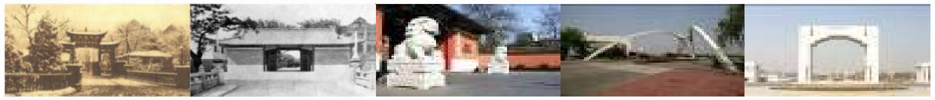
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY



Machine Learning
Lecture 6

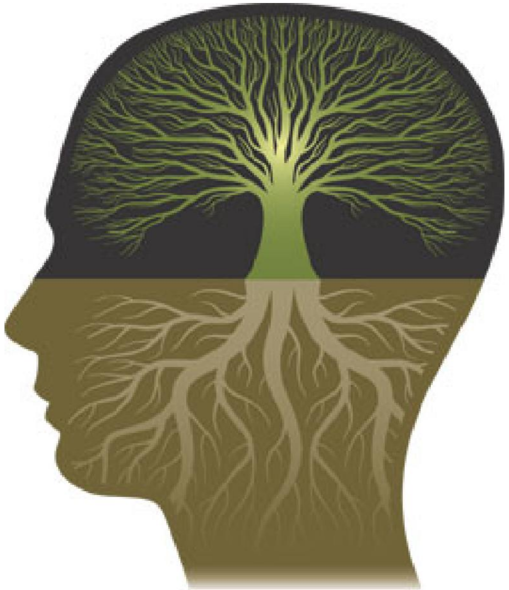
Yang Yang
Department of Computer Science & Engineering
Shanghai Jiao Tong University

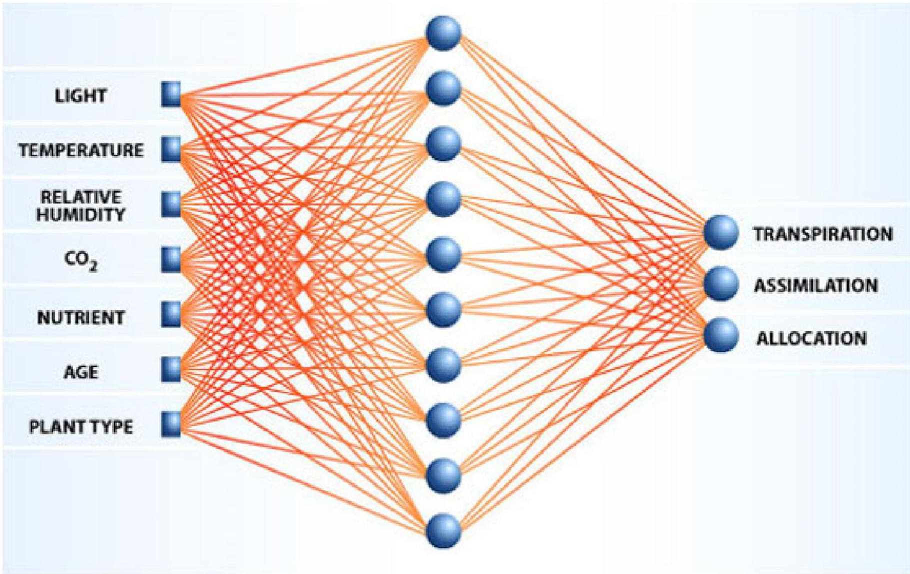
Neural Networks

Perceptron

- 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).

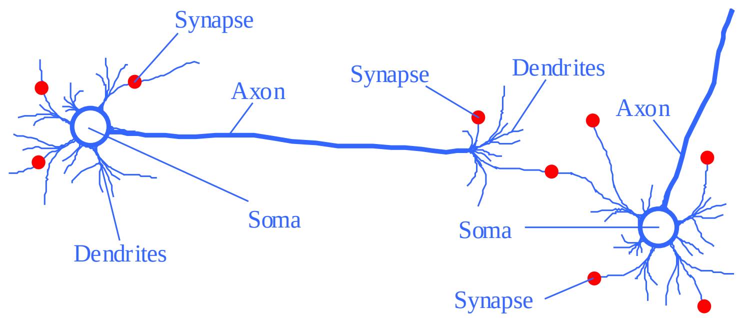
Inspired from Neural Networks

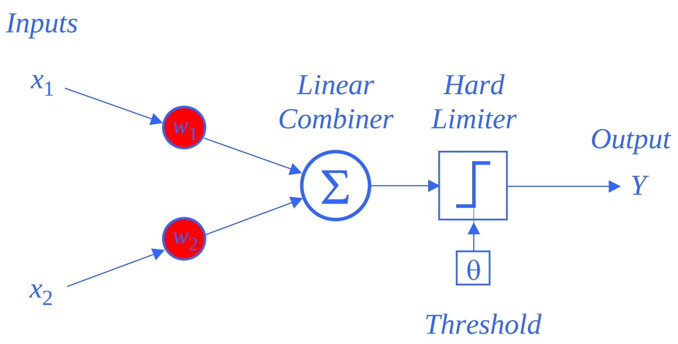




Perceptron and Neural Nets

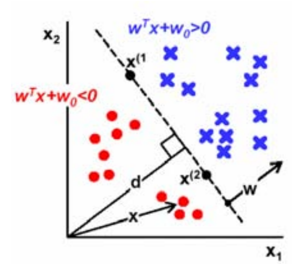
- From biological neuron to artificial neuron (perceptron)



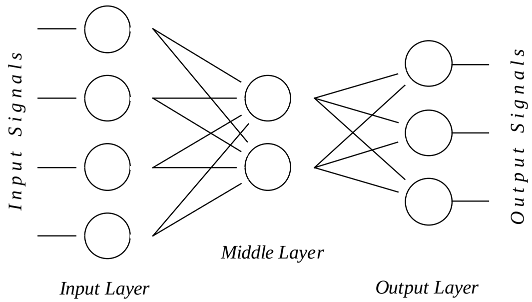


- Activation function

$$X = \sum_{i=1}^{n} x_i w_i \qquad Y = \begin{cases} +1, & \text{if } X \geq \omega_0 \\ -1, & \text{if } X < \omega_0 \end{cases}$$

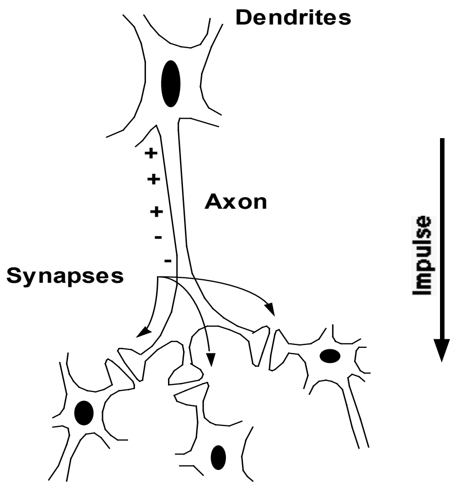


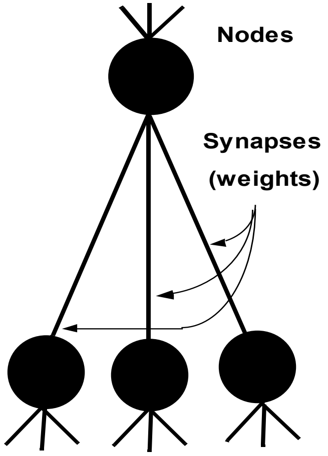
- Artificial neuron networks
  - supervised learning
  - gradient descent



