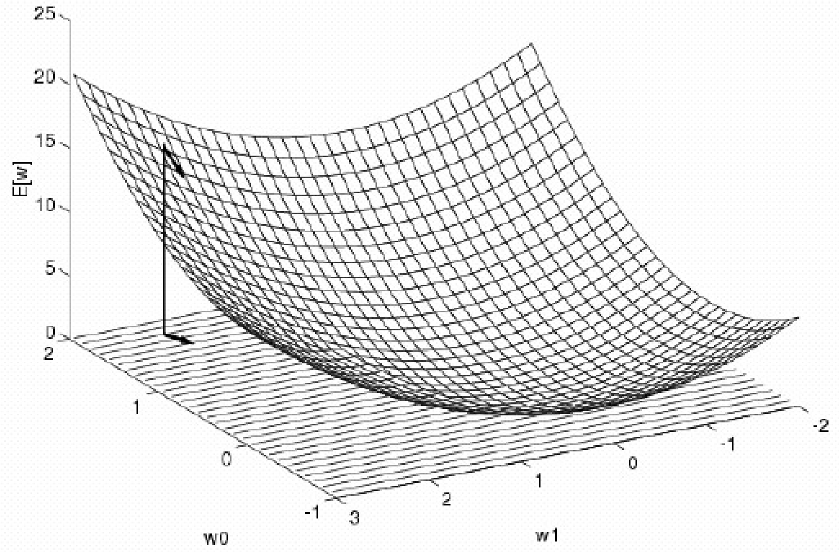


- Recall the nice property of sigmoid function $\dfrac{d\sigma}{dt} = \sigma(1-\sigma)$
- Consider regression problem f:X→Y , for scalar Y: $y = f(x) + \epsilon$
- Let's maximize the conditional data likelihood

$$\vec{w} = \arg\max_{\vec{w}} \ln \prod_i P(y_i | x_i; \vec{w})$$

$$\vec{w} = \arg\min_{\vec{w}} \sum_i \frac{1}{2}(y_i - \hat{f}(x_i; \vec{w}))^2$$

Gradient Descent



Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

x_d = input

t_d = target output

o_d =observed unit output

w_i =weigh i

$$\frac{\partial E[\vec{w}]}{\partial w_j} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum (t_d - o_d)^2$$

$$=$$

Gradient Descent

$$\frac{\partial E_D[\vec{w}])}{\partial w_j} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2$$
$$= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$
$$= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right)$$
$$= -\sum_d (t_d - o_d)\frac{\partial o_d}{\partial net_i}\frac{\partial net_d}{\partial w_i}$$
$$= -\sum_d (t_d - o_d)o_d(1 - o_d)x_d^i$$

x_d = input

t_d = target output

o_d =observed unit output

w_i =weigh i

Batch mode:

Do until converge:

1. compute gradient ∇**E$_D$[w]**

$$\vec{w} = \vec{w} - \eta\nabla E_D[\vec{w}]$$

Incremental mode:

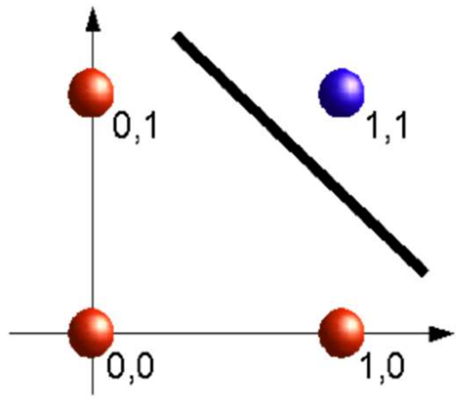
Do until converge:

- For each training example ***d*** in ***D***
  1. compute gradient ∇**E$_d$[w]**
  2. $\vec{w} = \vec{w} - \eta\nabla E_d[\vec{w}]$

  where

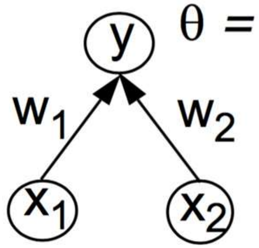
  $$\nabla E_d[\vec{w}] = -(t_d - o_d)o_d(1 - o_d)\vec{x}_d$$

What decision surface does a perceptron define?



NAND

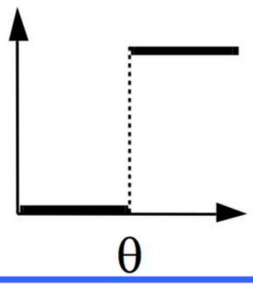
X	Y	Z(color)
0	0	1
0	1	1
1	0	1
1	1	0



$\theta = 0.5$

$$f(x_1 w_1 + x_2 w_2) = y$$

$$f(0w_1 + 0w_2) = 1$$
$$f(0w_1 + 1w_2) = 1$$
$$f(1w_1 + 0w_2) = 1$$
$$f(1w_1 + 1w_2) = 0$$

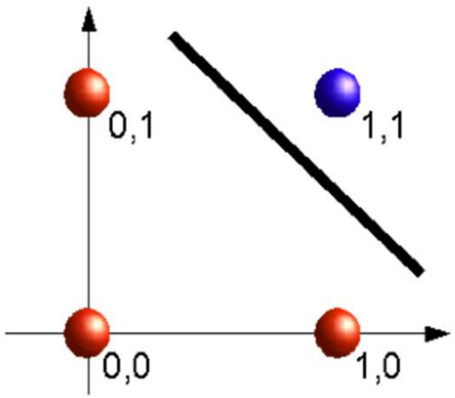


$$f(a) = \begin{cases} 1, \text{ for } a > \theta \\ 0, \text{ for } a \leq \theta \end{cases}$$

some possible values for w_1 and w_2

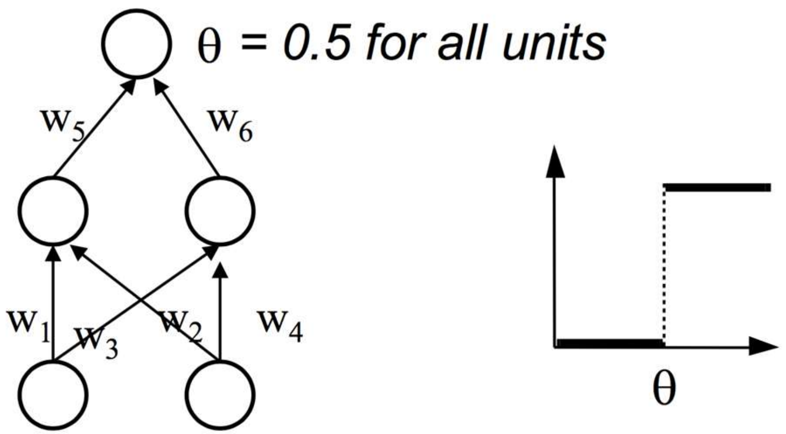
w_1	w_2
0.20	0.35
0.20	0.40
0.25	0.30
0.40	0.20

What decision surface does a perceptron define?



