Machine Learning (CS 567)

Fall 2008

Time: T-Th 5:00pm - 6:20pm
Location: GFS 118

Instructor: Sofus A. Macskassy (macskass@usc.edu)
Office: SAL 216
Office hours: by appointment

Teaching assistant: Cheol Han (cheolhan@usc.edu)
Office: SAL 229
Office hours: M 2-3pm, W 11-12

Class web page:
http://www-scf.usc.edu/~csci567/index.html

Administrative

- Presentations
  - Two long classes on Dec 2 + 4
  - Attendance mandatory for all on both days
- HW 3
  - Get weka
  - Start right away

Learning Neural Networks

- Neural Networks can represent complex decision boundaries
  - Variable size. Any boolean function can be represented. Hidden units can be interpreted as new (virtual) features.
  - Deterministic
  - Continuous Parameters
- Learning Algorithms for neural networks
  - Local Search. The same algorithm as for sigmoid threshold units
  - Eager
  - Batch or Online

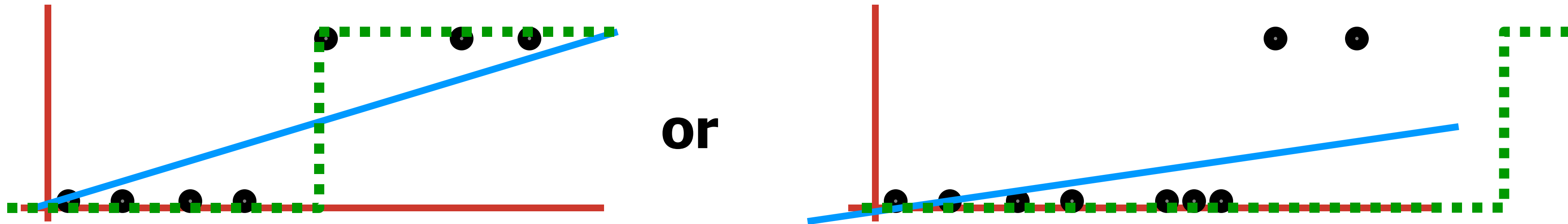
Perceptrons Revisited

- Recall basic perceptron:
  - $h(\mathbf{x}) = \text{sgn}(u(\mathbf{x},\mathbf{w}))$
- The perceptron is quasi-linear and minimizes an approximation to the 0-1 loss (hinge loss)
- Uses gradient descent to find optimal **w**:

$$\mathbf{w}_1 = \mathbf{w}_0 - \eta \nabla \tilde{J}(\mathbf{w}_0)$$

Perceptrons for Classification

What if all outputs are 0's or 1's ?



Blue = Out(x)

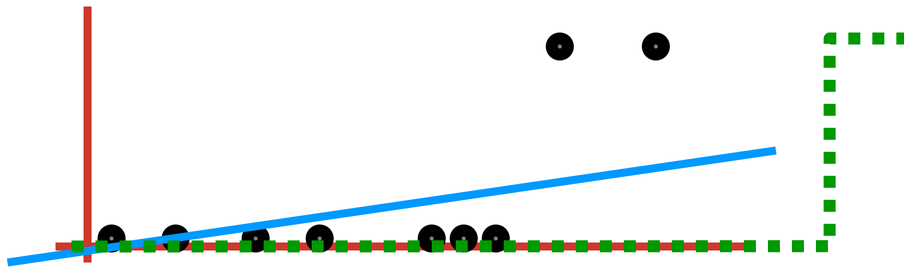
Green = Classification

We can do a linear fit.

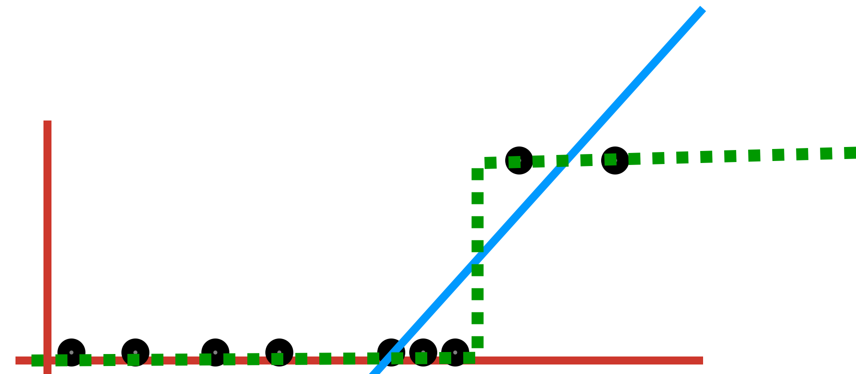
Our prediction is 0 if $out(\boldsymbol{x}) \leq ½$