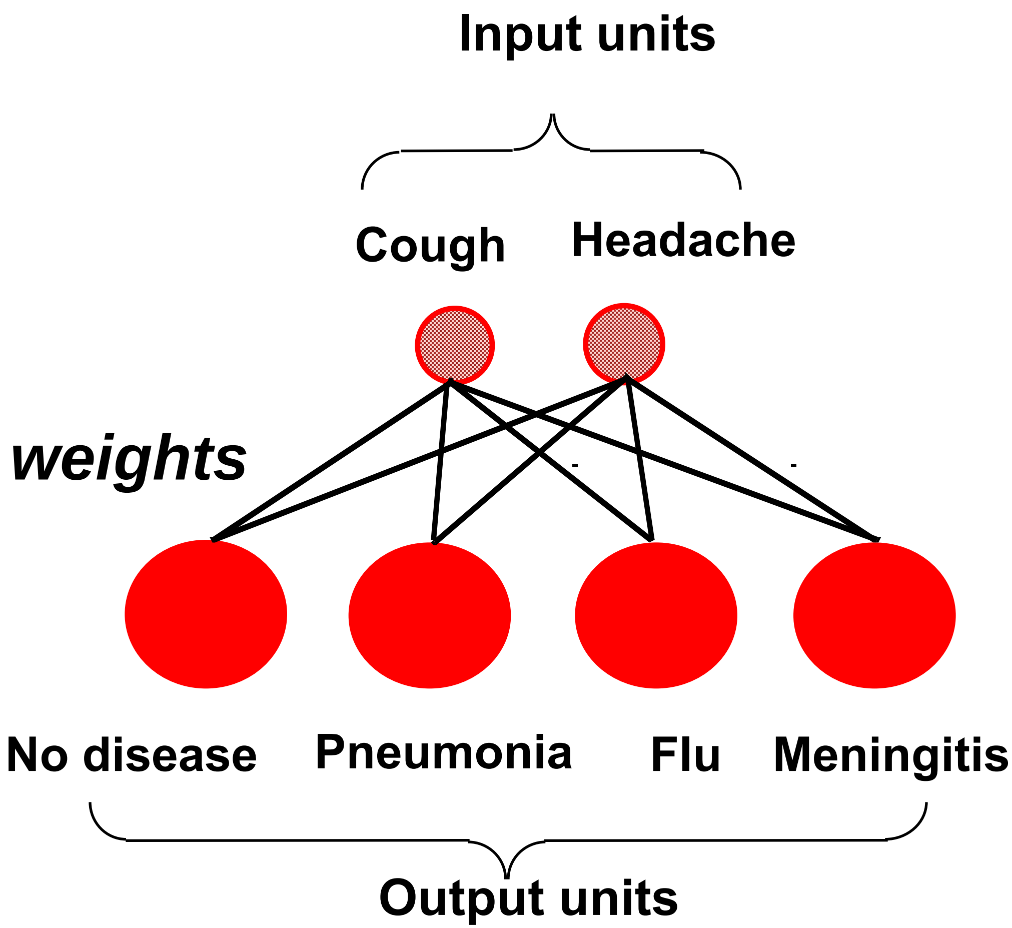
Connectionist Models

- 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





Perceptrons



Δ **rule**

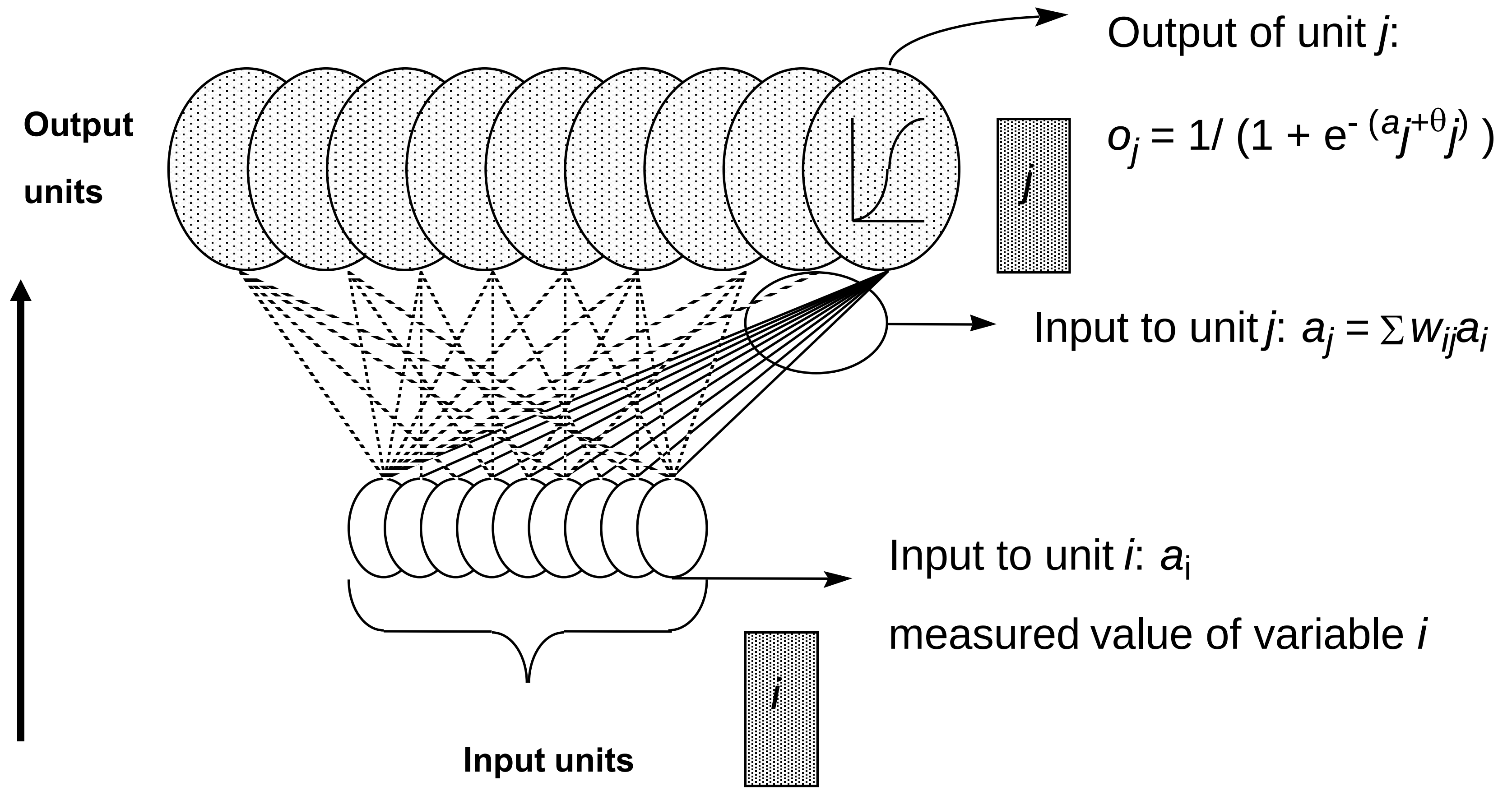
change weights to decrease the error

\- what we got

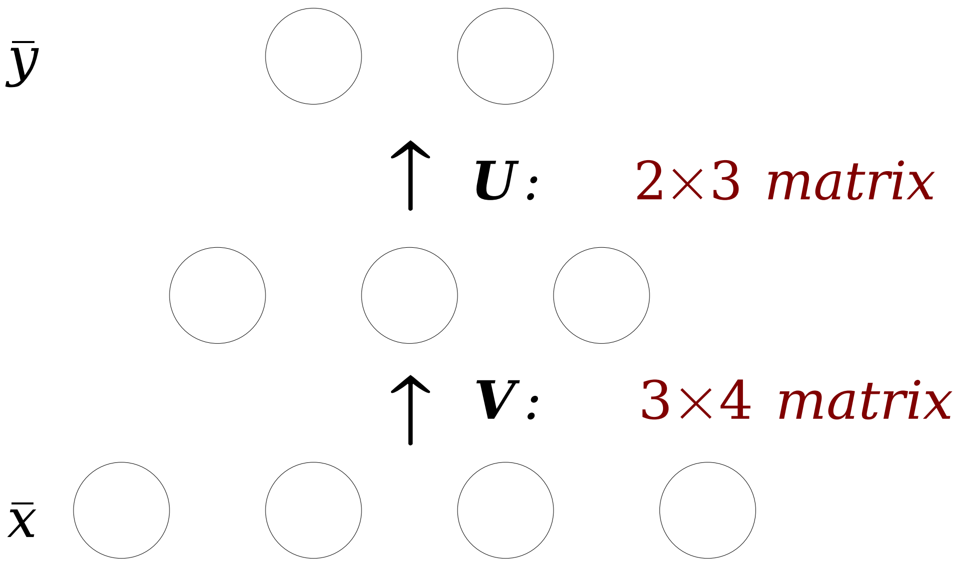
what we wanted

error

Perceptrons



With Linear Units, Multiple Layers Don't Add Anything



$$\bar{y} = \boldsymbol{U} \times (\boldsymbol{V}\bar{x}) = \underbrace{(\boldsymbol{U} \times \boldsymbol{V})}_{2 \times 4}\bar{x}$$

Linear operators are closed under composition.
Equivalent to a single layer of weights $\boldsymbol{W} = \boldsymbol{U} \times \boldsymbol{V}$

But with non-linear units, extra layers add computational power.