XOR

X	Y	Z(color)
0	0	1
0	1	1
1	0	1
1	1	0



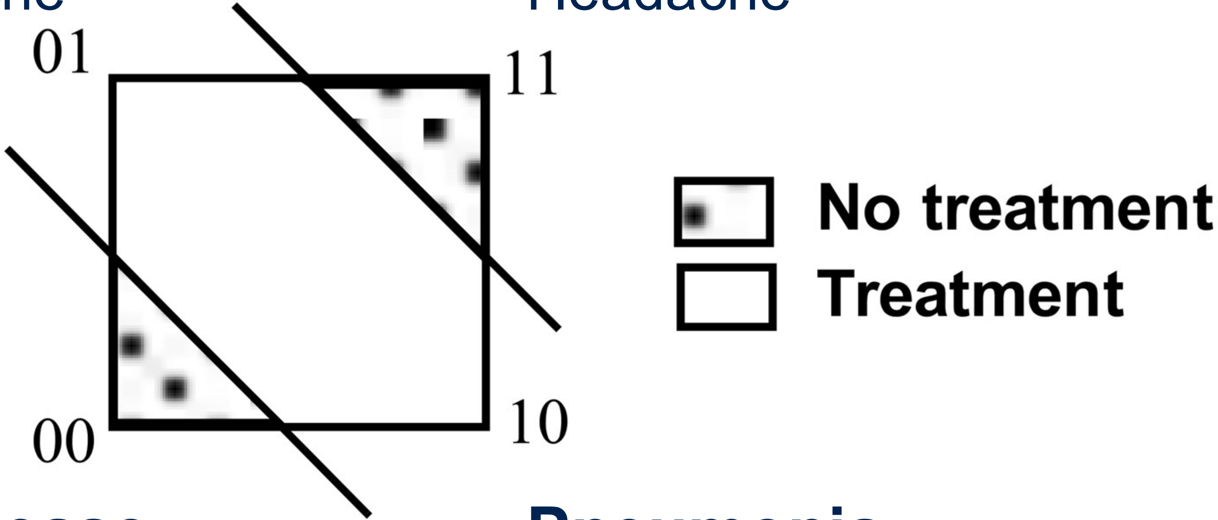
$$f(a) = \begin{cases} 1, \text{ for } a > \theta \\ 0, \text{ for } a \leq \theta \end{cases}$$

a possible set of values for $(w_1, w_2, w_3, w_4, w_5, w_6)$:
(0.6,-0.6,-0.7,0.8,1,1)

