

COMP 472: Artificial Intelligence

Machine Learning

Neural Networks

part 2
video #8

- Russell & Norvig: Sections 19.6, 21.1

Today

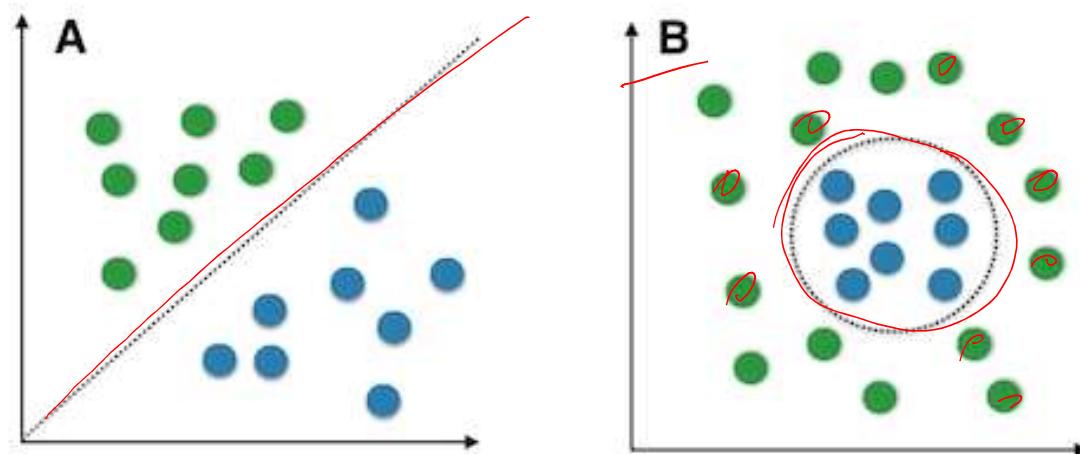
1. Introduction to ML
2. Naïve Bayes Classification
 - a. Application to Spam Filtering
3. Decision Trees
4. (Evaluation
5. Unsupervised Learning)
6. Neural Networks
 - a. Perceptrons
 - b. Multi Layered Neural Networks



Limits of the Perceptron

- can only model linear decision boundaries
- but real-world problems cannot always be represented by linearly-separable functions...

Linear vs. nonlinear problems



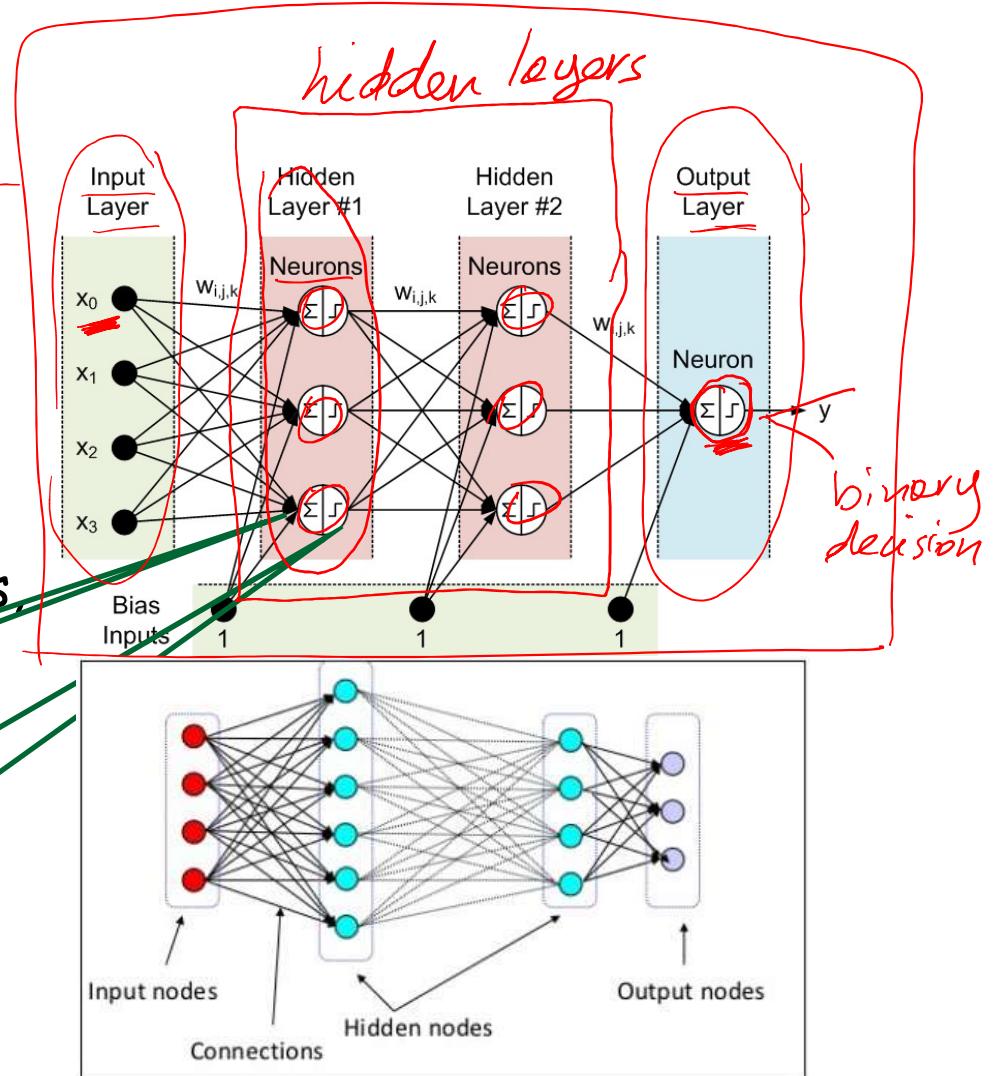
Multilayer Neural Networks

- Solution:

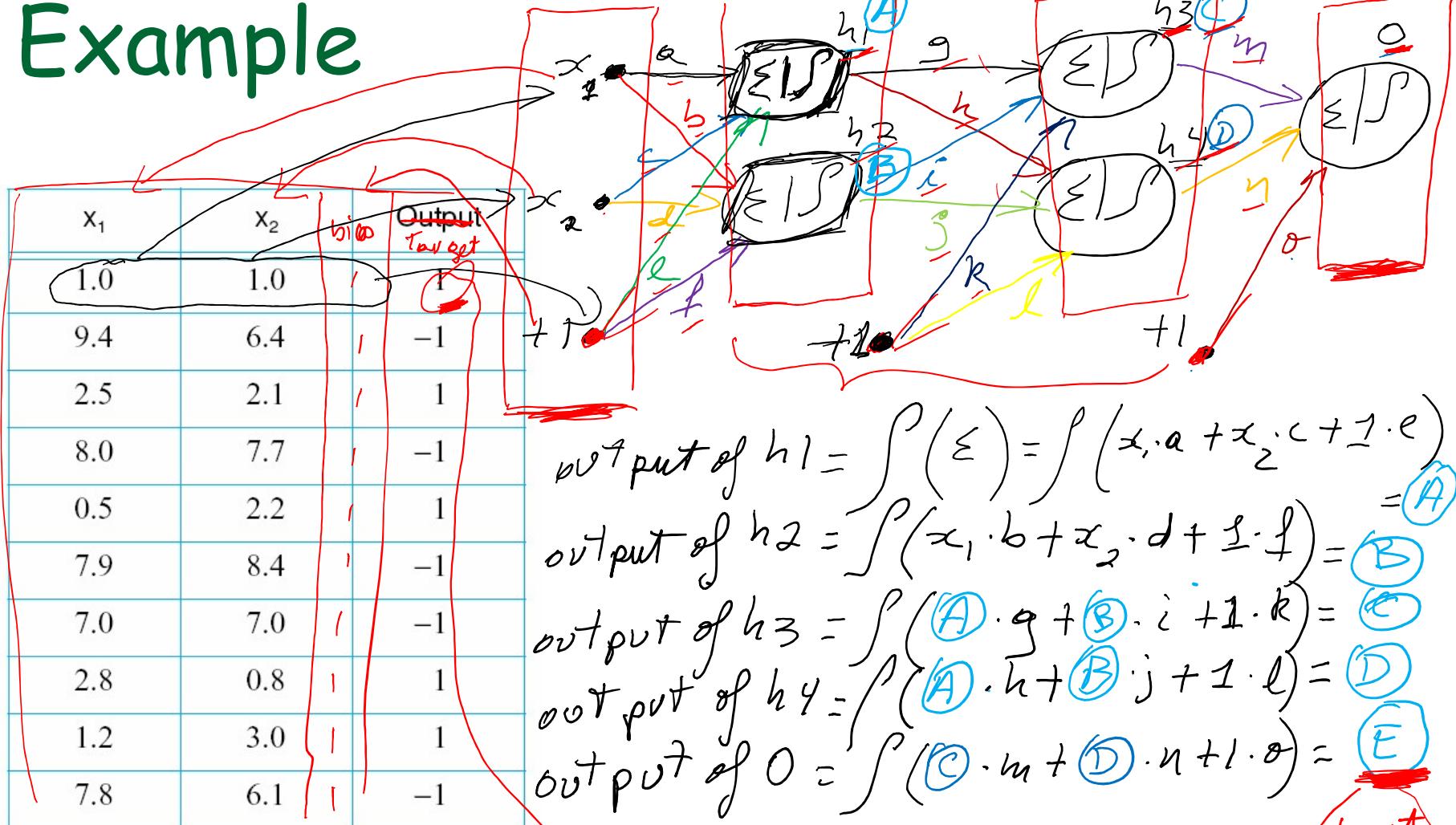
- use a non-linear activation function
- to learn more complex functions (more complex decision boundaries), have hidden nodes
- and for non-binary decisions, have multiple output nodes