Switch to Smooth Nonlinear Units

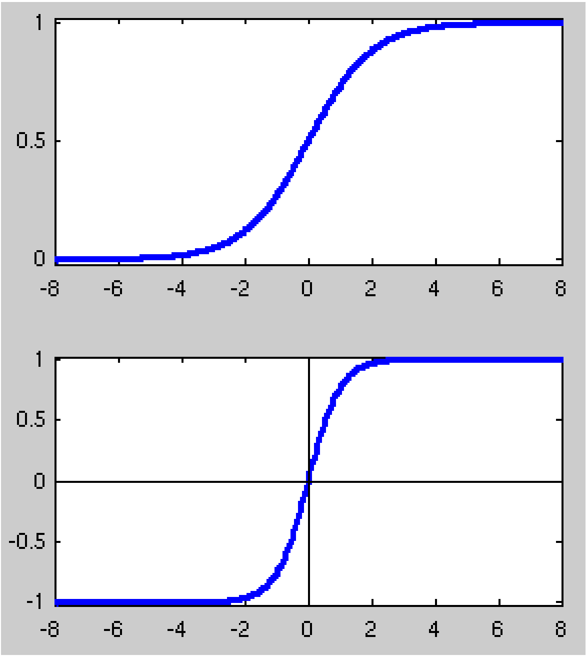
$$\text{net}_j = \sum_i w_{ij} y_i$$

$$y_j = \text{g}(\text{net}_j)$$ *g must be differentiable*

Common choices for g:

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

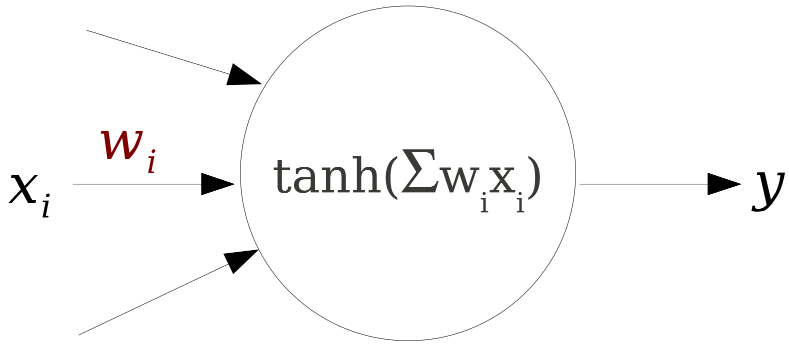
$$g'(x) = g(x)\cdot(1-g(x))$$



$$g(x) = \tanh(x)$$

$$g'(x) = 1/\cosh^2(x)$$

Gradient Descent with Nonlinear Units



$$y = g(net) = \tanh\left(\sum_i w_i x_i\right)$$

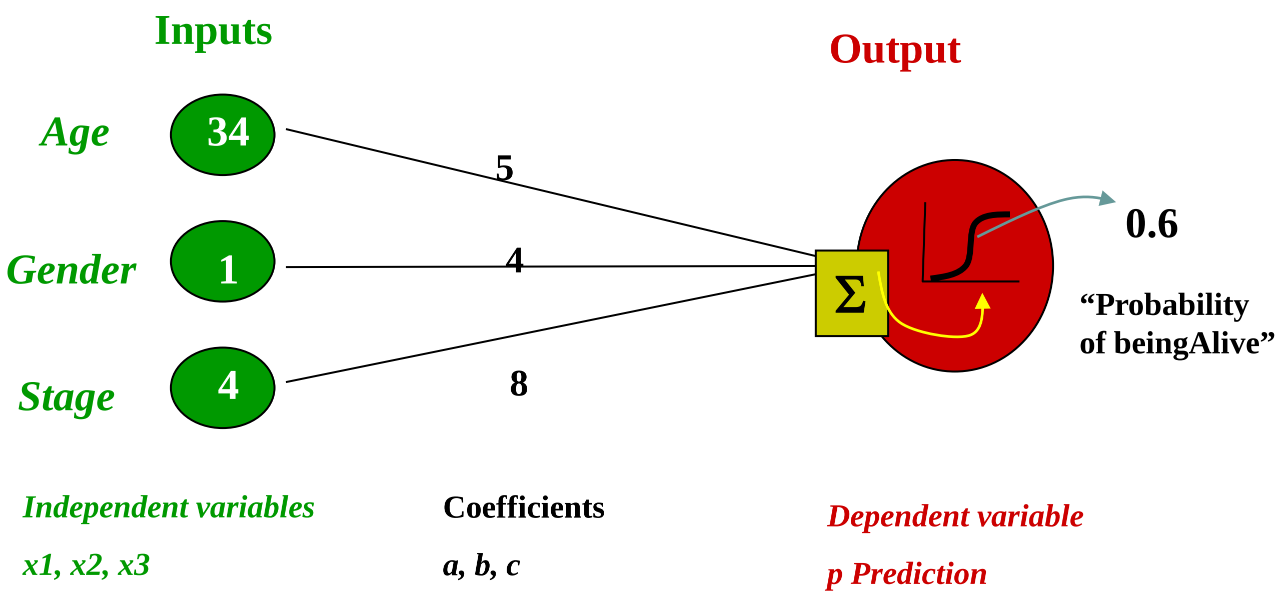
$$\frac{dE}{dy} = (y - d), \qquad \frac{dy}{d\,net} = 1/\cosh^2(net), \qquad \frac{\partial\, net}{\partial w_i} = x_i$$

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{dE}{dy} \cdot \frac{dy}{d\,net} \cdot \frac{\partial\, net}{\partial w_i} \\ &= (y-d)/\cosh^2\left(\sum_i w_i x_i\right) \cdot x_i \end{aligned}$$

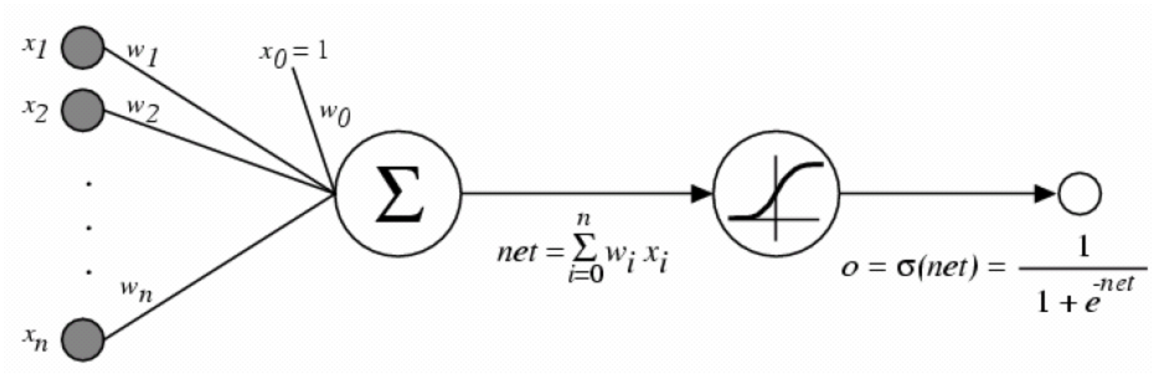
Jargon Pseudo-Correspondence

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

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

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

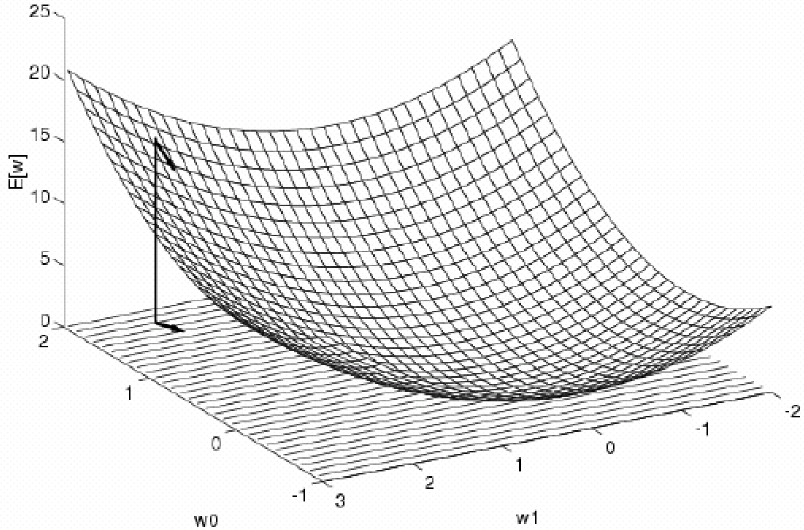
Gradient Descent

x_d = input

t_d = target output

o_d =observed unit output

w_i =weight i



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

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.,

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

The perceptron learning rules

x_d = input

t_d = target output

o_d =observed unit output

w_i =weight i

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

Batch mode:

Do until converge:

1. compute gradient $\nabla E_D[w]$
2. $\vec{w} = \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode:

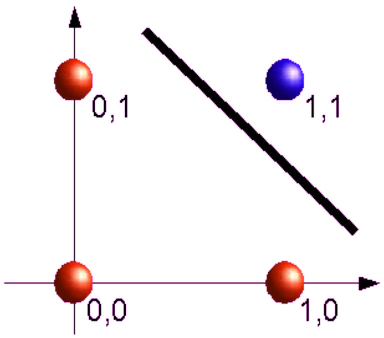
Do until converge:

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

  where

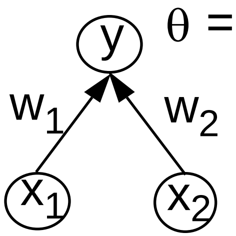
  $$\nabla E_d[\vec{w}] = -(t_d - o_d)o_d(1 - o_d)\vec{x}_d$$

What decision surface does a perceptron define?