Non Linear Separation

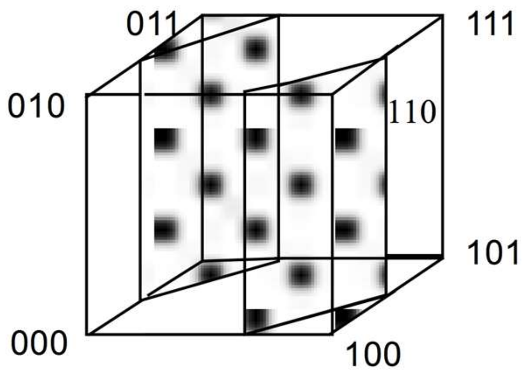
Meningitis
No cough
Headache

Flu
Cough
Headache

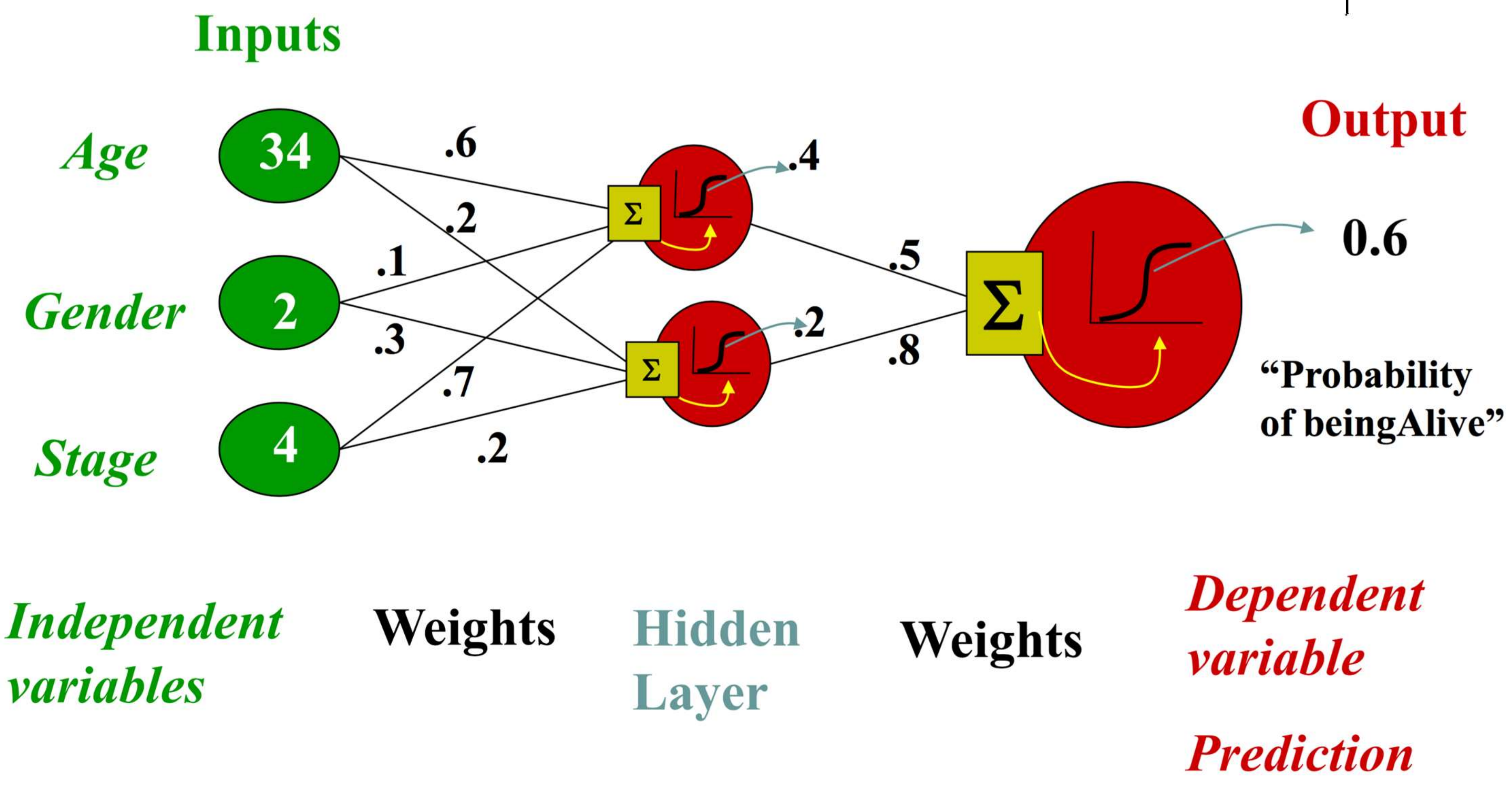


No disease
No cough
No headache