usual transfer function

Non-linear, differentiable activation function



Example



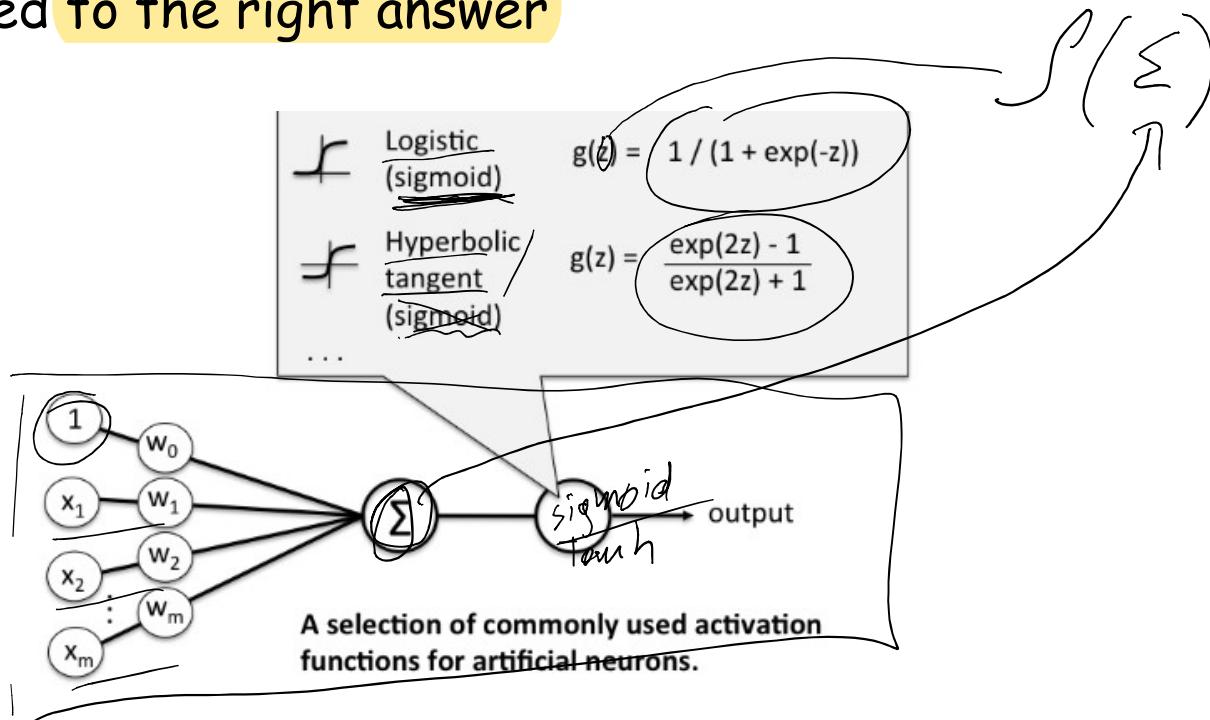
feed forward

output
of the
network

backprop

Activation Functions

- Backpropagation requires a differentiable activation function
- that returns continuous values within a range
 - e.g. a value between 0 and 1 (instead of 0 or 1 , like the perceptron)
- indicates how close/how far the output of the network is compared to the right answer



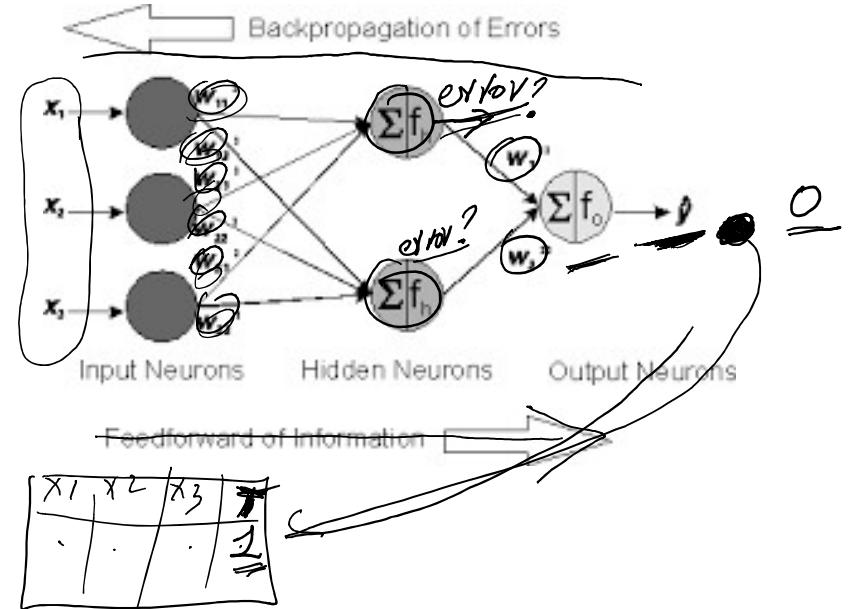
<https://medium.com/towards-data-science/activation-functions-and-its-types-which-is-better-a9a5310cc8f>

Learning in a Neural Network

- Learning is the same as in a perceptron:

1. Feed-forward:

- Input from the features is fed forward in the network from input layer towards the output layer



2. Backpropagation:

- Error rate flows backwards from the output layer to the input layer (to adjust the weights in order to minimize the error)

3. Iterate until error rate is minimized

- repeat the forward pass and back pass for the next data points until all data points are examined (1 epoch)
- repeat this entire exercise (several epochs) until the overall error is minimised

Typical Cost Functions

- Error of the network is computed via a cost function

1. Quadratic Cost:

- aka mean squared error (MSR)
- minimize the difference between output values and target values

$$C = \left(\frac{1}{n} \sum_{i=1}^n (T_i - O_i)^2 \right)$$

- used mostly for regression tasks

where n = nb of instances

$$\begin{aligned} i &= 1 & (T_1 - O_1)^2 \\ i &= 2 & (T_2 - O_2)^2 \\ i &= 3 & (T_3 - O_3)^2 \\ &\vdots & \sum \end{aligned}$$

2. Cross-entropy:

- used mostly for classification tasks
- where we don't care about the exact value of the network output, we only care about the final class

$$C = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K (T_{ik} \log(O_{ik}))$$

where:

n = nb of instances

K = nb of classes

\log = base 2, base 10, ln,

Example of Cross Entropy

- Assume 3 classes: red, blue and green

i.e. $K = 3$



- for a specific instance i , the target is green

i.e. $(\text{red}, \text{blue}, \text{green}) = (0, 0, 1)$ // real distribution

$T^1 = 0$ $T^2 = 0$ $T^3 = 1$

- but the model predicts the probabilities as:

$(\text{red}, \text{blue}, \text{green}) = (0.1, 0.4, 0.5)$ // predicted distribution

$O^1 = 0.1$ $O^2 = 0.4$ $O^3 = 0.5$

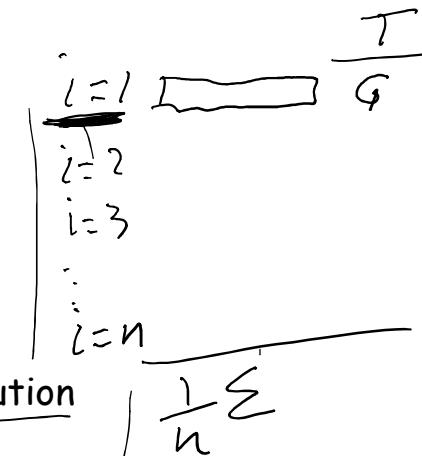
- the cross entropy of two distributions (real and predicted)

$$\sum_{k=1}^K (T^k \ln(O^k)) = -((0) \ln(0.1) + (0) \ln(0.4) + (1) \ln(0.5))$$

- but we have several instances in the test set, so we will take the average of the cross-entropy across all instances i in the test set

$$C = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K (T_i^k \ln(O_i^k))$$

- see an example calculation computation in a few slides



Backpropagation

- In a multilayer network...

- Computing the error in the output layer is clear.
 - Computing the error in the hidden layers is not clear, because we don't know what output it should be

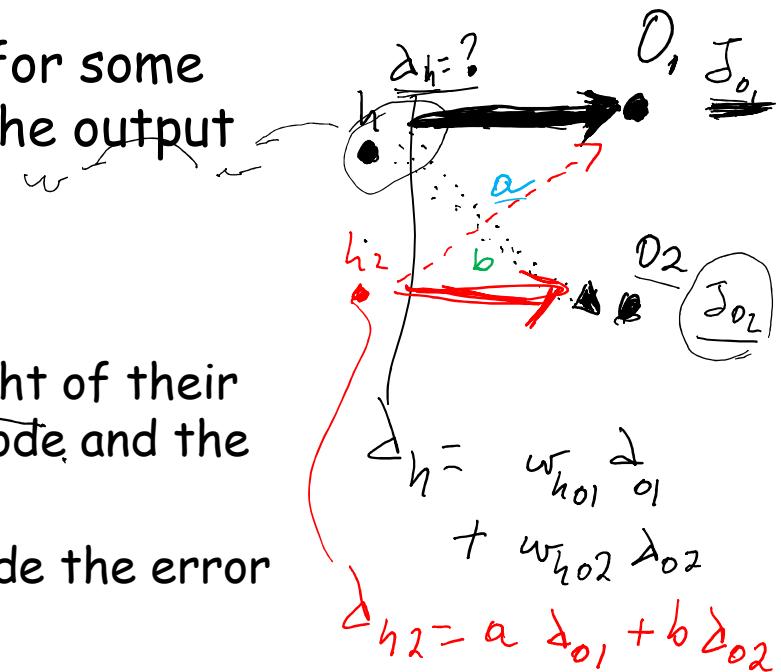
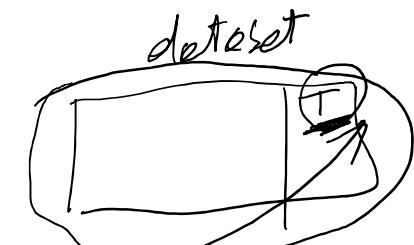
- Intuitively:

- A hidden node h is "responsible" for some fraction of the error in each of the output node to which it connects.