NAND

x	y	Z (color)
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**



$\theta = 0.5$

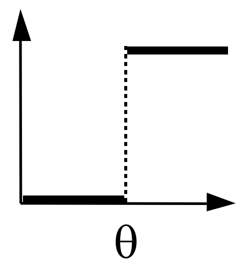
$f(x_1 w_1 + x_2 w_2) = y$

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

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

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

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

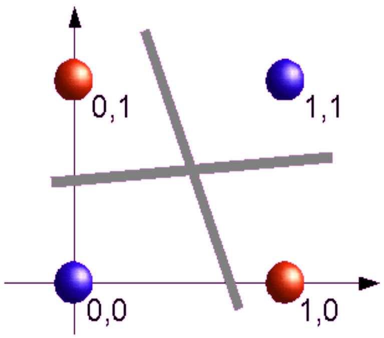


$$f(a) = \begin{cases} 1, \text{ for } a > \theta \\ 0, \text{ for } a \le \theta \end{cases}$$

some possible values for w_1 and w_2

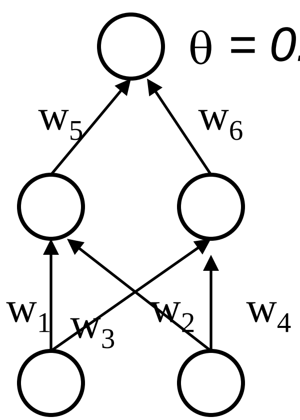
w_1	w_2
0.20	0.35
0.20	0.40
0.25	0.30
0.40	0.20

What decision surface does a perceptron define?

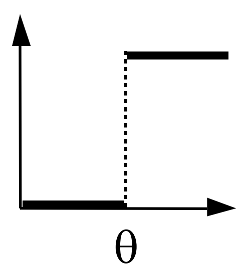


XOR

x	y	Z (color)
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**



θ = *0.5 for all units*



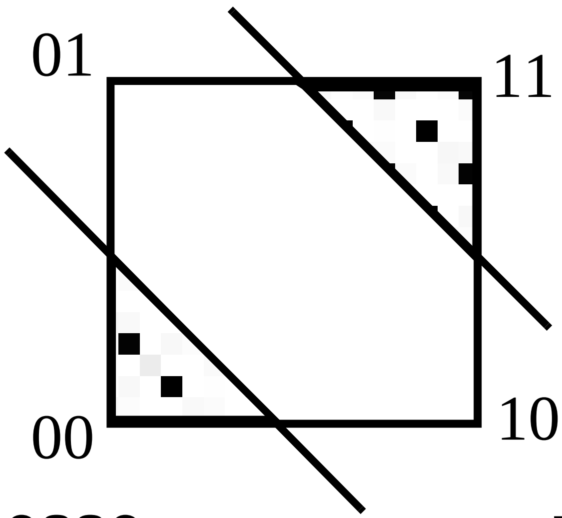
$$f(a) = \begin{cases} 1, \text{ for } a > \theta \\ 0, \text{ for } a \leq \theta \end{cases}$$