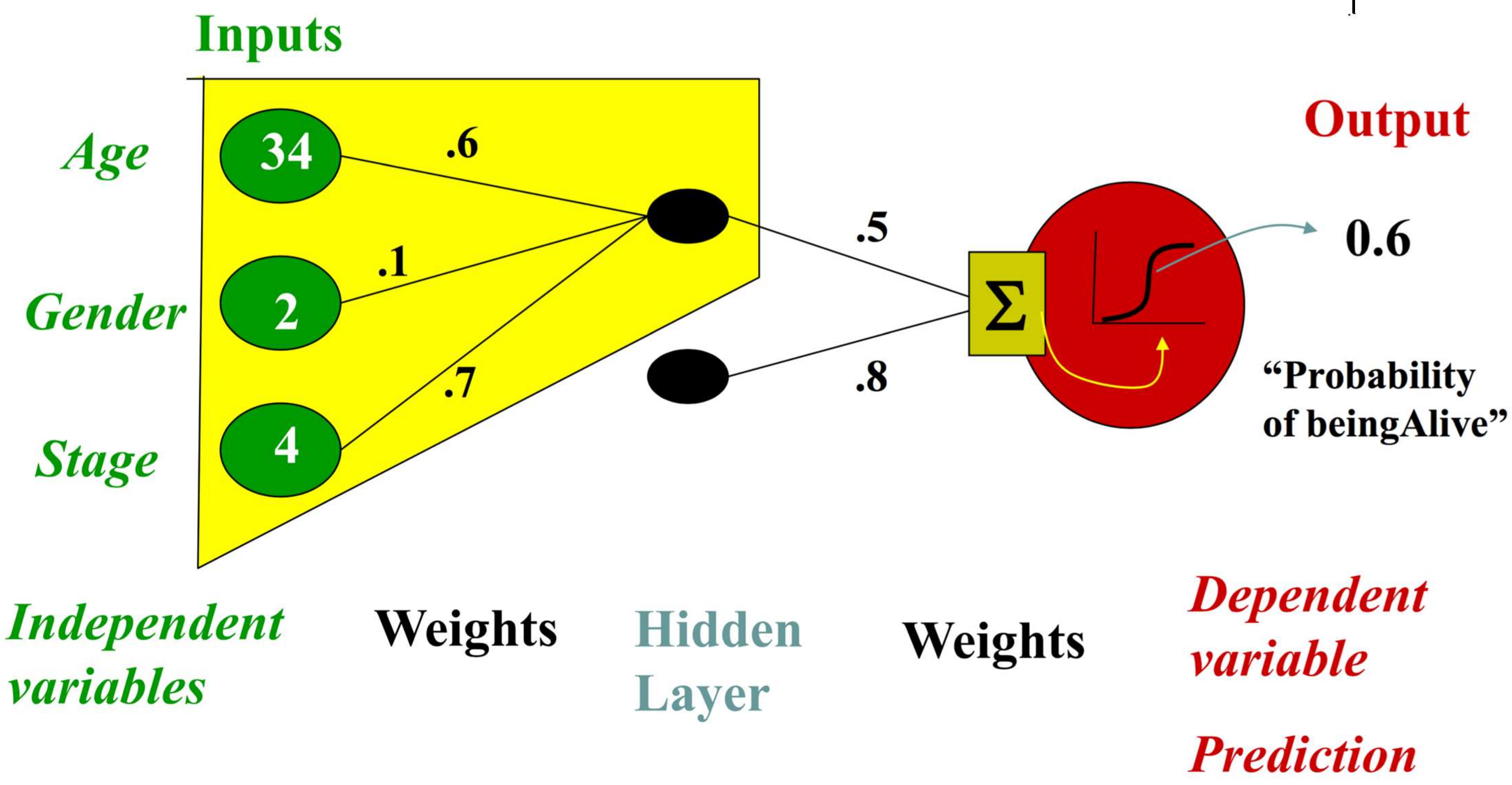
Pneumonia
Cough
No headache



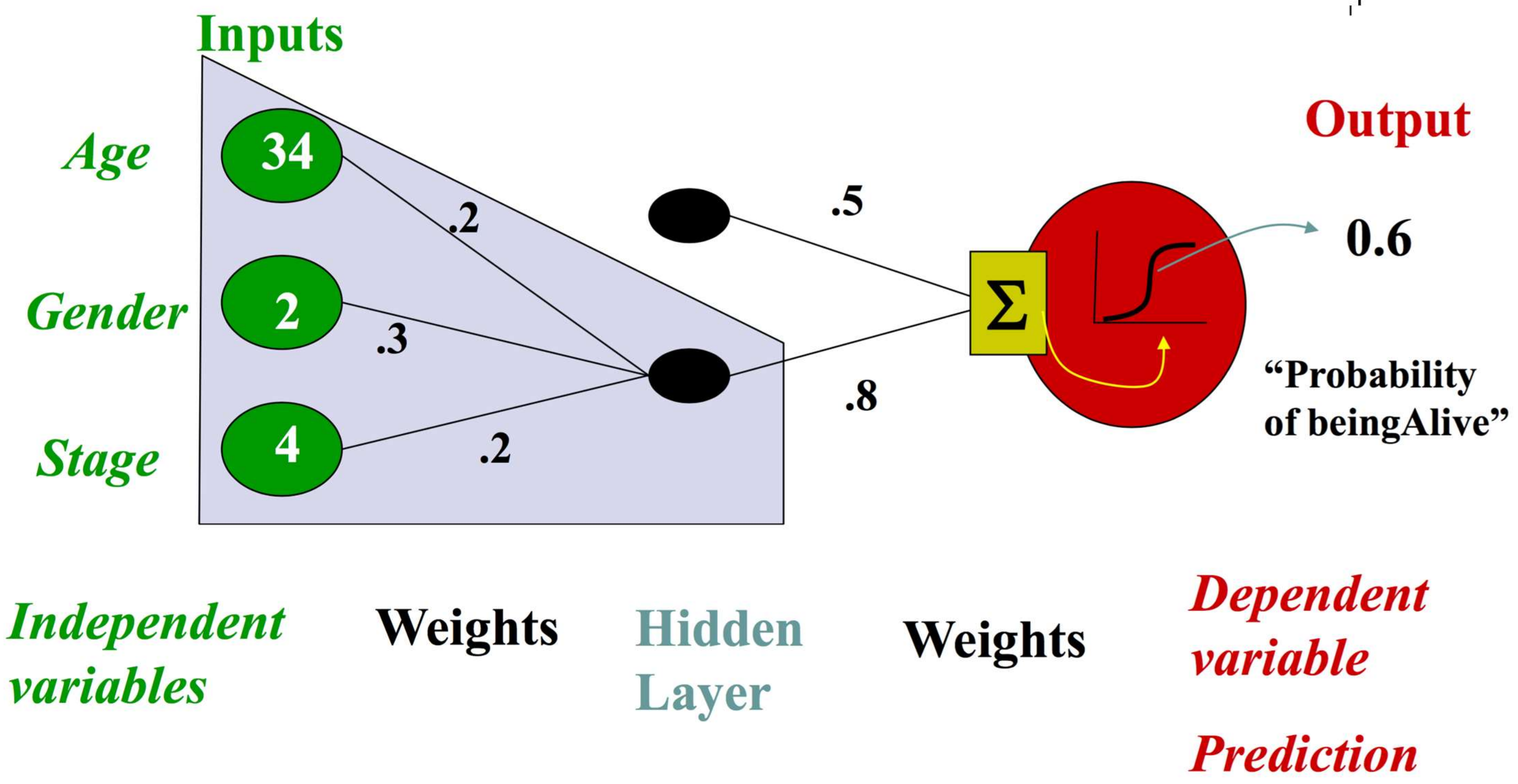
Neural Network Model



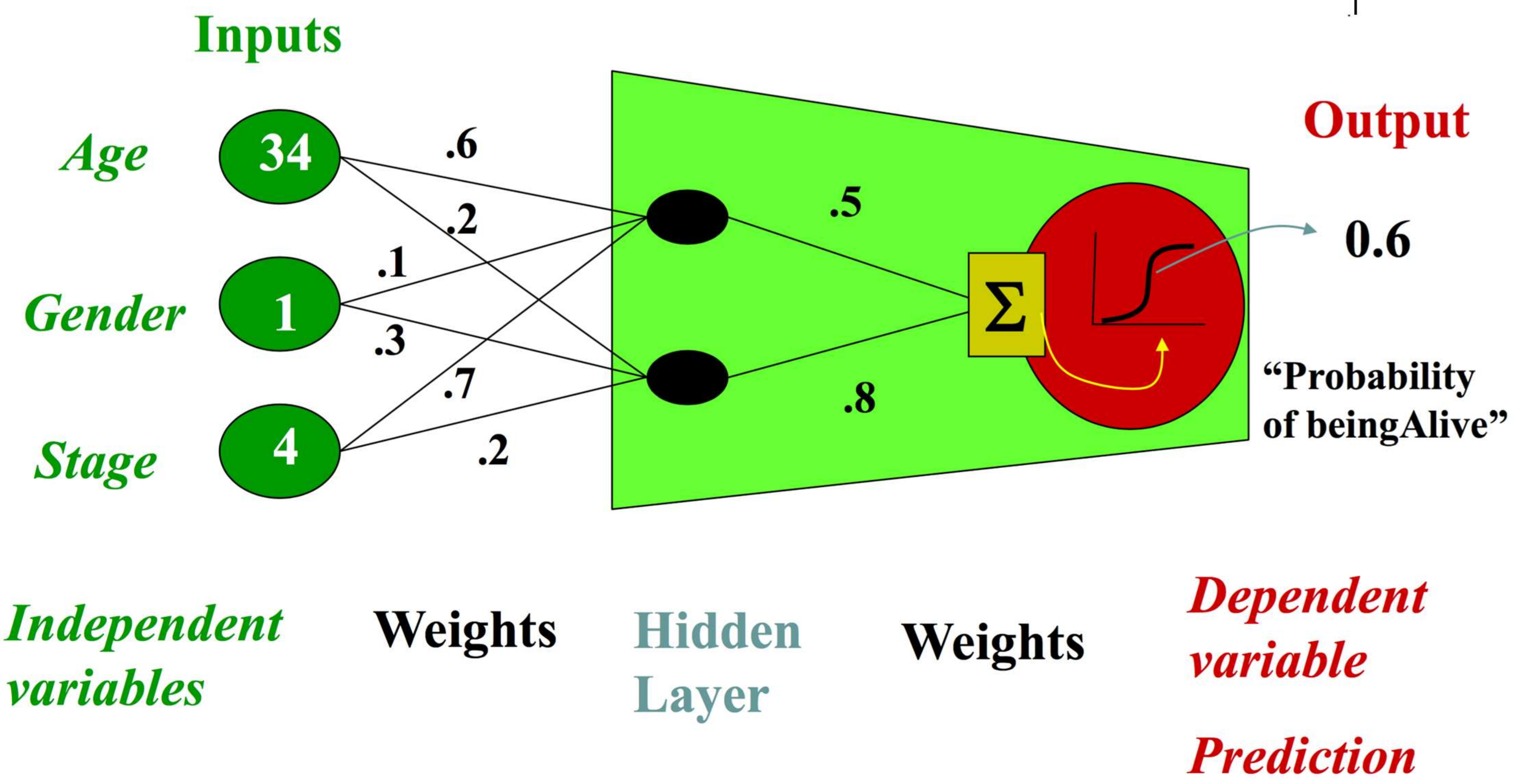
Combined logistic models