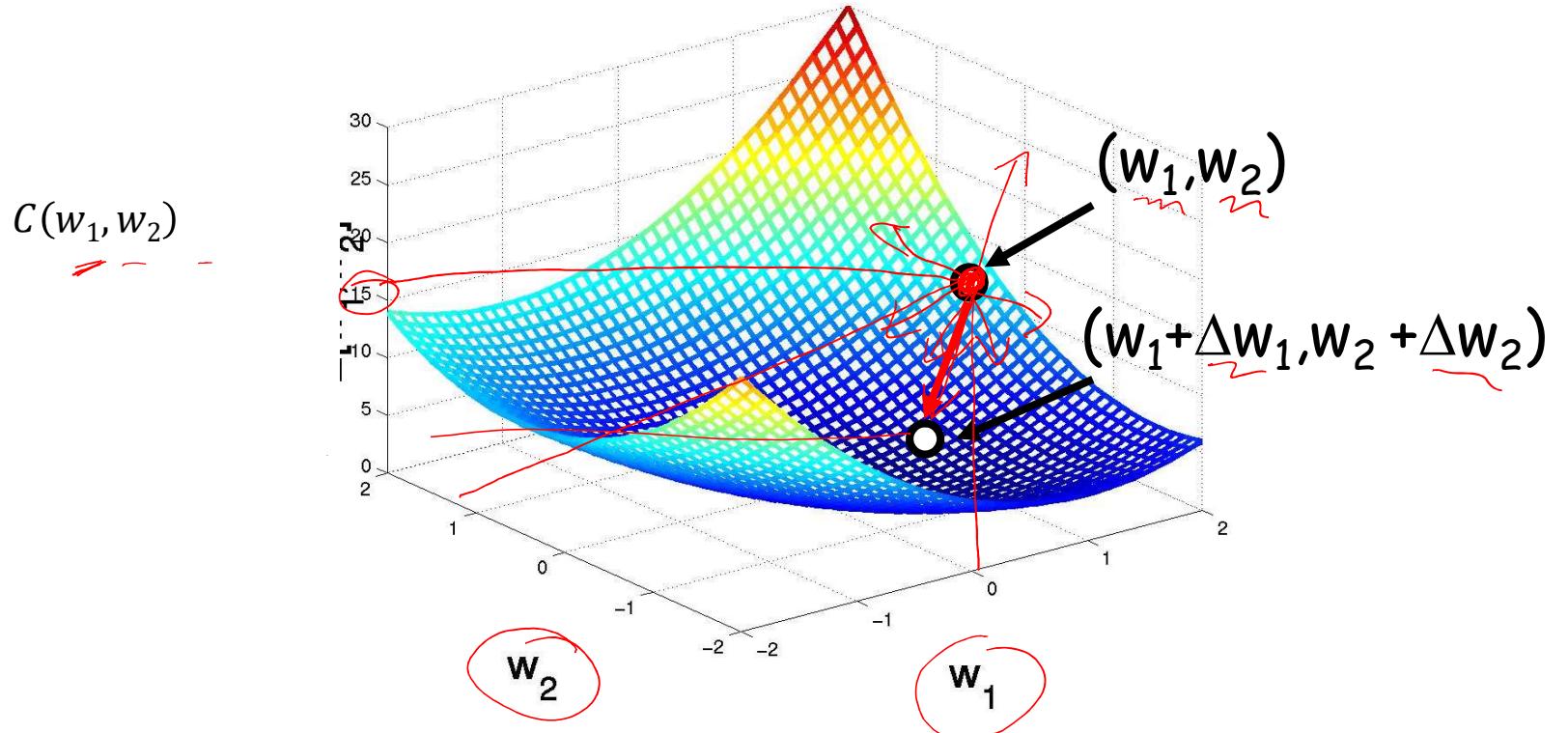
- So the error values (δ):

- are divided according to the weight of their connection between the hidden node and the output node

- and are propagated back to provide the error values (δ) for the hidden layer.



Gradient Descent - Adjusting the Weights

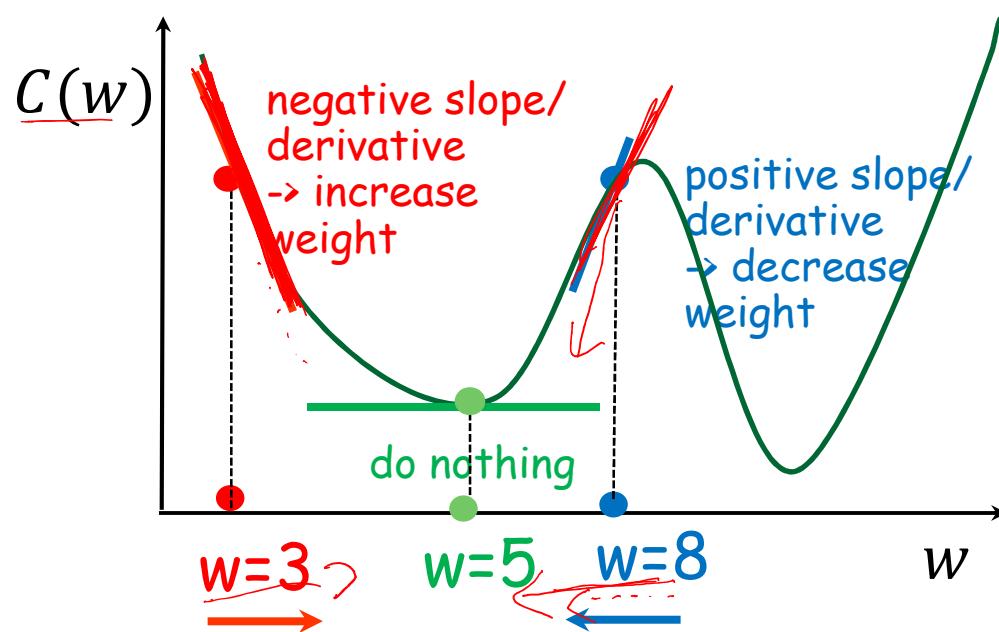


- Goal: minimize $C(w_1, w_2)$ by changing w_1 and w_2
- What is the best combination of change in w_1 and w_2 to minimize C faster?

Gradients

Gradient is just derivative in 1D

Eg: $C(w) = (w - 5)^2$ derivative is:
only 1 weight



$$\frac{\partial C}{\partial w} = 2(w - 5)$$

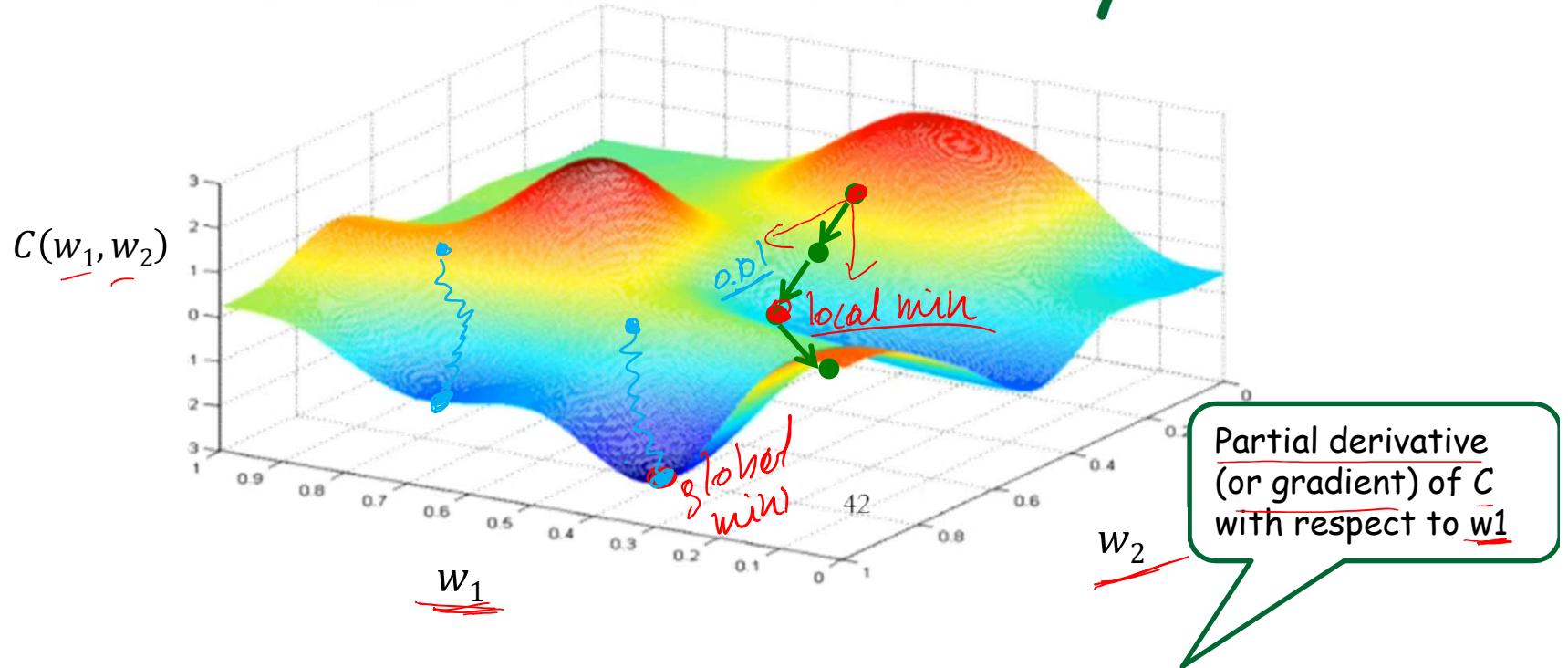
If $w=3$ $\frac{\partial C}{\partial w}(3) = 2(3 - 5) = -4$

derivative says increase w
(go in opposite direction
of derivative)

If $w=8$ $\frac{\partial C}{\partial w}(8) = 2(8 - 5) = 6$

derivative says decrease w
(go in opposite direction
of derivative)

Gradient Descent Visually



- need to know how much a change in w_1 will affect $C(w_1, w_2)$ i.e $\frac{\partial C}{\partial w_1}$
- need to know how much a change in w_2 will affect $C(w_1, w_2)$ i.e $\frac{\partial C}{\partial w_2}$
- Gradient ∇C points in the opposite direction of steepest decrease of $C(w_1, w_2)$
- i.e. hill-climbing approach...