1 if $out(\boldsymbol{x}) > ½$

Classification with Perceptrons



Least squares fit useless



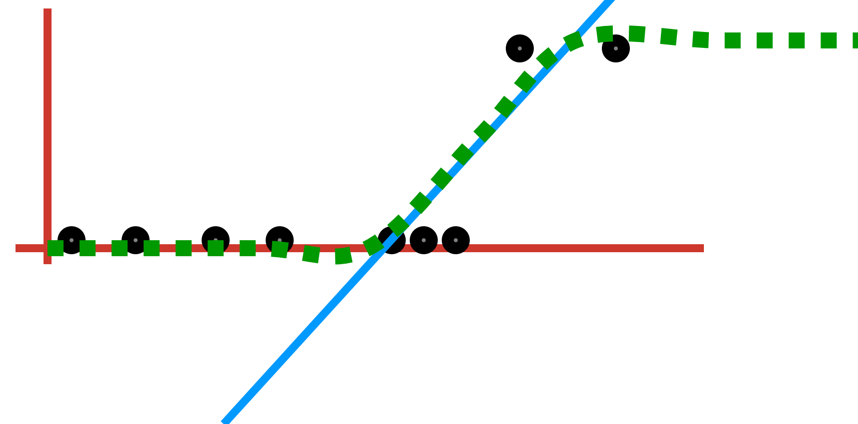
This fit would classify much better. But not a least squares fit.

SOLUTION:

Instead of g(***x***) = sgn(u(***w,x***))

We'll use g(***x***) = *g*(***w·x***)

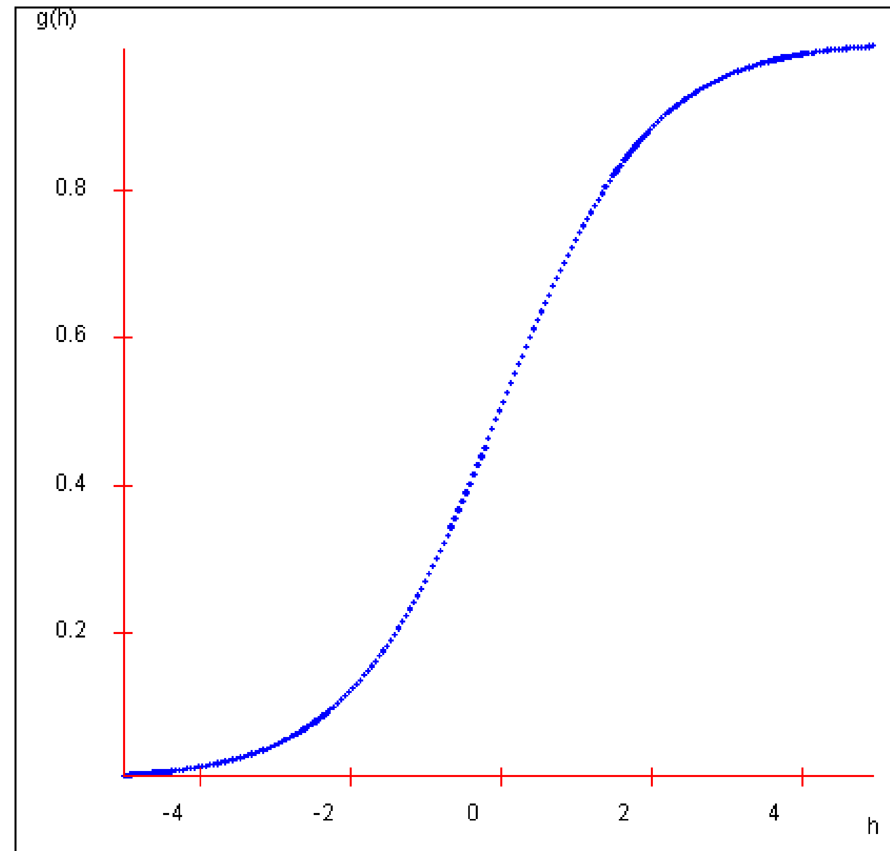
where $g(x):\Re \to (0,1)$ is a squashing function