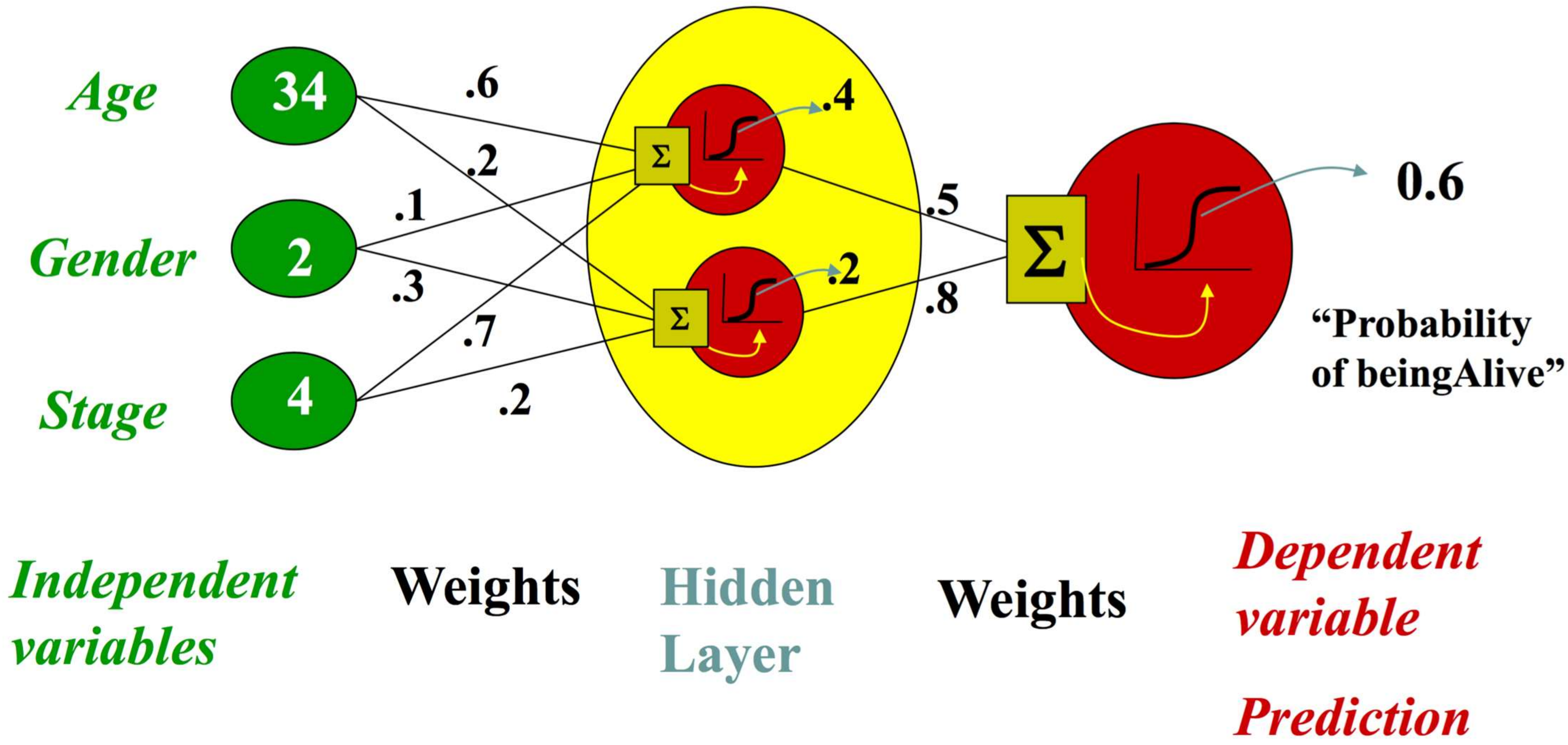
Combined logistic models



Combined logistic models

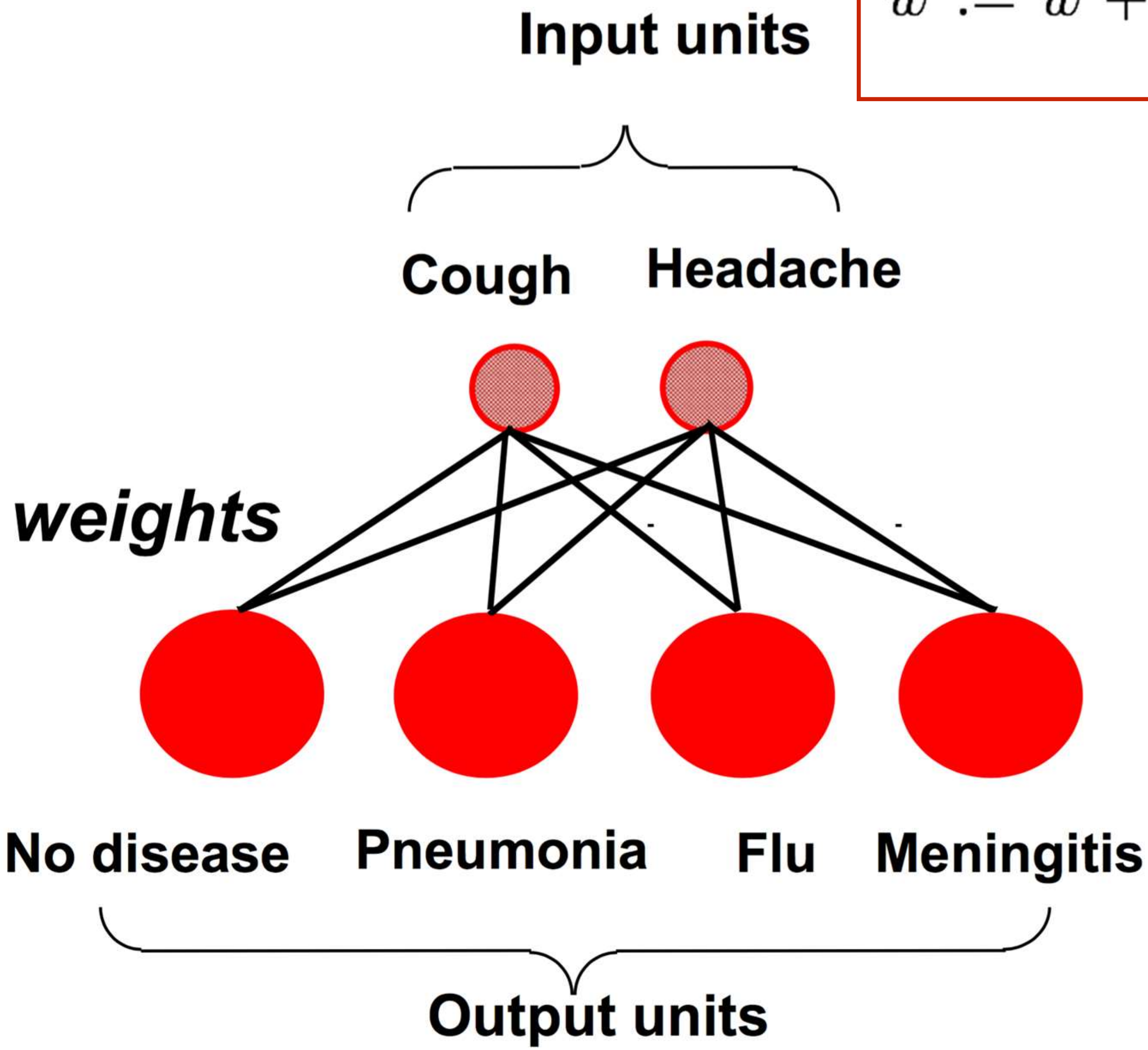


Not really,
no target for hidden units...