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

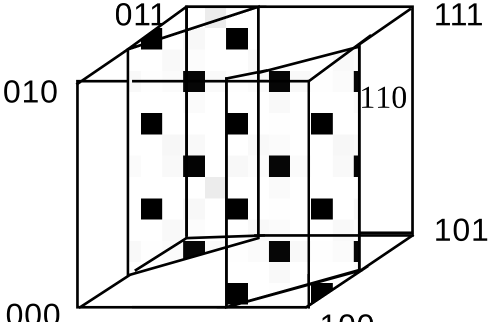


No treatment

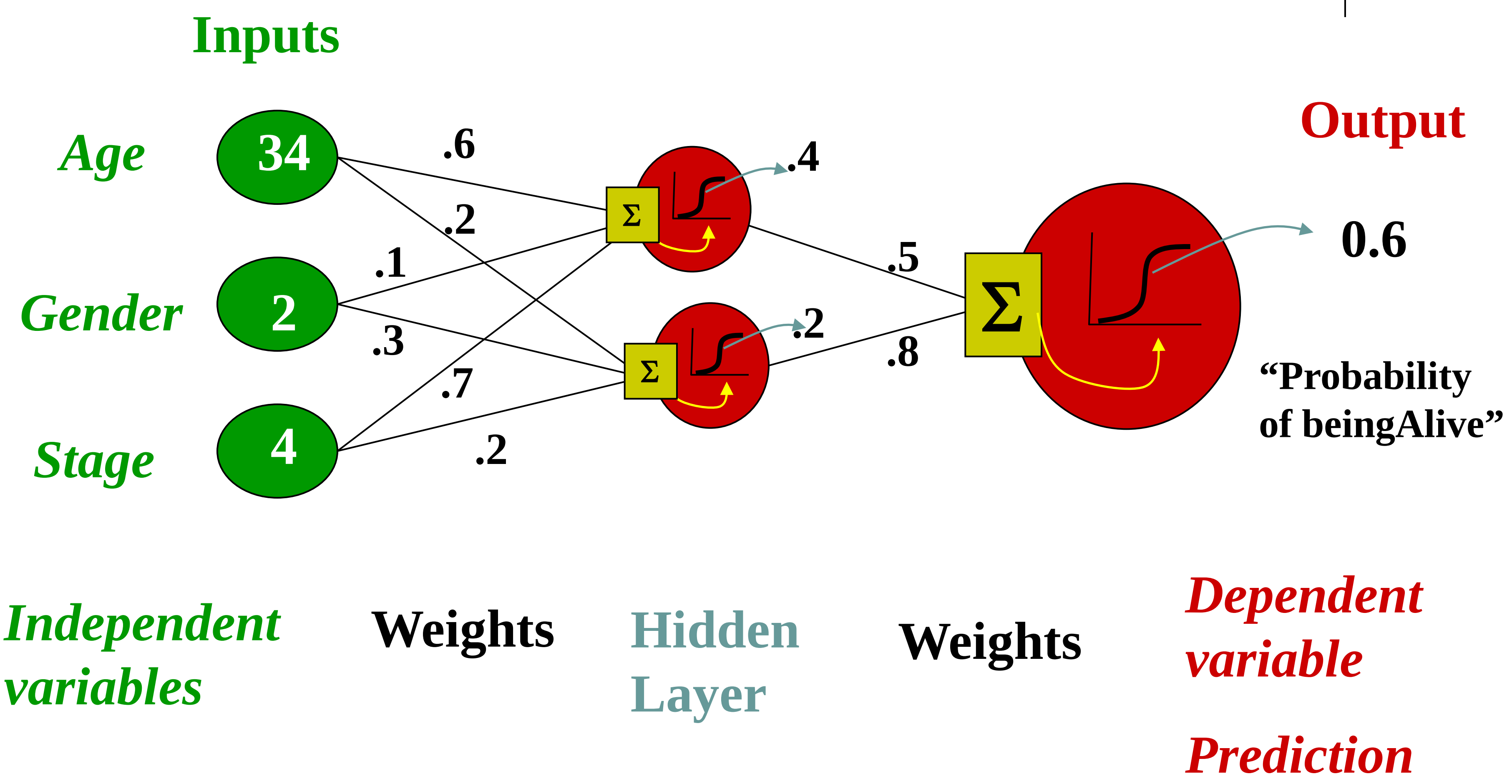
Treatment

No disease
No cough
No headache

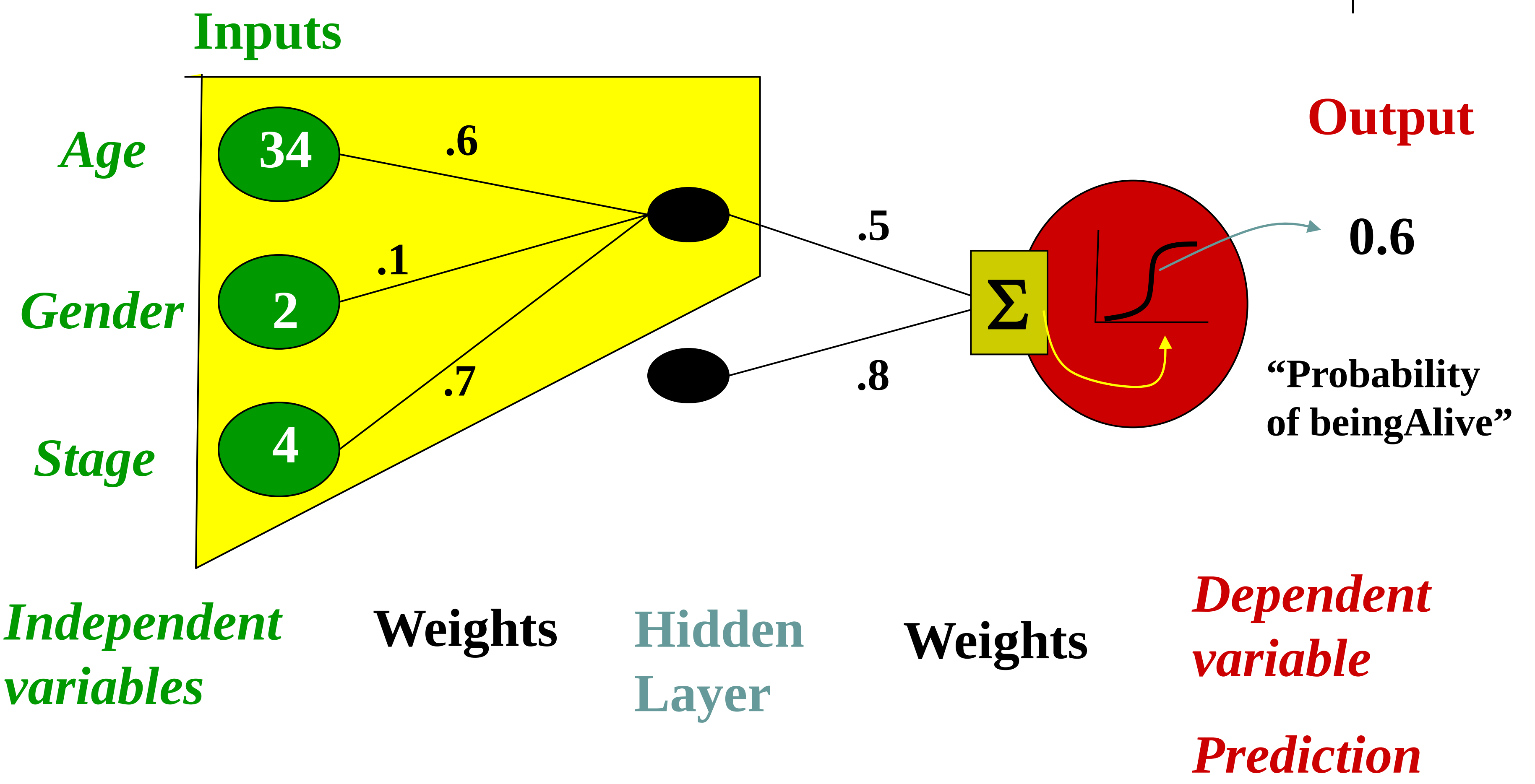
Pneumonia
Cough
No headache

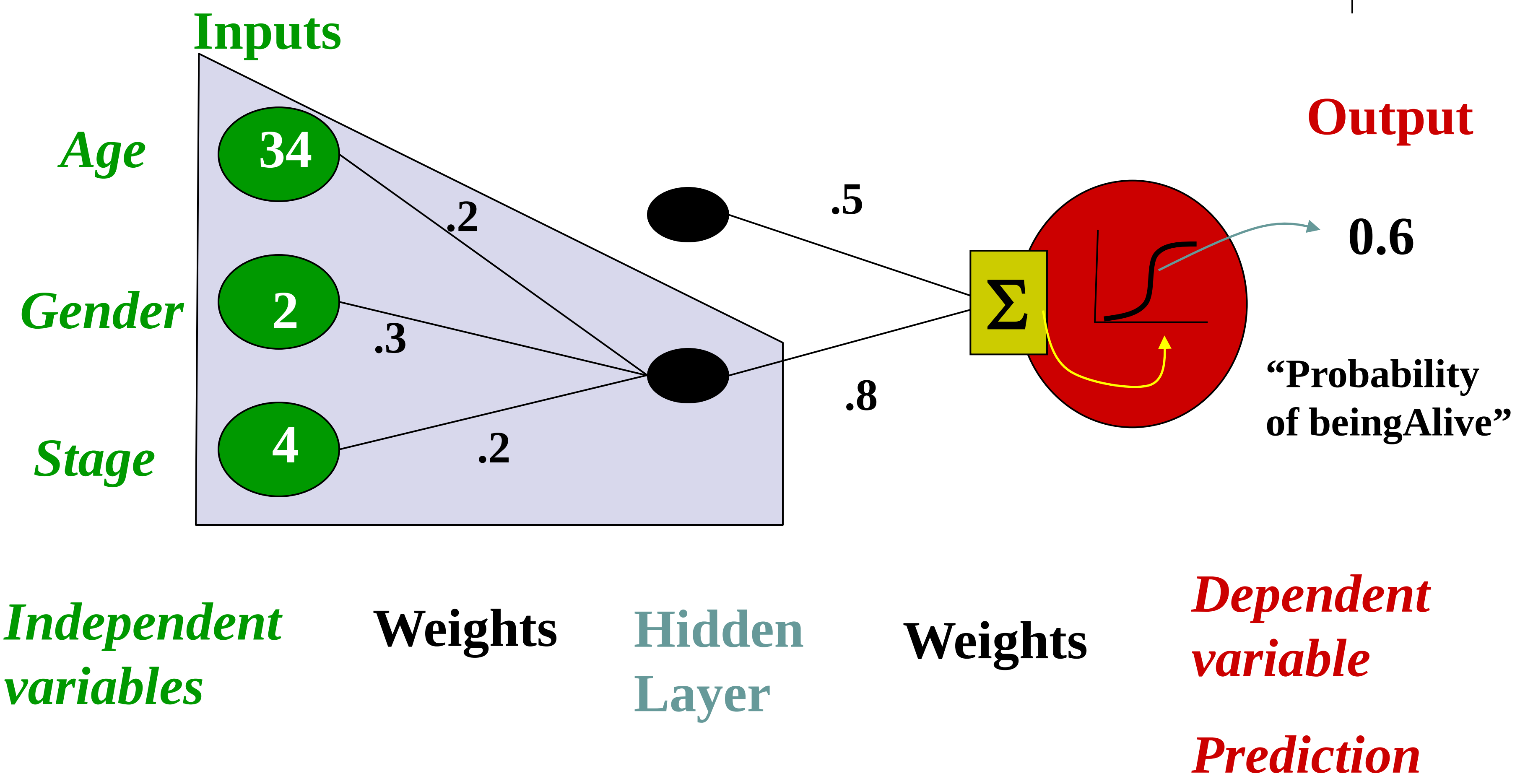


Neural Network Model

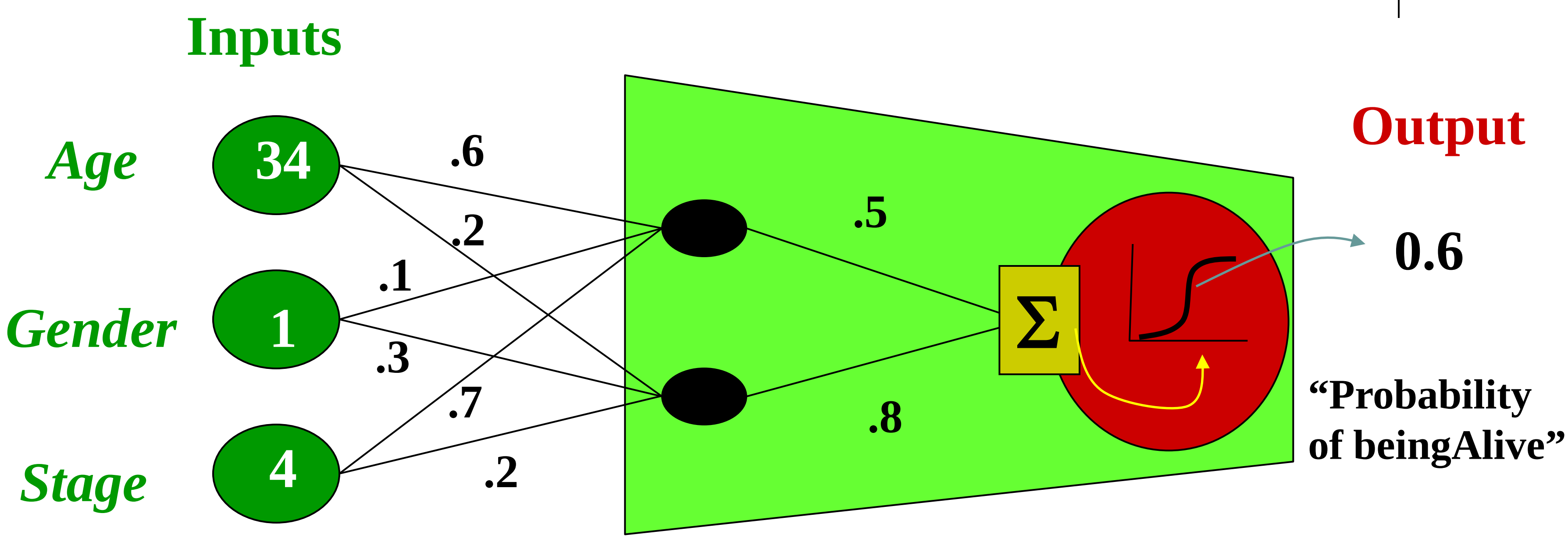


“Combined logistic models”