The Sigmoid Objective Function

$$g(\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{x})}$$

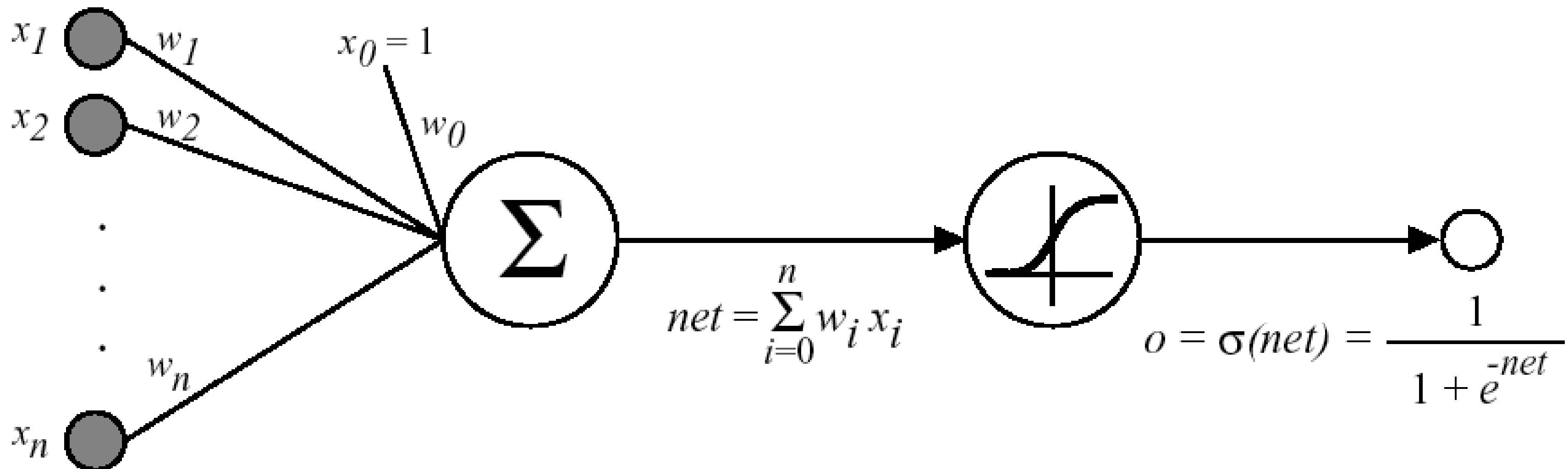
$$\begin{aligned} J(\mathbf{x}) &= \frac{1}{2}\sum_{i=0}^{N}[y_i - f(\mathbf{x_i})]^2 \\ &= \frac{1}{2}\sum_{i=0}^{N}[y_i - g(\mathbf{w}\cdot\mathbf{x_i})]^2 \end{aligned}$$



This is known as the minimum squared error (MSE) function.

Now choose ***w*** to minimize MSE.

Sigmoid perceptron unit



- $\sigma(x)$ is the sigmoid function $\frac{1}{1+e^{-x}}$
- Nice property: $\frac{d\sigma(x)}{dx} = -\sigma(x)(1-\sigma(x))$
- We can compute gradient rules to train
  - One sigmoid unit
  - Multi-layer networks of sigmoid units $\rightarrow$ backpropagation