Perceptrons

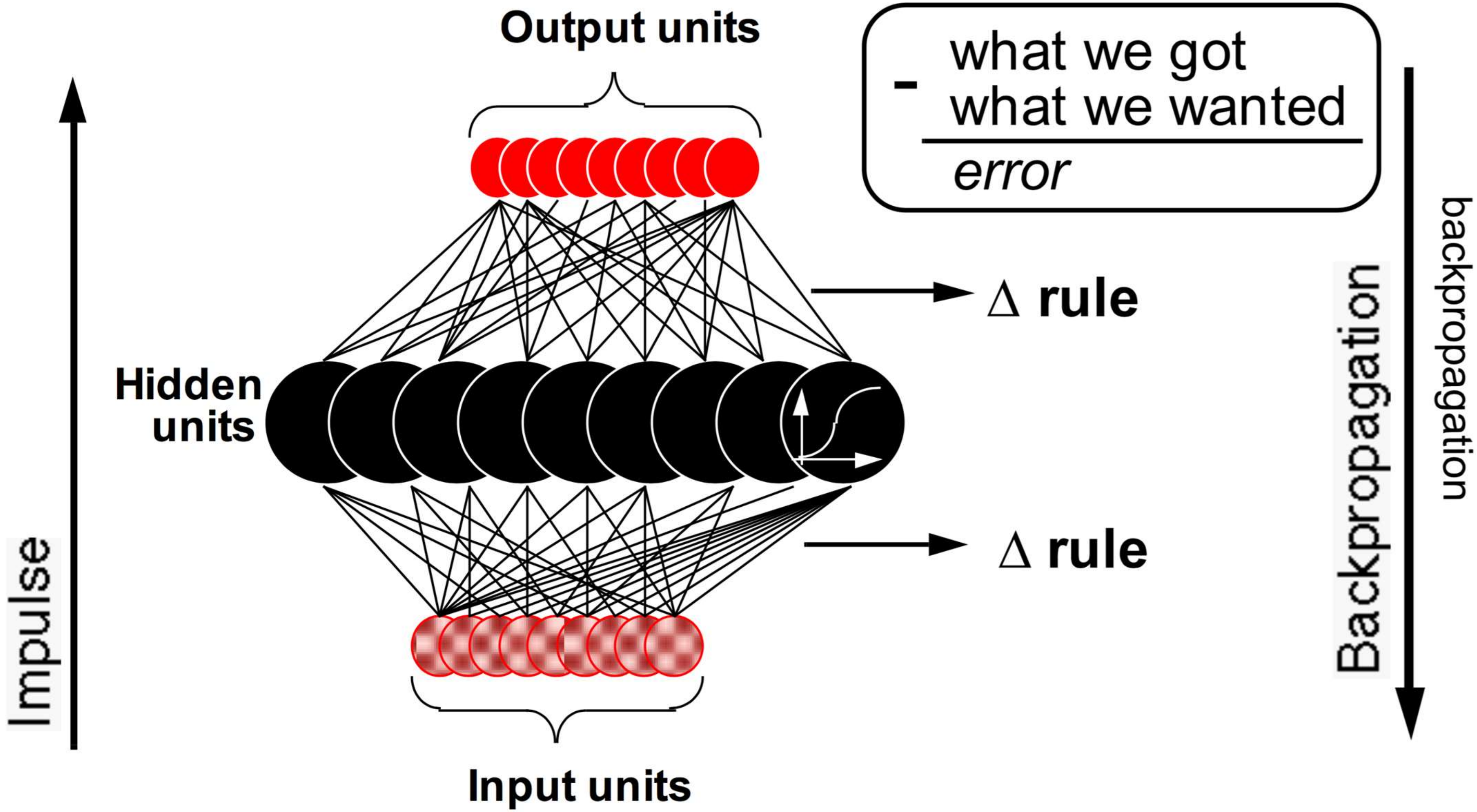
$$\vec{w} := \vec{w} + \eta \sum_d (t_d - o_d) o_d (1 - o_d) \vec{x}_d$$



Δ rule
change weights to decrease the error

what we got
\- what we wanted
error

Hidden Units and Backpropagation



Back Propagation Algorithm

x_d = input

t_d = target output

o_d =observed unit output

w_i =weigh i

- Initialize all weights to small random numbers

  Until convergence, Do

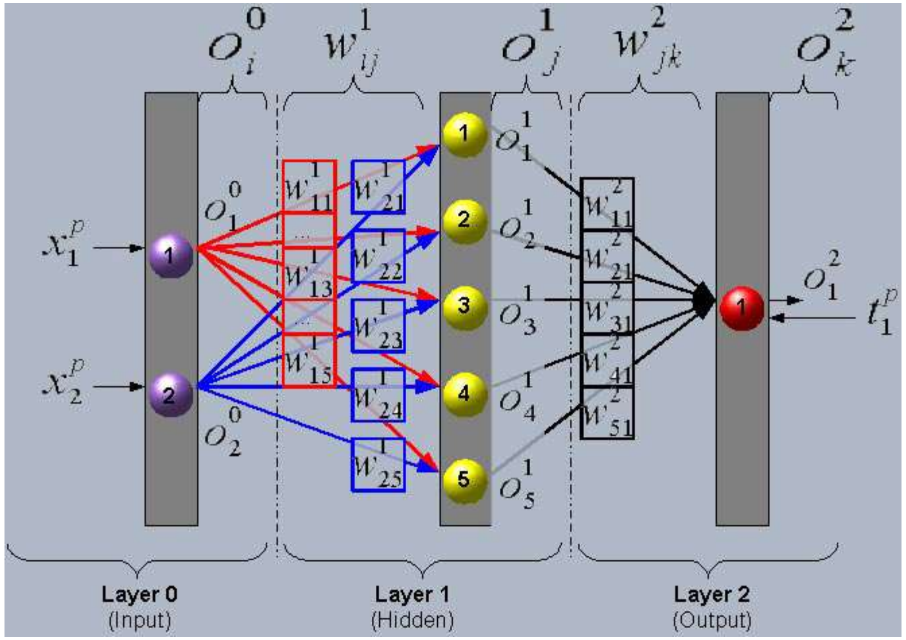
  1. Input the training example to the network and compute the network outputs