Inputs
Age
34
Gender
2
Stage
4
.2
.3
.2
.5
.8
Σ
Output
0.6
"Probability of beingAlive"
Independent variables
Weights
Hidden Layer
Weights
Dependent variable
Prediction



Independent variables

Weights

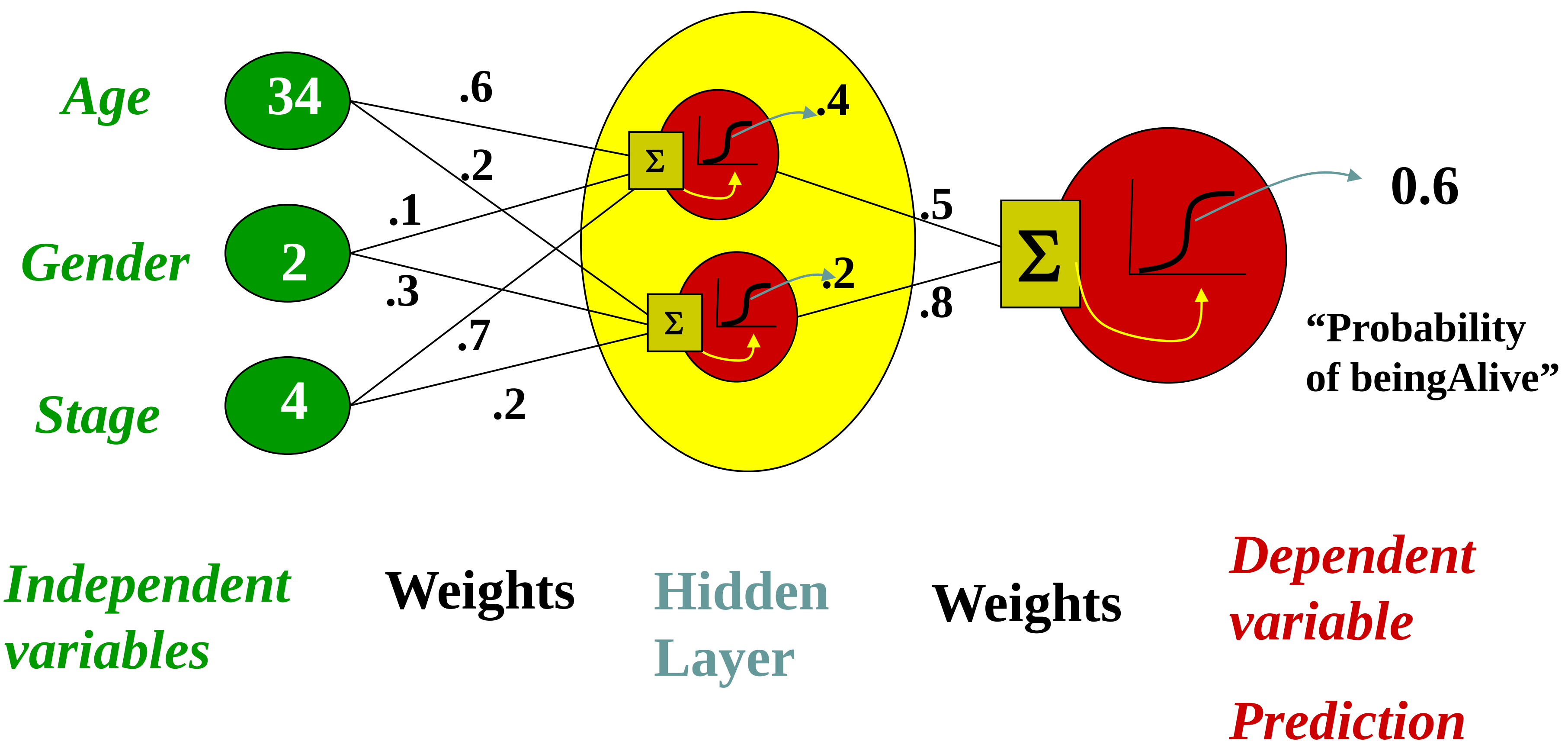
Hidden Layer

Weights

Dependent variable

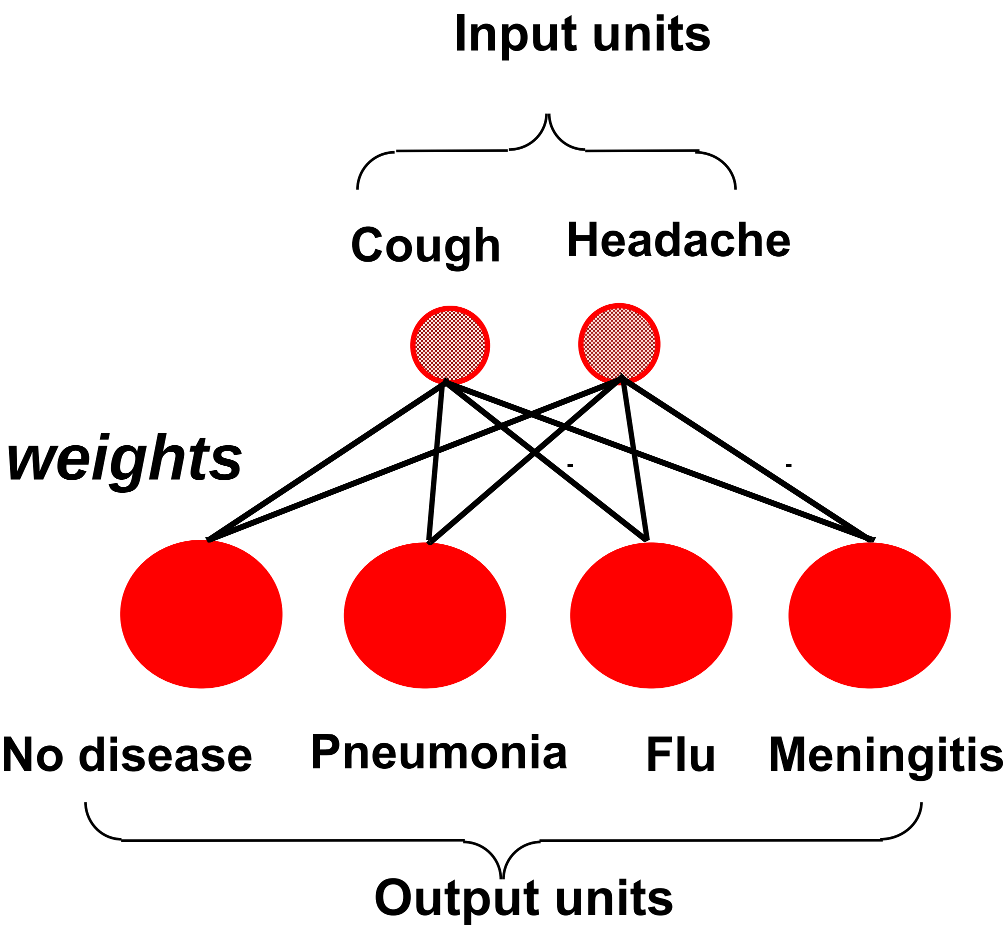
Prediction

Not really, no target for hidden units...