Training the Network

After some calculus (see: <https://en.wikipedia.org/wiki/Backpropagation> we get...)

- ✓ Step 0: Initialise the weights of the network randomly
- // feedforward
- Step 1: Do a forward pass through the network

$$O_i = g\left(\sum_j w_{ji} x_j\right) = \text{sigmoid}\left(\sum_j w_{ji} x_j\right) = \frac{1}{1 + e^{-\left(\sum_j w_{ji} x_j\right)}}$$

// propagate the errors backwards

- Step 2: For each output unit k , calculate its error term δ_k

$$\delta_k \leftarrow g'(x_k) \times Err_k = O_k(1 - O_k) \times (O_k - T_k)$$

- Step 3: For each hidden unit h , calculate its error term δ_h

$$\delta_h \leftarrow g'(x_h) \times Err_h = O_h(1 - O_h) \times \sum_{k \in \text{outputs}} w_{hk} \delta_k$$

- Step 4: Update each network weight w_{ij} :

$$w_{ij} \leftarrow w_{ij} \pm \Delta w_{ij} \text{ where } \Delta w_{ij} = -n_s \delta_j O_i$$

- Repeat steps 1 to 4 until the cost (C) is minimised

Note: To be consistent with Wikipedia, we'll use O-T instead of T-O, but we will subtract the error in the weight update

Derivative of sigmoid

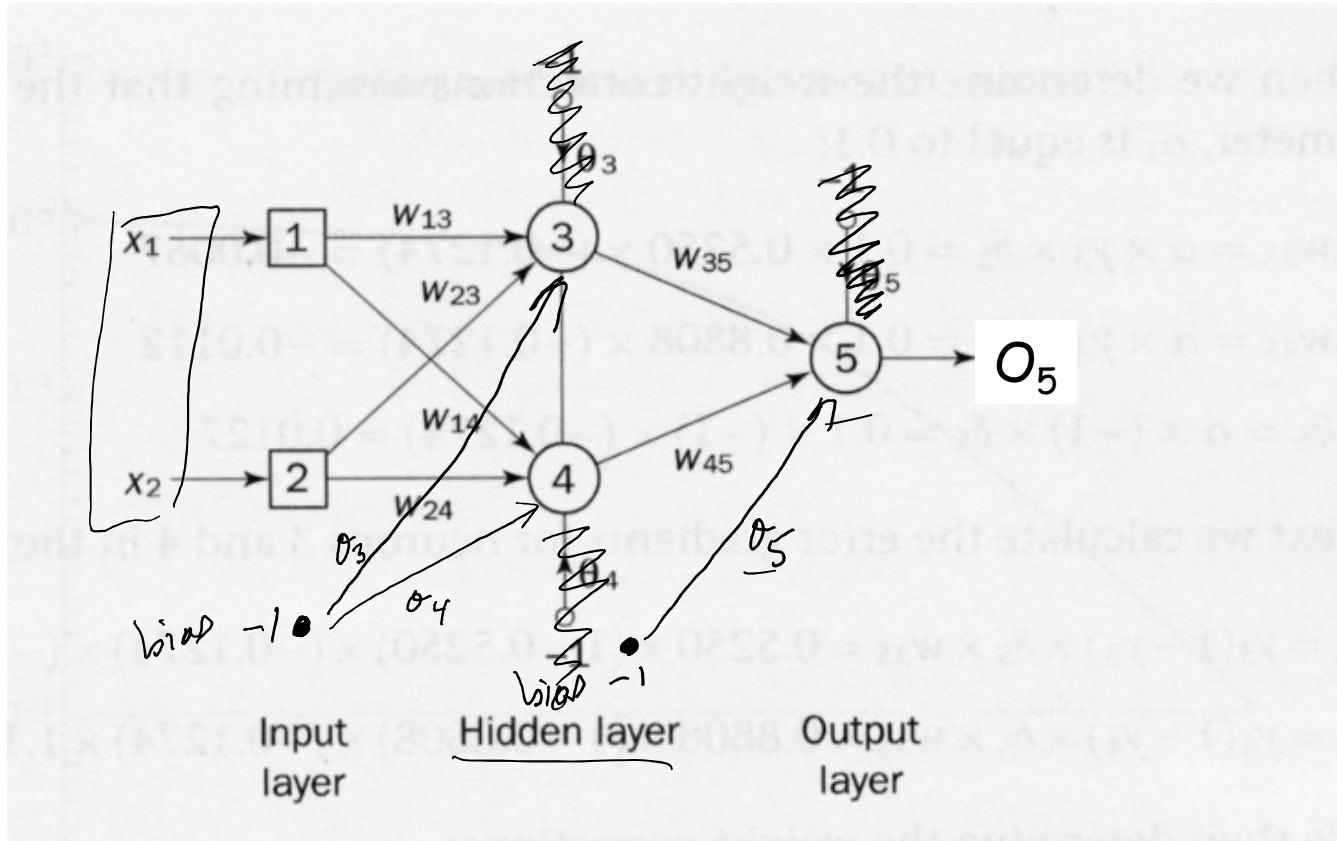
note, if we use $g = \text{sigmoid}$:
 $g'(x) = g(x)(1 - g(x))$

Sum of the weighted error term of the output nodes that h is connected to (ie. h contributed to the errors δ_k)

$\equiv x_i$ in slide #22

Example: XOR

$x_1 \text{ XOR } x_2$

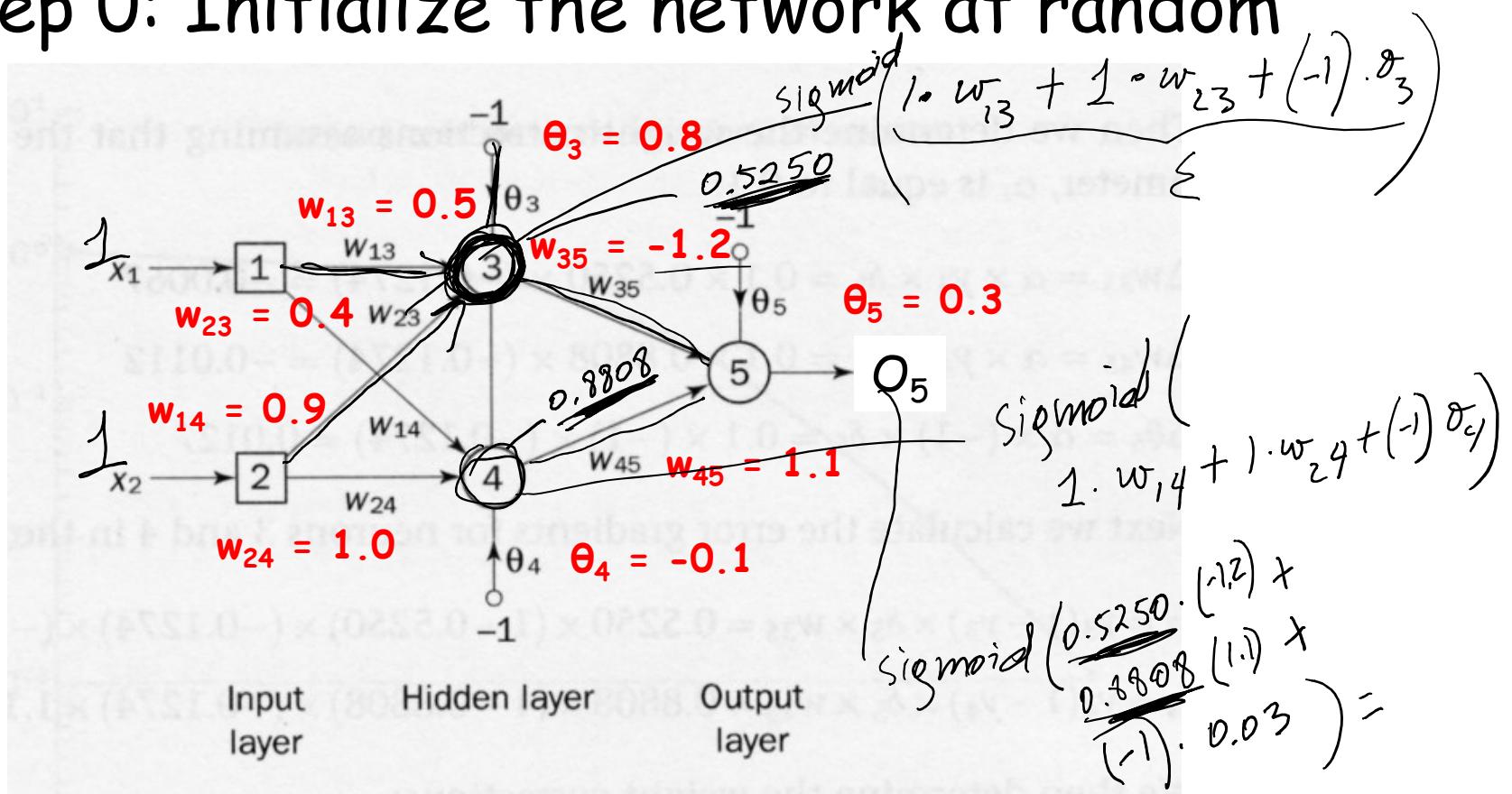


- 2 input nodes + 2 hidden nodes + 1 output node + 3 biases

source: Negnevitsky, Artificial Intelligence, p. 181

Example: Step 0 (initialization)

- Step 0: Initialize the network at random

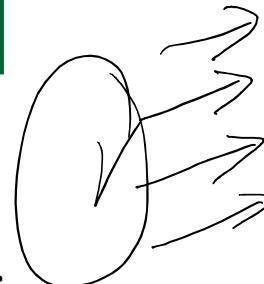


Step 1: Feed Forward

- Step 1: Feed the inputs and calculate the output

$$O_i = \text{sigmoid} \left(\sum_j w_{ji} x_j \right) = \frac{1}{1 + e^{-\sum_j w_{ji} x_j}}$$