  2. For each output unit k

$$\delta_k = o_k^{(2)}(1 - o_k^{(2)})(o_k^{(2)} - t)$$

$$\frac{\partial E}{\partial net_k},$$ net_k is the input of output unit k

$$net_k = \sum_j w_{jk}^{(2)} O_j^{(1)}$$



Back Propagation Algorithm

x_d = input

t_d = target output

o_d =observed unit output

w_i =weigh i

❖ Initialize all weights to small random numbers

Until convergence, Do

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

2. For each output unit k

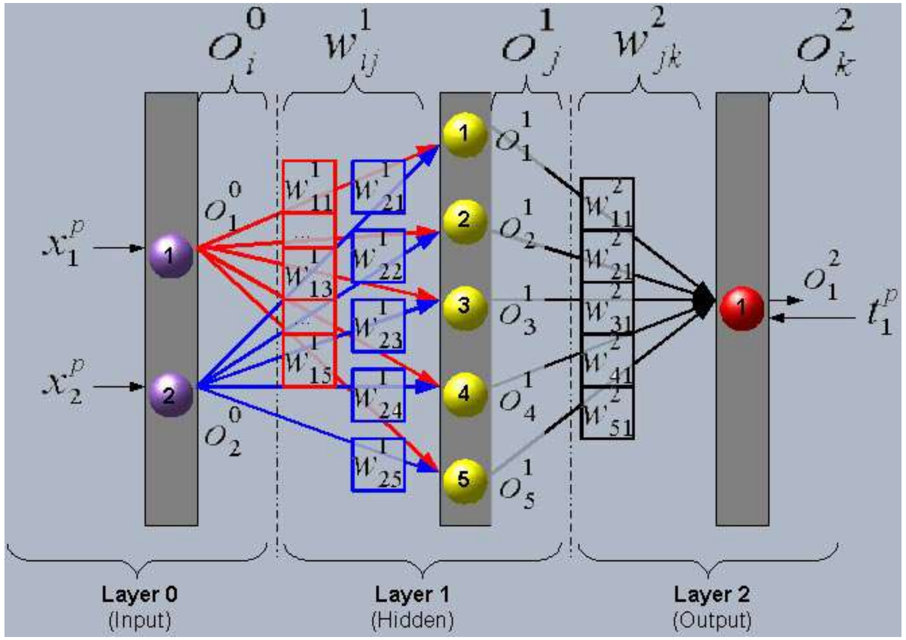
$$\delta_k = o_k^{(2)}(1 - o_k^{(2)})(o_k^{(2)} - t)$$

3. For each hidden unit h

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

$\frac{\partial E}{\partial net_h}$, net_h is the input of hidden unit h

$$net_h = \sum_i w_{ij}^{(1)} O_i^0$$



Back Propagation Algorithm

x_d = input

t_d = target output

o_d =observed unit output

w_i =weigh i

- Initialize all weights to small random numbers

  Until convergence, Do

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

  2. For each output unit k

$$\delta_k = o_k^{(2)}(1 - o_k^{(2)})(o_k^{(2)} - t)$$

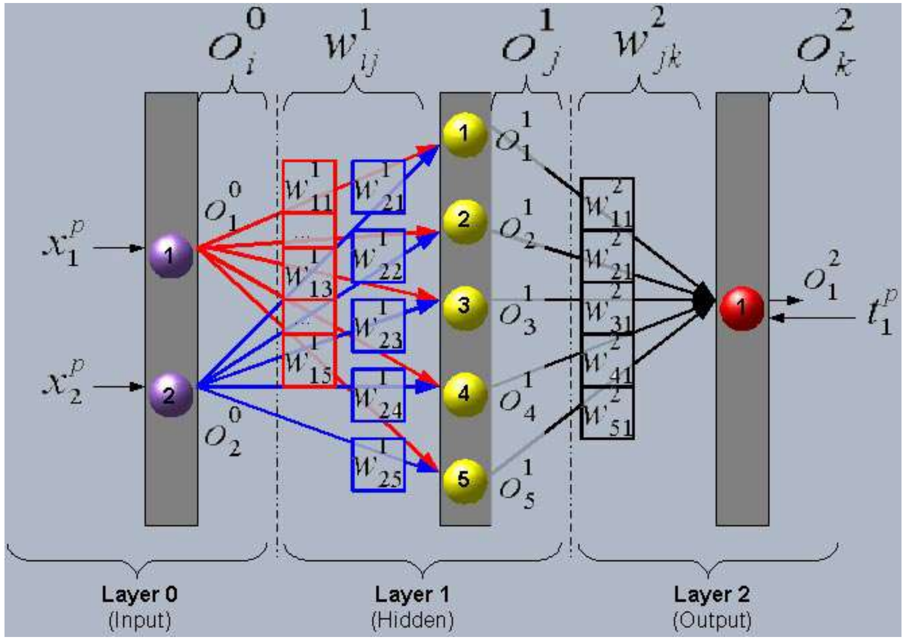
  3. For each hidden unit h

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

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

$$\frac{\partial E}{\partial w_{i,j}^{(1)}} = \underline{\delta_j x_i}, \qquad \frac{\partial E}{\partial w_{j,k}^{(2)}} = \underline{\delta_k o_j},$$

$$w_{i,j} := w_{i,j} + \Delta w_{i,j}$$



multiply δ (the unit at the output end of the weight) by the value for the unit at the input end of the weight

More on Backpropatation

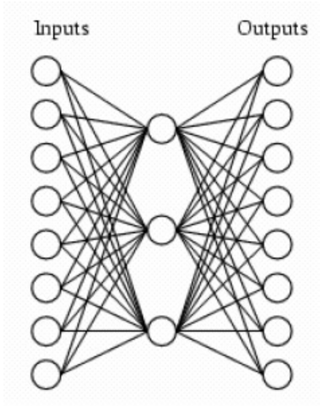
- It is doing gradient descent over entire network weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(t) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(t-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to subsequent testing examples?
- Training can take thousands of iterations, → very slow!
- Using network after training is very fast

Learning Hidden Layer Representation

- A network:



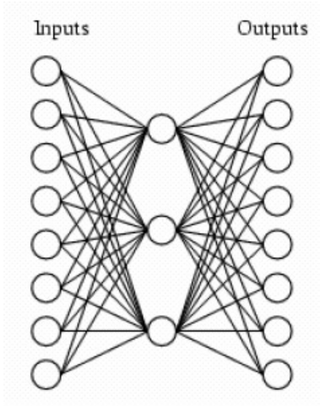
- A target function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

- Can this be learned?