Perceptrons

$$\boxed{\vec{w} \leftarrow \vec{w} + \eta \sum_{d} (t_d - o_d) o_d (1 - o_d) \vec{x}_d}$$

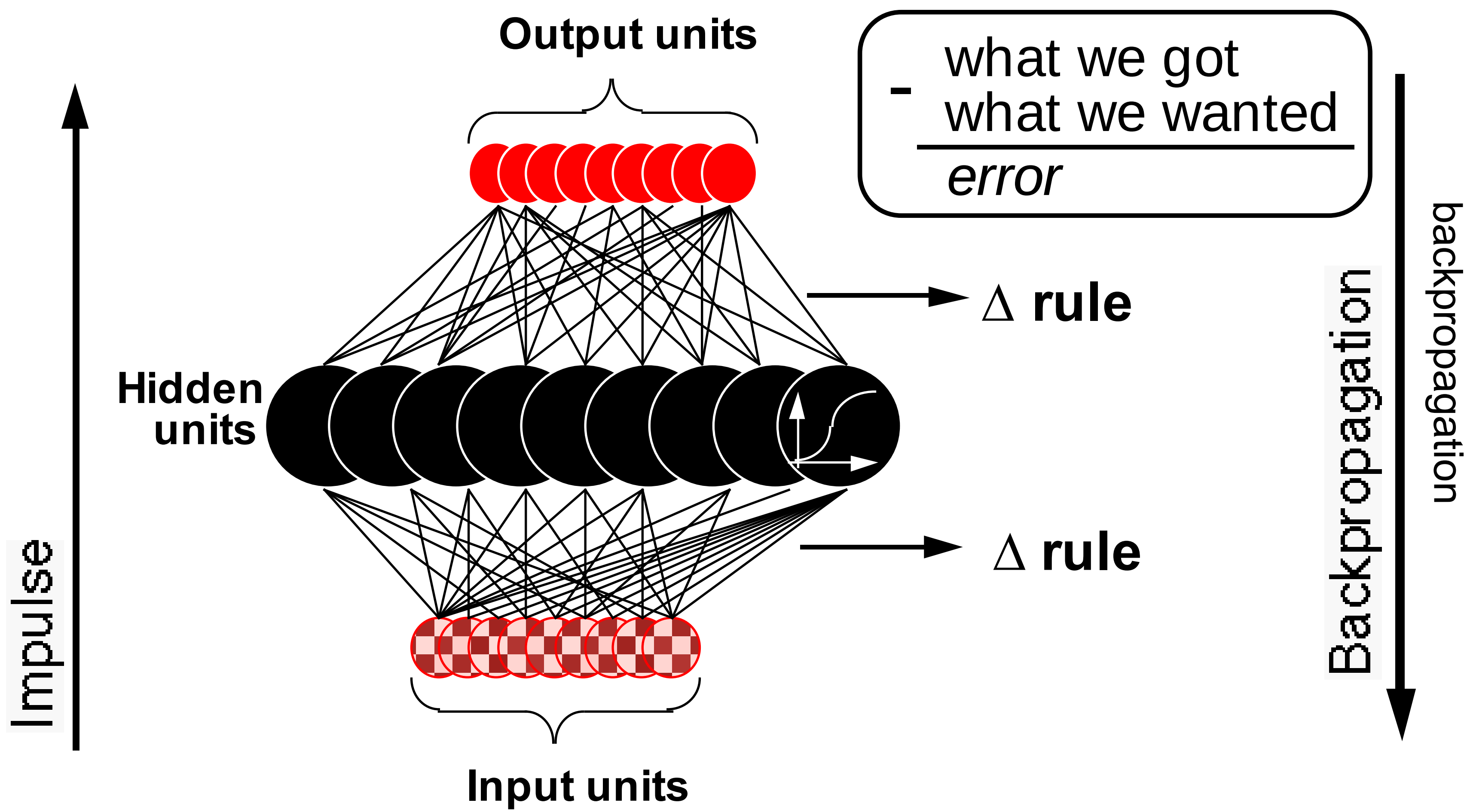


Δ rule

change weights to decrease the error

$$\begin{array}{r l} & \text{what we got} \\ - & \text{what we wanted} \\ \hline & \textit{error} \end{array}$$

Hidden Units and Backpropagation



Backpropagation Algorithm

x_d = input

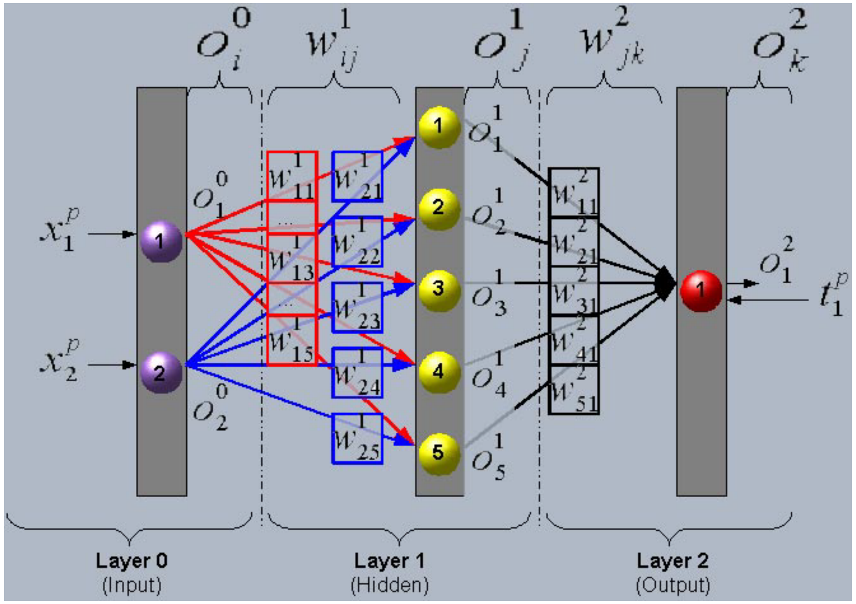
t_d = target output

o_d =observed unit output

w_i =weight i

- Initialize all weights to small random numbers

  Until convergence, Do



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

1. For each output unit k

$$\delta_k \leftarrow o_k^2(1 - o_k^2)(t - o_k^2)$$

2. For each hidden unit h

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

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

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j} \quad \text{where} \quad \Delta w_{i,j} = \eta \delta_j x^j$$

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
- 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

Improving Backprop Performance

- Avoiding local minima
- Keep derivatives from going to zero
- For classifiers, use reachable targets
- Compensate for error attenuation in deep layers
- Compensate for fan-in effects
- Use momentum to speed learning
- Reduce learning rate when weights oscillate
- Use small initial random weights and small initial learning rate to avoid “herd effect”
- Cross-entropy error measure

Avoiding Local Minima

One problem with backprop is that the error surface is no longer bowl-shaped.

Gradient descent can get trapped in local minima.

- In practice, this does not usually prevent learning.

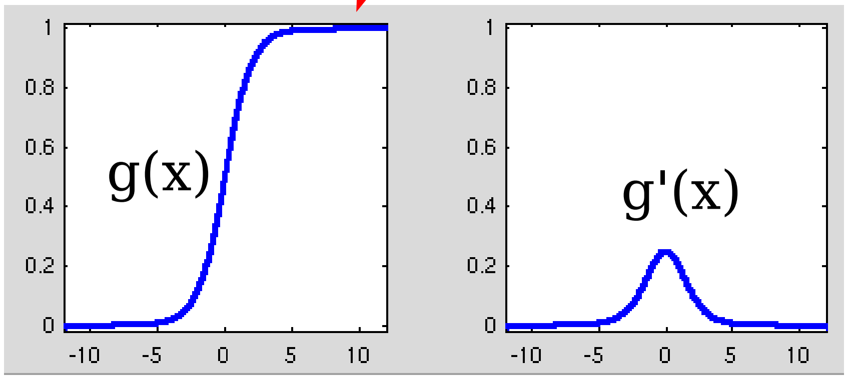
“Noise” can get us out of local minima:

- Stochastic update (one pattern at a time).
- Add noise to training data, weights, or activations.
- Large learning rates can be a source of noise due to overshooting.