- With $(x_1=1, x_2=1)$ as input:



x_1	x_2	Target output T
1	1	0
0	0	0
1	0	1
0	1	1

- Output of the hidden node 3:

□ $O_3 = \text{sigmoid}(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1 / (1 + e^{-(1 \times 5 + 1 \times 4 - 1 \times 8)}) = 0.5250 \checkmark$

- Output of the hidden node 4:

□ $O_4 = \text{sigmoid}(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1 / (1 + e^{-(1 \times 9 + 1 \times 1.0 + 1 \times 0.1)}) = 0.8808$

- Output of neuron 5:

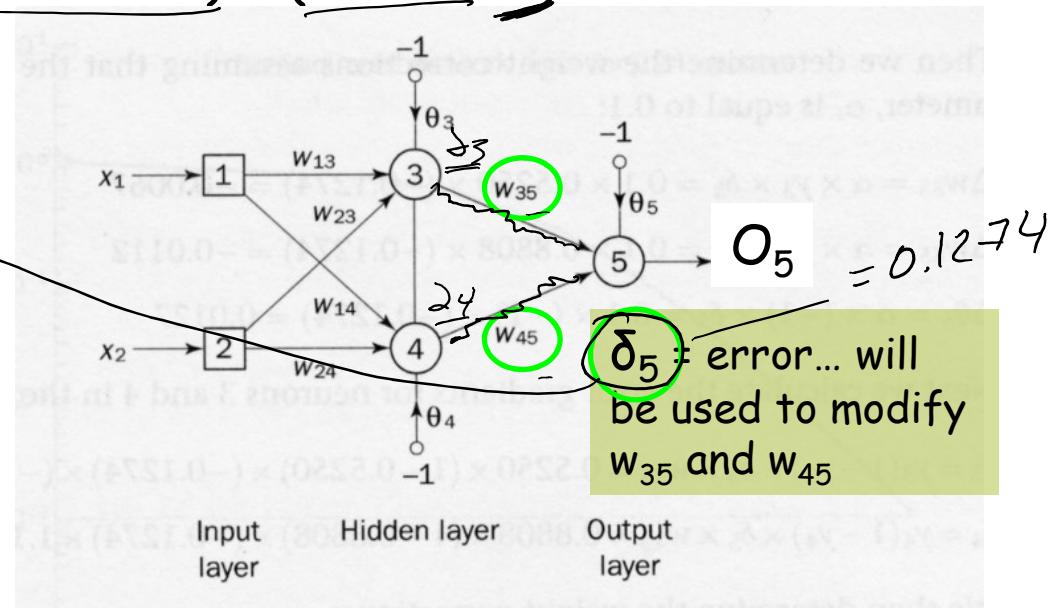
□ $O_5 = \text{sigmoid}(O_3 w_{35} + O_4 w_{45} - \theta_5) = 1 / (1 + e^{-(0.5250 \times 1.2 + 0.8808 \times 1.1 - 1 \times 0.3)}) = 0.5097$

Step 2: Calculate error term of output layer

$$\delta_k \leftarrow g'(x_k) \times \text{Err}_k = O_k(1 - O_k) \times (O_k - T_k)$$

- Error term of neuron 5 in the output layer:

□ $\delta_5 = O_5 (1-O_5) (O_5 - T_5)$
= $(0.5097) \times (1-0.5097) \times (0.5097-0)$
= 0.1274



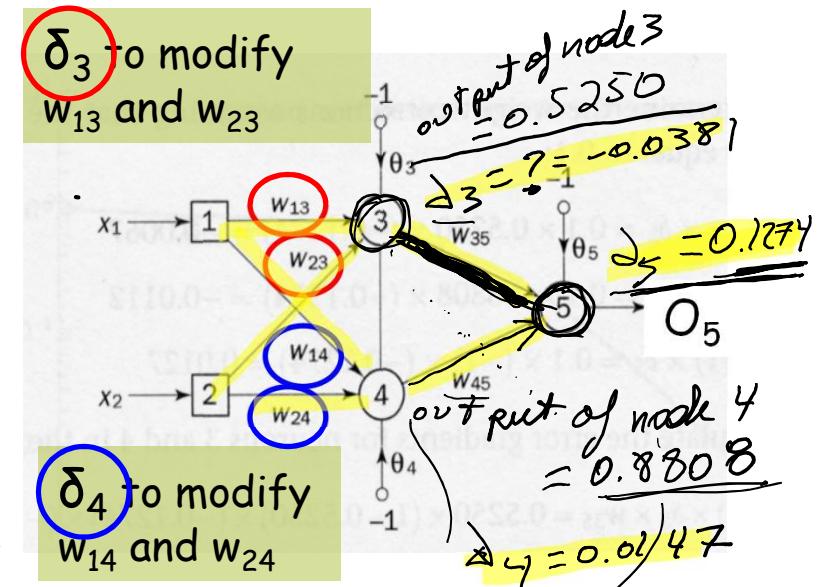
Step 3: Calculate error term of hidden layer

$$\delta_h \leftarrow g'(x_h) \times Err_h = O_k(1 - O_k) \times \sum_{k \in \text{outputs}} w_{kh} \delta_k$$

- Error term of neurons 3 & 4 in the hidden layer:

$\delta_3 = O_3(1-O_3) \delta_5 w_{35}$
 $= (0.5250) \times (1-0.5250) \times (0.1274) \times (-1.2)$
 $= -0.0381$

$\delta_4 = O_4(1-O_4) \delta_5 w_{45}$
 $= (0.8808) \times (1-0.8808) \times (0.1274) \times (1.1)$
 $= 0.0147$

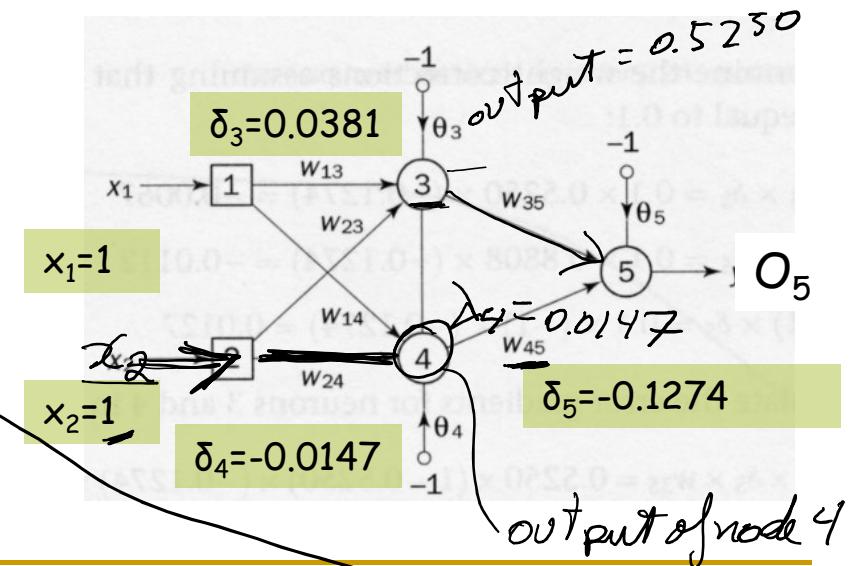


Step 4: Update Weights

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \text{ where } \Delta w_{ij} = -\eta \delta_j x_i$$

- Update all weights (assume a constant learning rate $\eta = 0.1$)

- $\Delta w_{13} = -\eta \delta_3 x_1 = -0.1 \times -0.0381 \times 1 = 0.0038$
- $\Delta w_{14} = -\eta \delta_4 x_1 = -0.1 \times 0.0147 \times 1 = -0.0015$
- $\Delta w_{23} = -\eta \delta_3 x_2 = -0.1 \times -0.0381 \times 1 = 0.0038$
- $\Delta w_{24} = -\eta \delta_4 x_2 = -0.1 \times 0.0147 \times 1 = -0.0015$
- $\Delta w_{35} = -\eta \delta_5 O_3 = -0.1 \times 0.1274 \times 0.5250 = -0.00669 // O_3 \text{ is seen as } x_5 \text{ (output of 3 is input to 5)}$
- $\Delta w_{45} = -\eta \delta_5 O_4 = -0.1 \times 0.1274 \times 0.8808 = -0.01122 // O_4 \text{ is seen as } x_5 \text{ (output of 4 is input to 5)}$
- $\Delta \theta_3 = -\eta \delta_3 (-1) = -0.1 \times -0.0381 \times -1 = -0.0038$
- $\Delta \theta_4 = -\eta \delta_4 (-1) = -0.1 \times 0.0147 \times -1 = -0.0015$
- $\Delta \theta_5 = -\eta \delta_5 (-1) = -0.1 \times 0.1274 \times -1 = -0.0127$

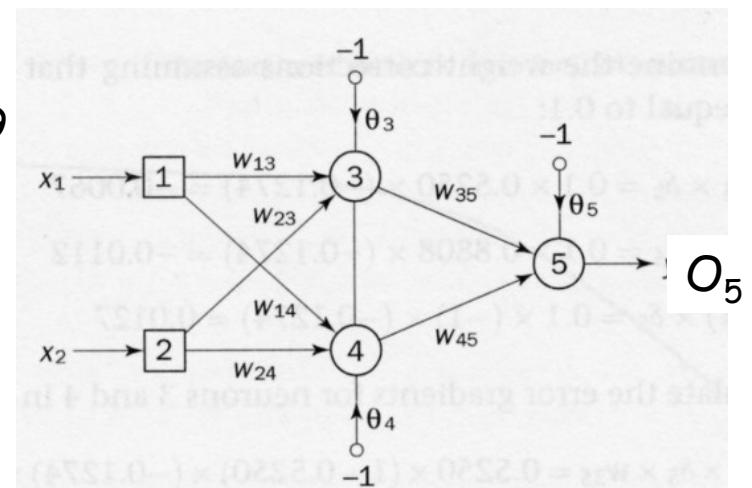


Step 4: Update Weights (con't)