Learning Hidden Layer Representation

- A network:



- Learned hidden layer representation:

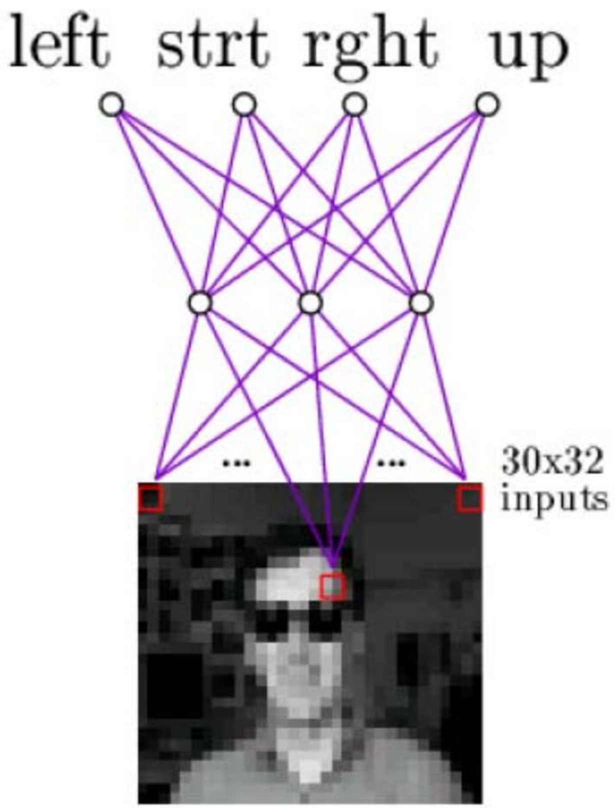
Input		Hidden Values				Output
10000000	$\rightarrow$	.89	.04	.08	$\rightarrow$	10000000
01000000	$\rightarrow$	.01	.11	.88	$\rightarrow$	01000000
00100000	$\rightarrow$	.01	.97	.27	$\rightarrow$	00100000
00010000	$\rightarrow$	.99	.97	.71	$\rightarrow$	00010000
00001000	$\rightarrow$	.03	.05	.02	$\rightarrow$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

Expressive Capabilities of ANNs

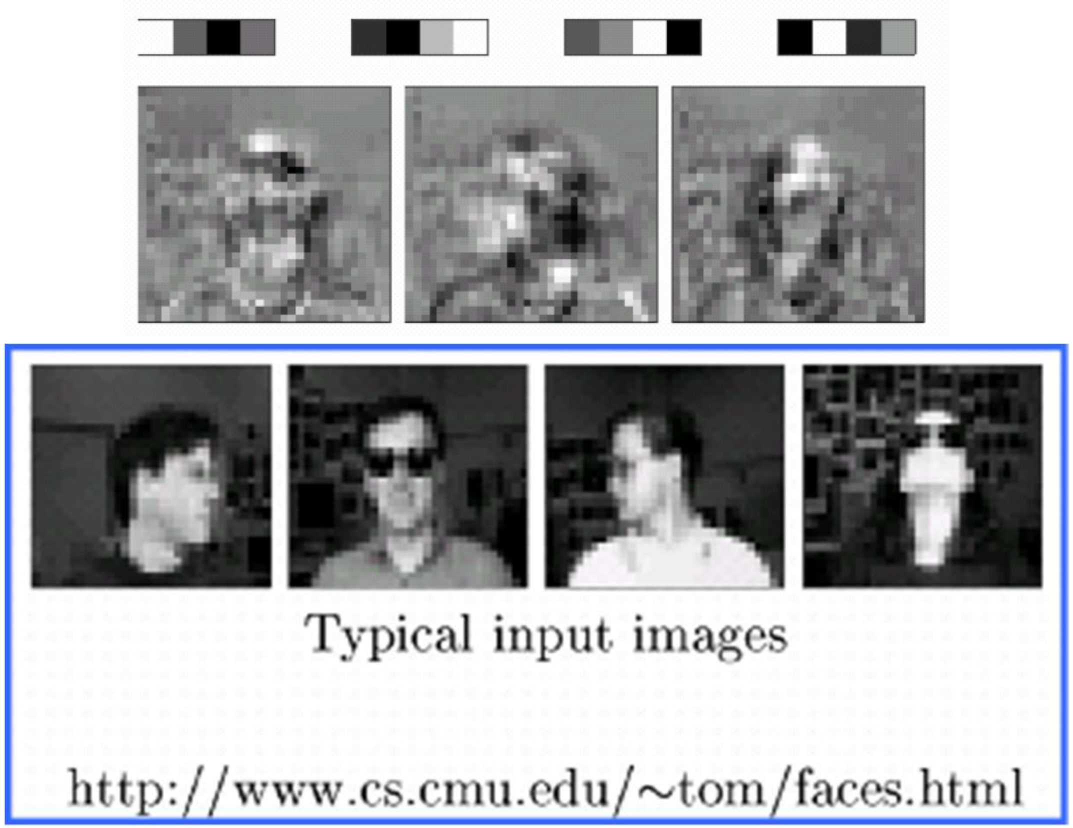
- Boolean functions:
  - Every Boolean function can be represented by network with single hidden layer
  - But might require exponential (in number of inputs) hidden units
- Continuous functions:
  - Every bounded continuous function can be approximated with arbitrary small
    error, by network with one hidden layer [Cybenko 1989; Hornik et al 1989]
  - Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Application: ANN for Face Reco.

- The model



- The learned hidden unit weights



Artificial neural networks – what you should know

- Highly expressive non-linear functions
- Highly parallel network of logistic function units
- Minimizing sum of squared training errors
  - Gives MLE estimates of network weights if we assume zero mean Gaussian noise on output values
- Minimizing sum of sq errors plus weight squared(regularization)
  - MAP estimates assuming weight priors are zero mean Gaussian
- Gradient descent as training procedure
  - How to derive your own gradient descent procedure
- Discover useful representations at hidden units
- Local minima is greatest problem
- Overfitting, regularization, early stopping