Flat Spots

If weights become large, net_j becomes large, derivative of g() goes to zero.

flat spot



Fahlman's trick: add a small constant to g'(x) to keep the derivative from going to zero. Typical value is 0.1.

Reachable Targets for Classifiers

Targets of 0 and 1 are unreachable by the logistic or tanh functions.

Weights get large as the algorithm tries to force each output unit to reach its asymptotic value.

Trying to get a “correct” output from 0.95 up to 1.0 wastes time and resources that should be concentrated elsewhere.

Solution: use “reachable targets” of 0.1 and 0.9 instead of 0/1. And don't penalize the network for overshooting these targets.

Error Signal Attenuation

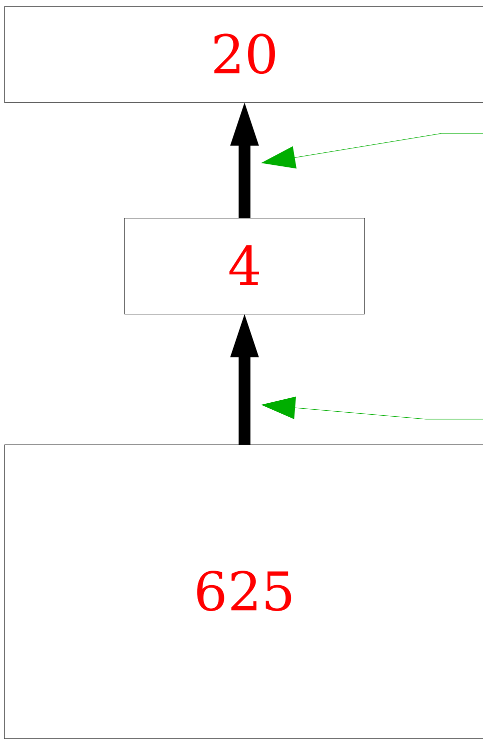
The error signal δ gets attenuated as it moves backward through multiple layers.

So different layers learn at different rates.

Input-to-hidden weights learn more slowly than hidden-to-output weights.

Solution: have different learning rates η for different layers.

Fan-In Affects Learning Rate



One learning step for y_k changes 4 parameters.

One learning step for y_j changes 625 parameters: big change in net_j results!

Solution: scale learning rate by fan-in.

Momentum

Learning is slow if the learning rate is set too low.

Gradient may be steep in some directions but shallow in others.

Solution: add a momentum term α.

$$\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}(t)} + \alpha \cdot \Delta w_{ij}(t-1)$$

Typical value for α is 0.5.

If the direction of the gradient remains constant, the algorithm will take increasingly large steps.

Weights Can Oscillate If Learning Rate Set Too High

Solution: calculate the cosine of the angle between successive weight vectors.

$$\cos\theta = \frac{\vec{\Delta} w(t) \cdot \vec{\Delta} w(t-1)}{\|\vec{\Delta} w(t)\| \cdot \|\vec{\Delta} w(t-1)\|}$$

If cosine close to 1, things are going well.

If cosine < 0.95, reduce the learning rate.

If cosine < 0, we're oscillating: cancel the momentum.

$$\Delta w(t) = -\eta \frac{\partial E}{\partial w} + \alpha \cdot \Delta w(t-1)$$

The “Herd Effect” (Fahlman)

Hidden units all move in the same direction at once, instead of spreading out to divide and conquer.

Solution: use initial random weights, not too large (to avoid flat spots), to encourage units to diversify.

Use a small initial learning rate to give units time to sort out their “specialization” before taking large steps in weight space.

Add hidden units one at a time. (Cascor algorithm.)

Cross-Entropy Error Measure

- Alternative to sum-squared error for binary outputs; diverges when the network gets an output completely wrong.

$$E = \sum_p \left[d^p \log \frac{d^p}{y^p} + (1-d^p) \log \frac{1-d^p}{1-y^p} \right]$$

- Can produce faster learning for some types of problems.

- Can learn some problems where sum-squared error gets stuck in a local minimum, because it heavily penalizes “very wrong” outputs.

How Many Layers Do We Need?

Two layers of weights suffice to compute any “reasonable” function.

But it may require a lot of hidden units!

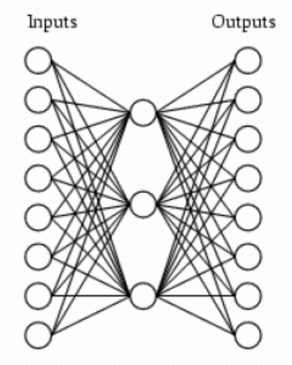
Why does it work out this way?

Lapedes & Farmer: any reasonable function can be approximated by a linear combination of localized “bumps” that are each nonzero over a small region.

These bumps can be constructed by a network with two layers of weights.

Learning Hidden Layer Representation

- A network:



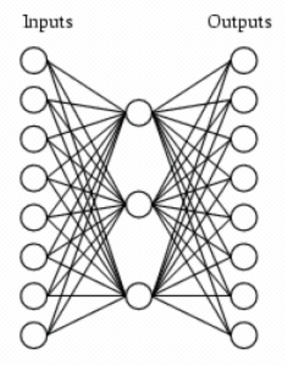
- A target function:

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	→	.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

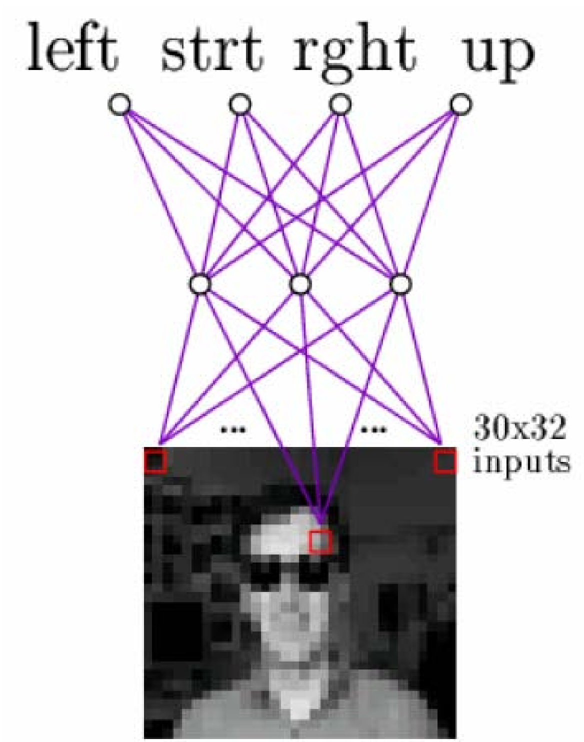
Expressive Capabilities of ANNs

- Boolean functions:
  - Every Boolean function can be represented by network with single hidden layer
  - But might require exponential (in number of inputs) hidden units

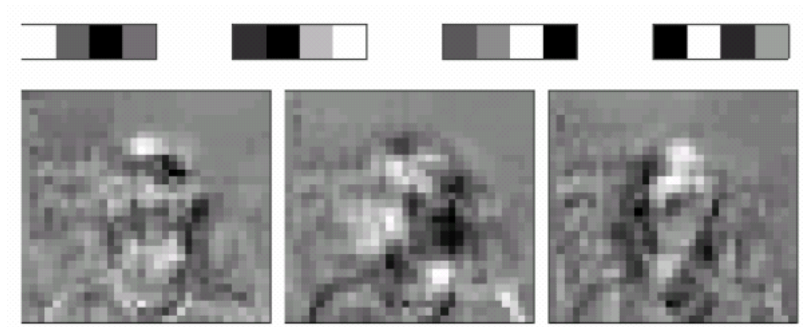
- Continuous functions:
  - Every bounded continuous function can be approximated with arbitrary small error, by network with one hidden layer [Cybenko 1989; Hornik et al 1989]
  - Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Application: ANN for Face Reco.

- The model



- The learned hidden unit weights



Typical input images

http://www.cs.cmu.edu/~tom/faces.html

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