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \text{ where } \Delta w_{ij} = -\eta \delta_j x_i$$

- Update all weights (assume a constant learning rate $\eta = 0.1$)

- $w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$
- $w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$
- $w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$
- $w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$
- $w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.00669 = -1.20669$
- $w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.01122 = 1.08878$
- $\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$
- $\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$
- $\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$



Step 4: Iterate through data

- after adjusting all the weights, repeat the forward pass and back pass for the next data point until all data points are examined
- repeat this entire exercise until the cost function is minimised

$$\text{Eg. } \underline{C} = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K (T^k_i \ln(O^k_i))$$

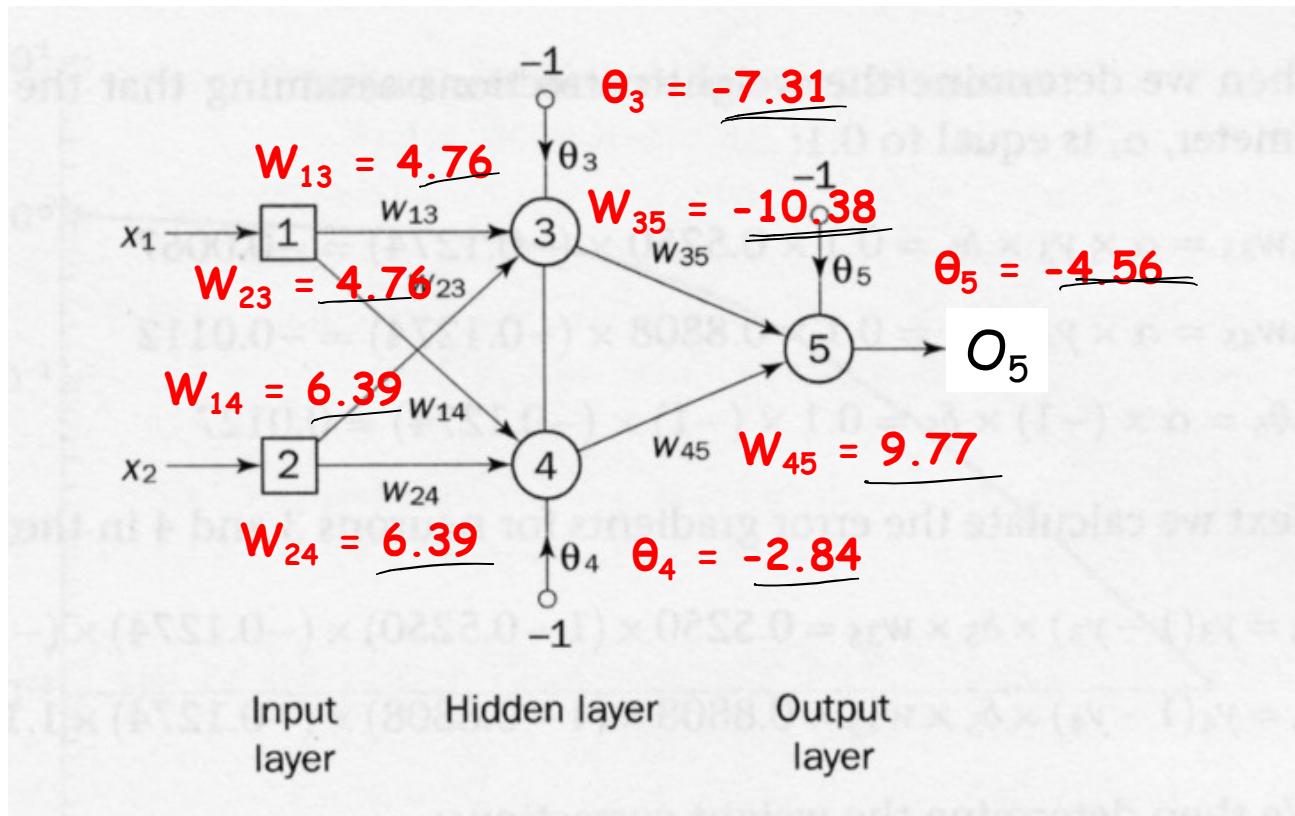
where

n = nb of training examples

K = nb of classes

The Result...

- After 224 epochs, we get:
 - (1 epoch = going through all data once)



Error is minimized

1-0.9845

Inputs		Target Output T	Actual Output O
x_1	x_2		
1	1	false (0)	0.0155
0	1	true (1)	0.9849
1	0	true (1)	0.9849
0	0	false (0)	0.0175

i	real distribution (false, true)	predicted distribution (false, true)
1	(1, 0)	0.9845 / 0.0155
2	(0, 1)	0.0151 / 0.9849
3	(0, 1)	(0.0151, 0.9849)
4	(1, 0)	(0.9825, 0.0175)

$$\begin{aligned}
 C = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K (T^k_i \ln(O^k_i)) &= -\frac{1}{n} \\
 &\quad (1) \ln(0.9845) + (0) \ln(0.0155) // \text{for } i=1 \\
 &\quad + (0) \ln(0.0151) + (1) \ln(0.9849) // \text{for } i=2 \\
 &\quad + (0) \ln(0.0151) + (1) \ln(0.9849) // \text{for } i=3 \\
 &\quad + (1) \ln(0.9825) + (0) \ln(0.0175) // \text{for } i=4 \\
 &= 0.01592 < \epsilon
 \end{aligned}$$

$n=4$

$K=2$ classes (false, true)

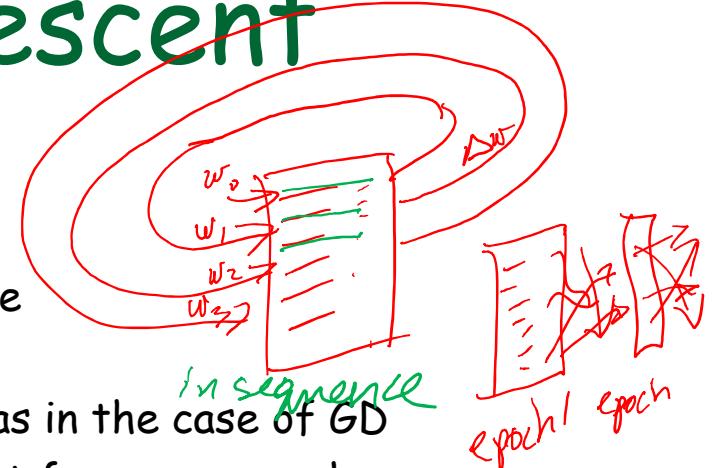


May be a local minimum...

Types of Gradient Descent

■ Stochastic Gradient Descent (SGD)

- updates the weights after each training example
- often converges faster compared to GD
- but the error function is not as well minimized as in the case of GD
- to obtain better results, shuffle the training set for every epoch



■ Batch Gradient Descent (GD)

- updates the weights after 1 epoch
- can be costly (time & memory) since we need to evaluate the whole training dataset before we take one step towards the minimum.



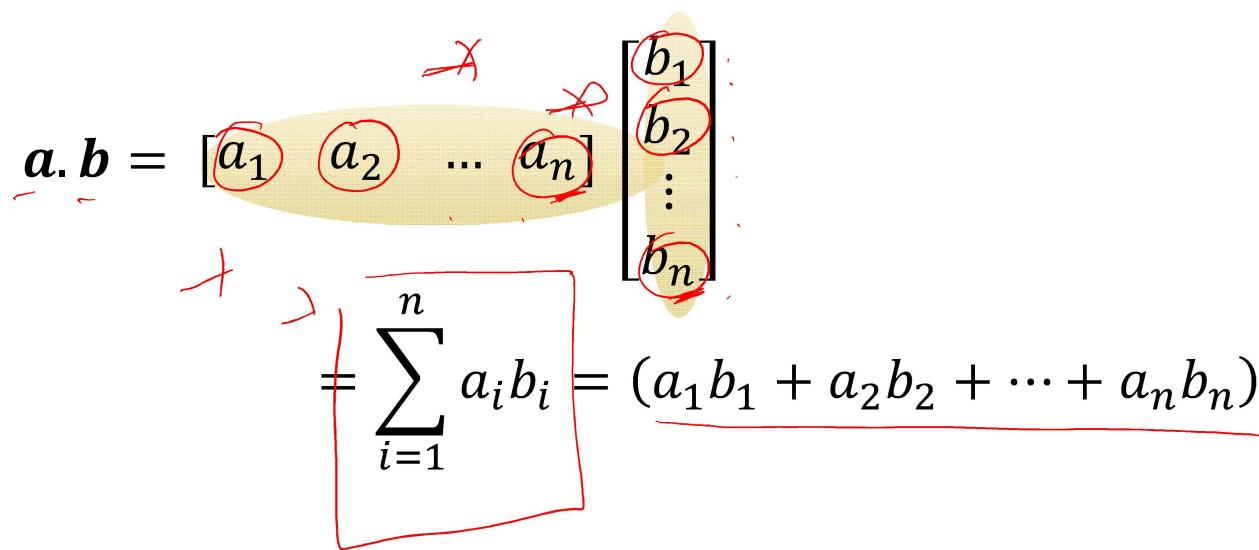
■ MiniBatch Gradient Descent:

- compromise between GD and SGD
- cut your dataset into sections, and update the weights after training on each section

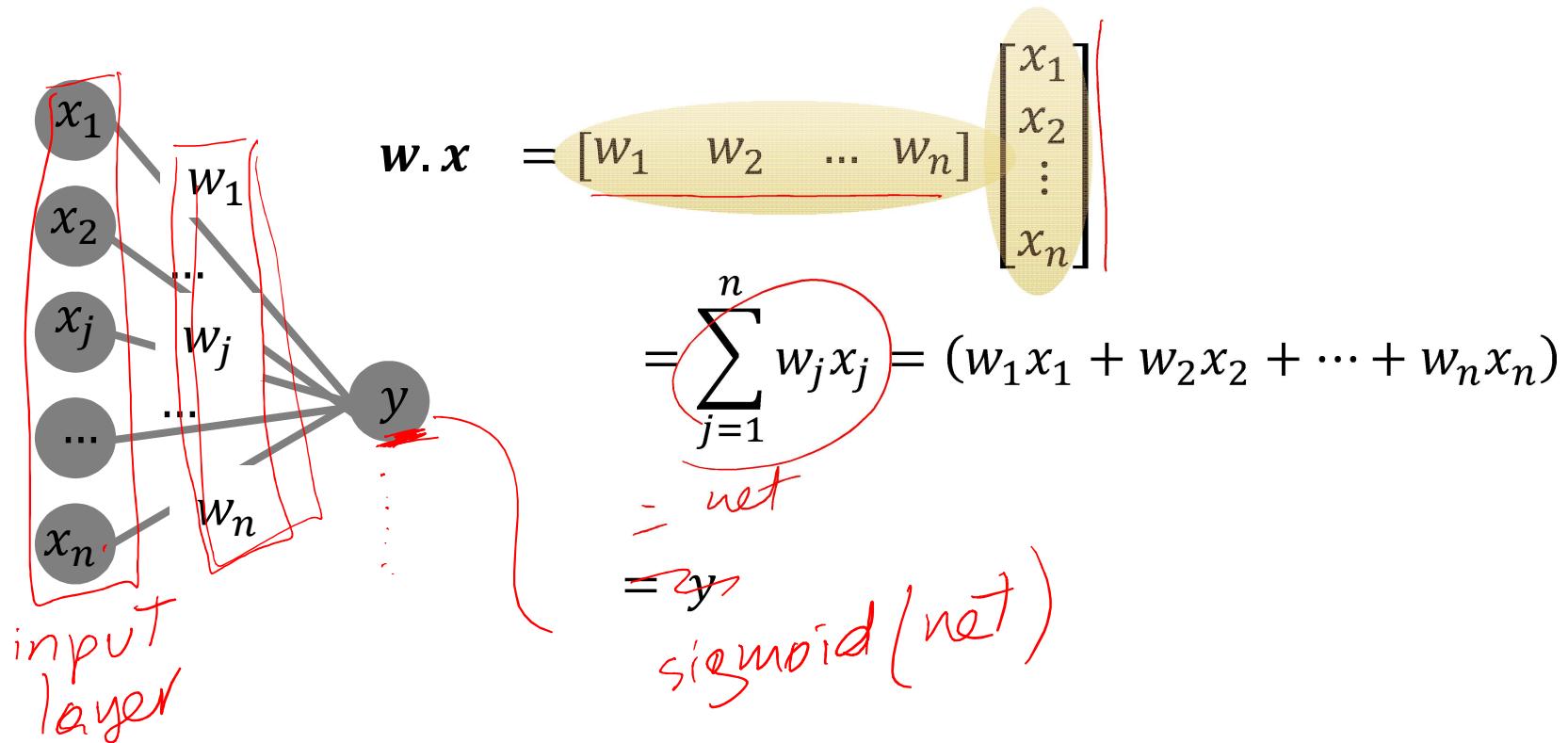


Remember your Linear Algebra

- Dot product (inner product) of 2 vectors

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= [a_1 \ a_2 \ \dots \ a_n] \begin{matrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{matrix} \\ &= \sum_{i=1}^n a_i b_i = (a_1 b_1 + a_2 b_2 + \dots + a_n b_n) \end{aligned}$$


so what?



Remember your Linear Algebra

- matrix-vector product

$$\begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \quad \begin{array}{c} m \times n \\ \curvearrowleft \\ \curvearrowleft \\ \curvearrowleft \\ \curvearrowleft \end{array} \quad \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array}$$
$$\begin{bmatrix} w_{11} & \dots & w_{1j} & \dots & w_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & w_{ij} & \dots & w_{in} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & w_{mj} & \dots & w_{mn} \end{bmatrix} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{bmatrix} \quad \begin{array}{c} n \times 1 \\ \curvearrowleft \\ \curvearrowleft \\ \curvearrowleft \\ \curvearrowleft \end{array} \quad \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array}$$
$$\begin{bmatrix} y_1 \\ \vdots \\ y_i \\ \vdots \\ y_m \end{bmatrix} \quad \begin{array}{c} m \times 1 \\ \curvearrowleft \\ \curvearrowleft \\ \curvearrowleft \\ \curvearrowleft \end{array}$$

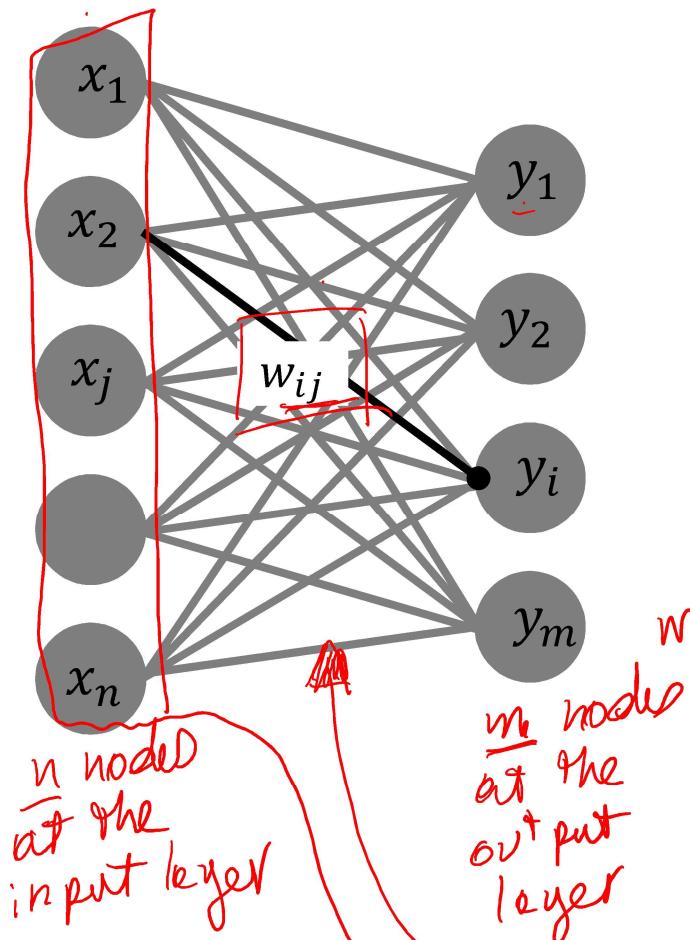
$j = 1 \rightarrow n$
 $i = 1 \rightarrow m$

where:

$y_i = \text{dot product of } i^{\text{th}} \text{ row of } W \text{ with } x$

$$y_i = \sum_{j=1}^n w_{ij} x_j$$

so what?



m hidden nodes

n input nodes

$w_{ij} = \text{weight from node } x_j \text{ to } y_i$

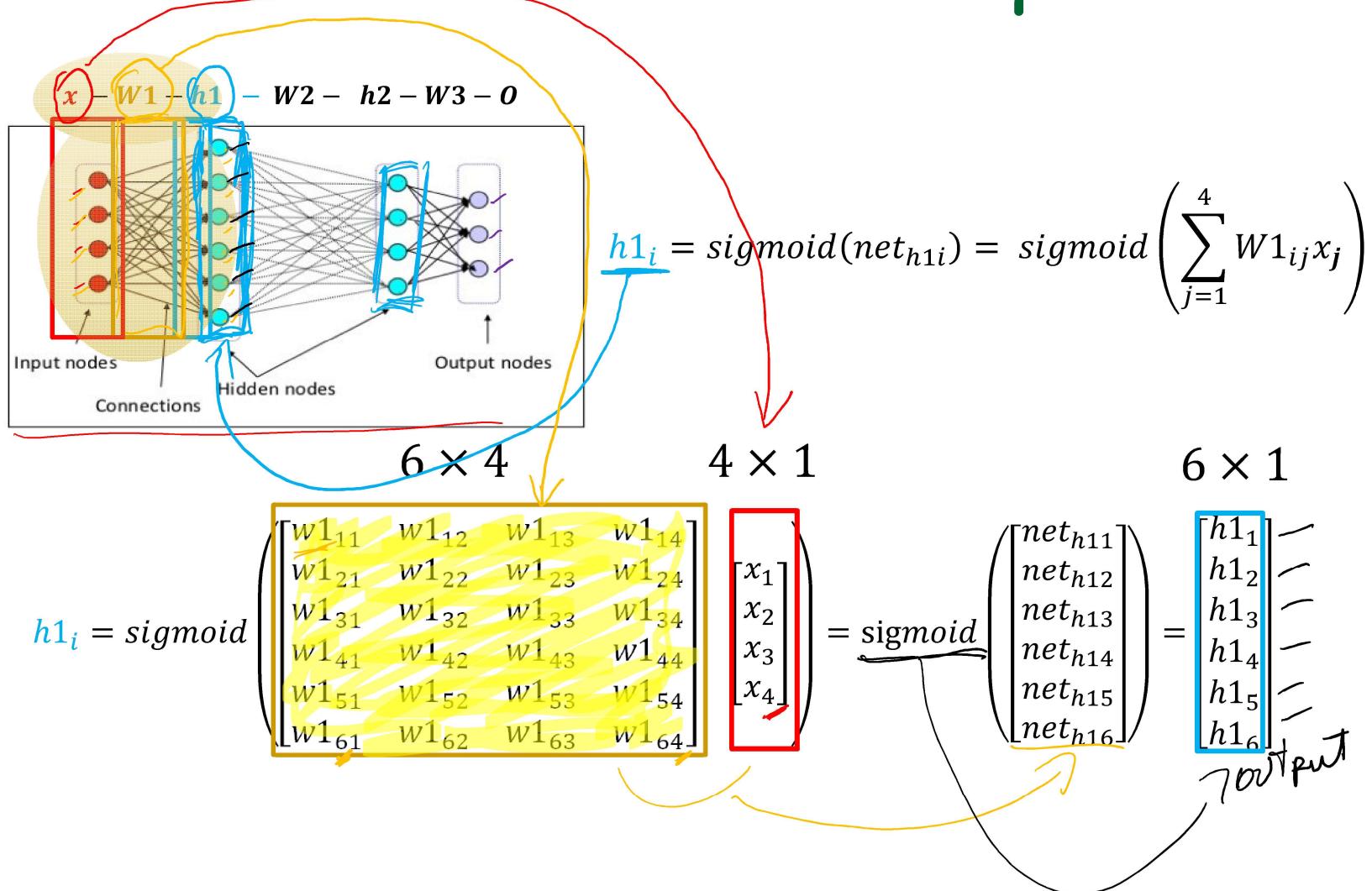
$$\underline{y_i} = \underline{\text{sigmoid}} \left(\sum_{j=1}^n w_{ij} x_j \right)$$

$$\begin{matrix} & & n \\ & & \vdots \\ \begin{bmatrix} w_{11} & \dots & w_{1j} & \dots & w_{1n} \\ \vdots & & \vdots & & \vdots \\ w_{i1} & w_{i2} & w_{ij} & \dots & w_{in} \\ \vdots & & \vdots & & \vdots \\ w_{m1} & w_{m2} & w_{mj} & \dots & w_{mn} \end{bmatrix} & \times & \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{bmatrix} & = & \begin{bmatrix} \text{net}_1 \\ \text{net}_2 \\ \vdots \\ \text{net}_i \\ \vdots \\ \text{net}_m \end{bmatrix} \end{matrix}$$

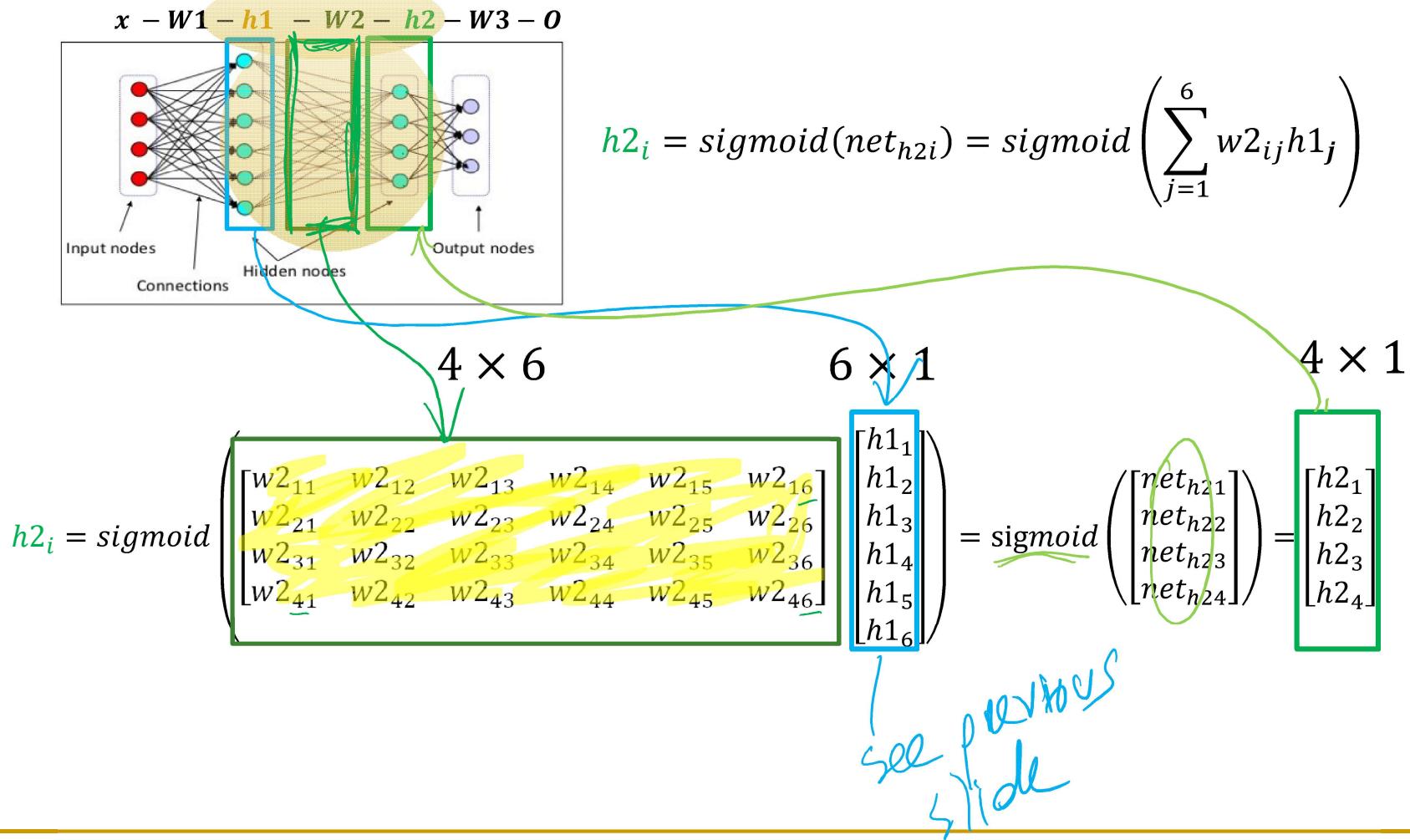
$\text{sigmoid} \left[\text{net} \right] = \underline{y_i}$

$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$

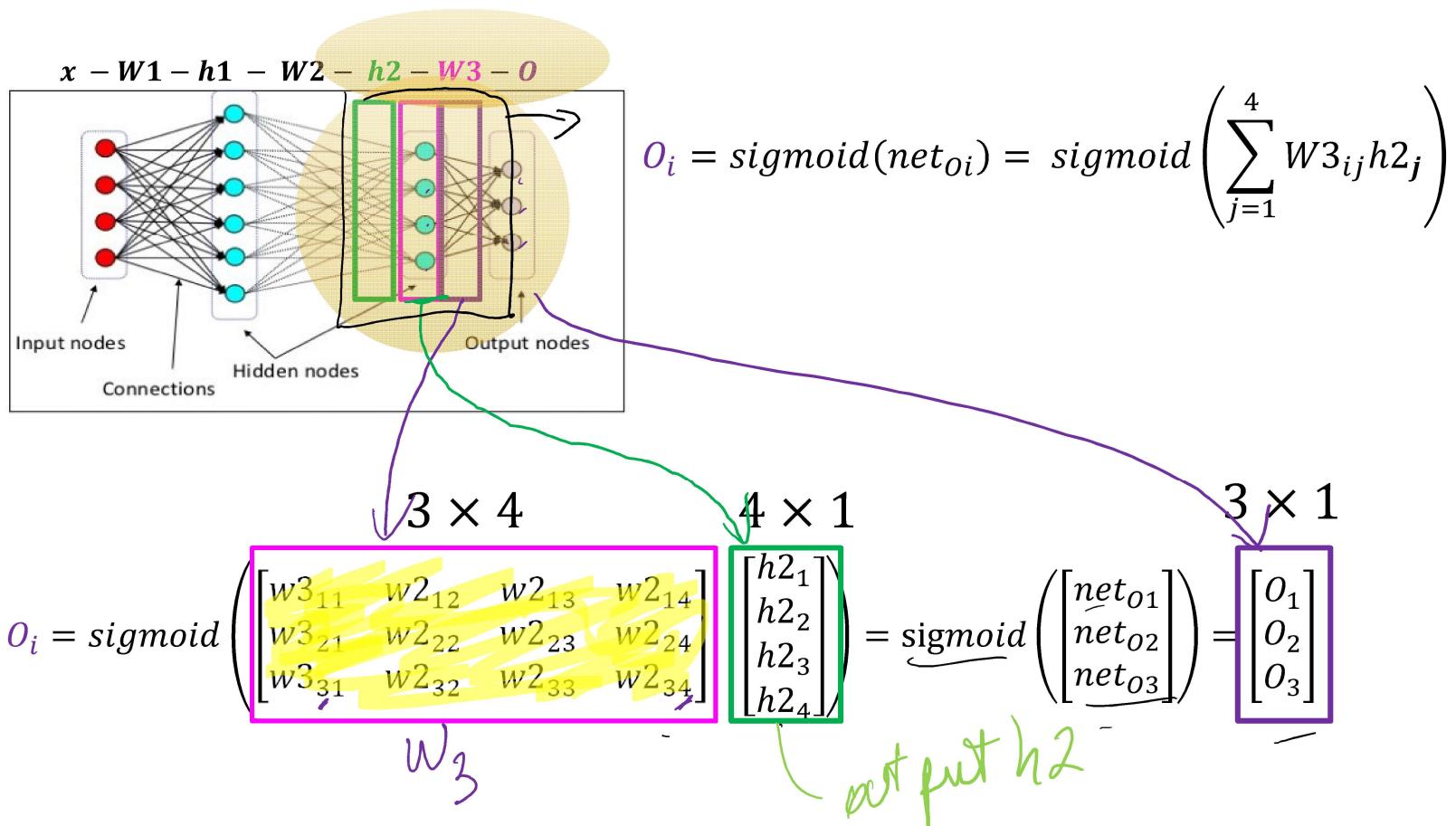
Matrix Notation - Example



Repeat on next level



Repeat on next level



Neural Networks

- Disadvantage:
 - result is not easy to understand by humans (set of weights compared to decision tree)... it is a black box
- Advantage:
 - robust to noise in the input (small changes in input do not normally cause a change in output) and graceful degradation

Today

1. Introduction to ML 
2. Naïve Bayes Classification

 - a. Application to Spam Filtering

3. Decision Trees
4. (Evaluation 
5. Unsupervised Learning) 
6. Neural Networks
 - a. Perceptrons 
 - b. Multi Layered Neural Networks 

Up Next

Part 4: Search