Let's show that $\frac{d\sigma(x)}{dx} = -\sigma(x)(1-\sigma(x))$

$$\sigma(x) = \frac{1}{1+e^{-x}} \rightarrow \frac{d\sigma(x)}{dx} = \frac{-e^{-x}}{\left(1+e^{-x}\right)^2}$$

$$= \frac{1-1-e^{-x}}{\left(1+e^{-x}\right)^2}$$

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

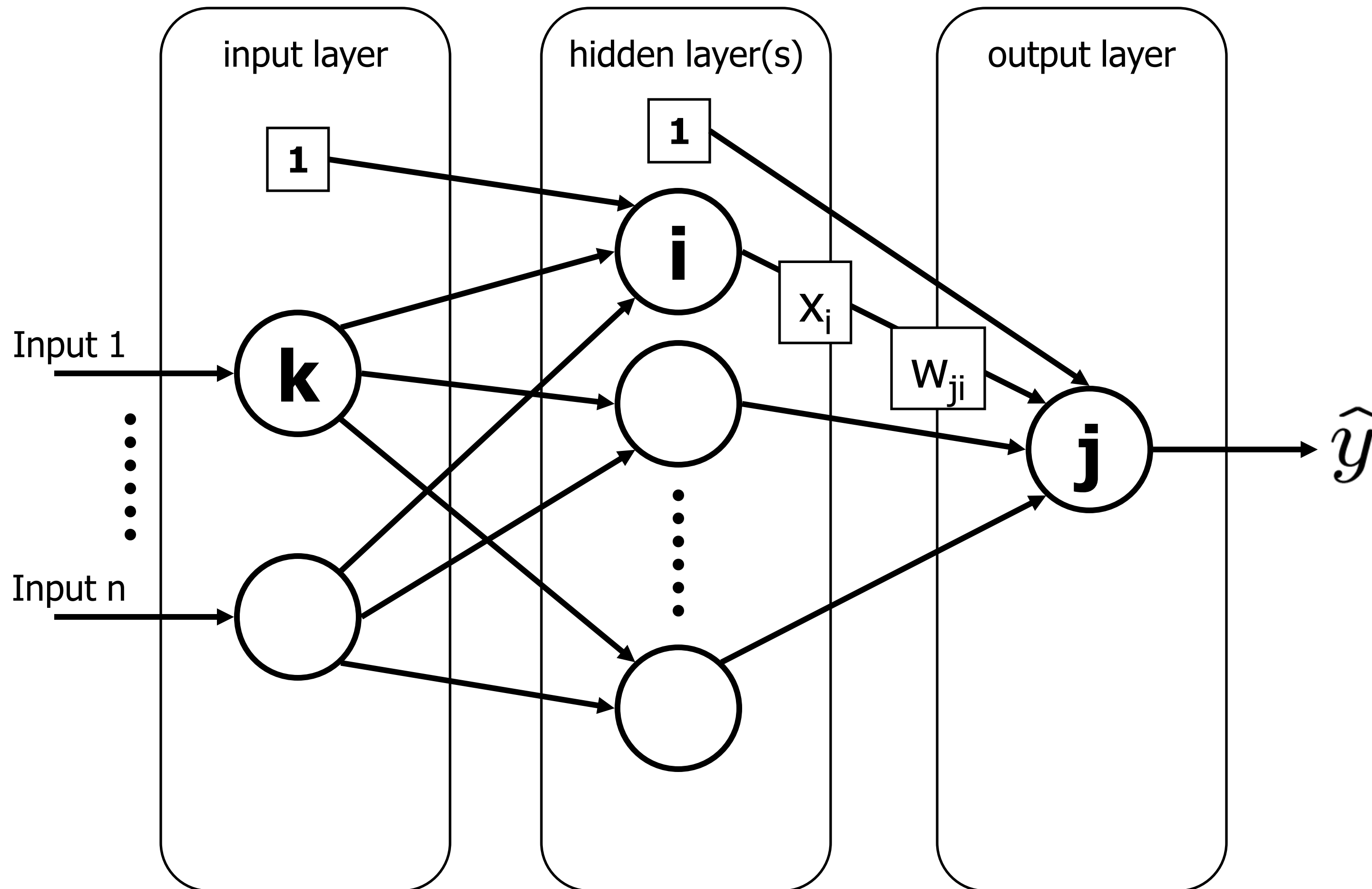
$$= -\sigma(x)(1-\sigma(x))$$

The need for networks

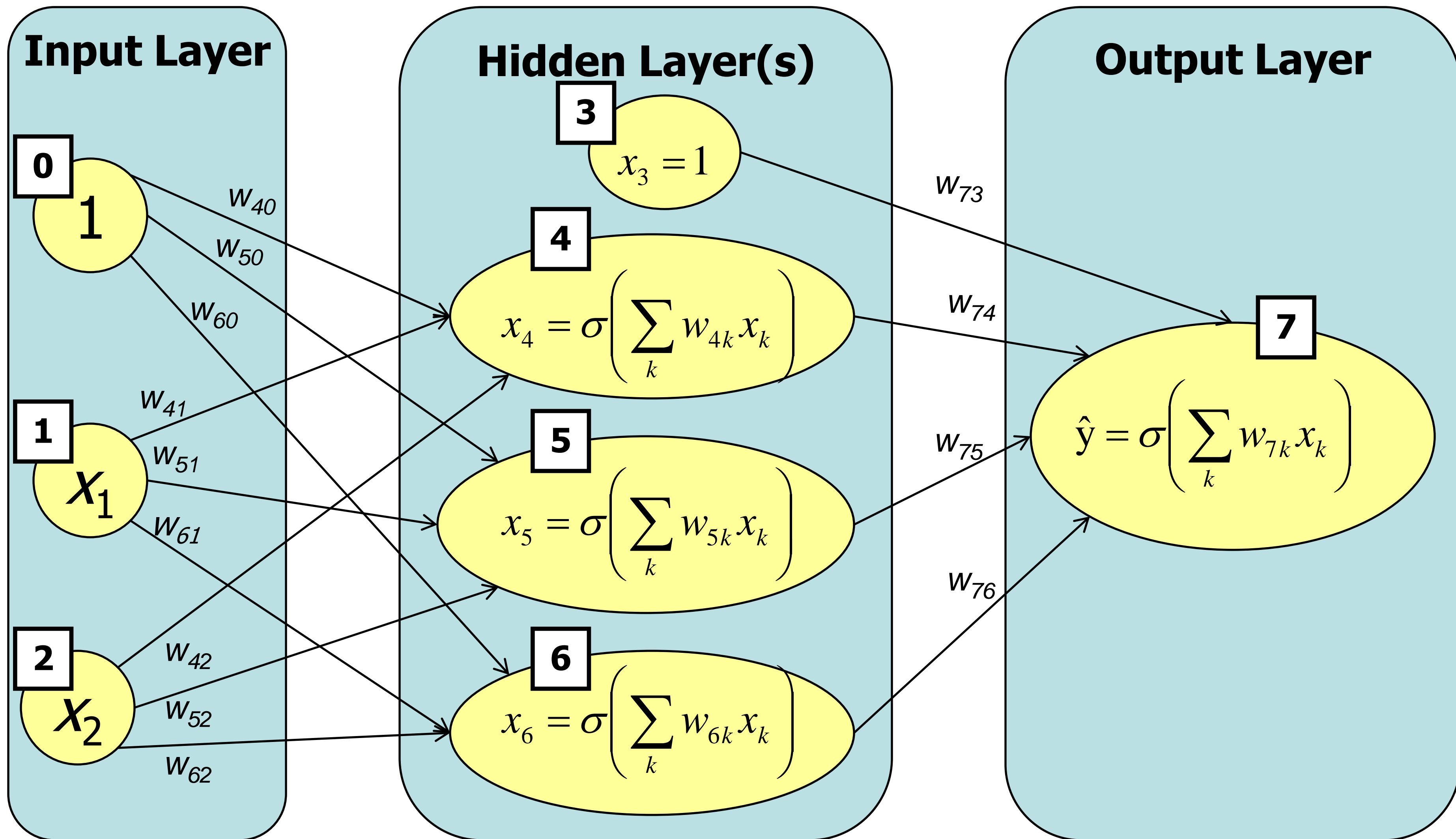
- Sigmoid units are very similar to perceptrons, but provide a “soft” threshold
- However, their expressive power is still the same as perceptrons: limited to linearly separable instances.
  - One sigmoid neuron can learn the AND, OR, NOT functions, but not XOR
- In order to learn discrimination in data sets that are not linearly separable, we need *networks of sigmoid units*

Some Terminology and Notation

Feed-forward (neural) network:



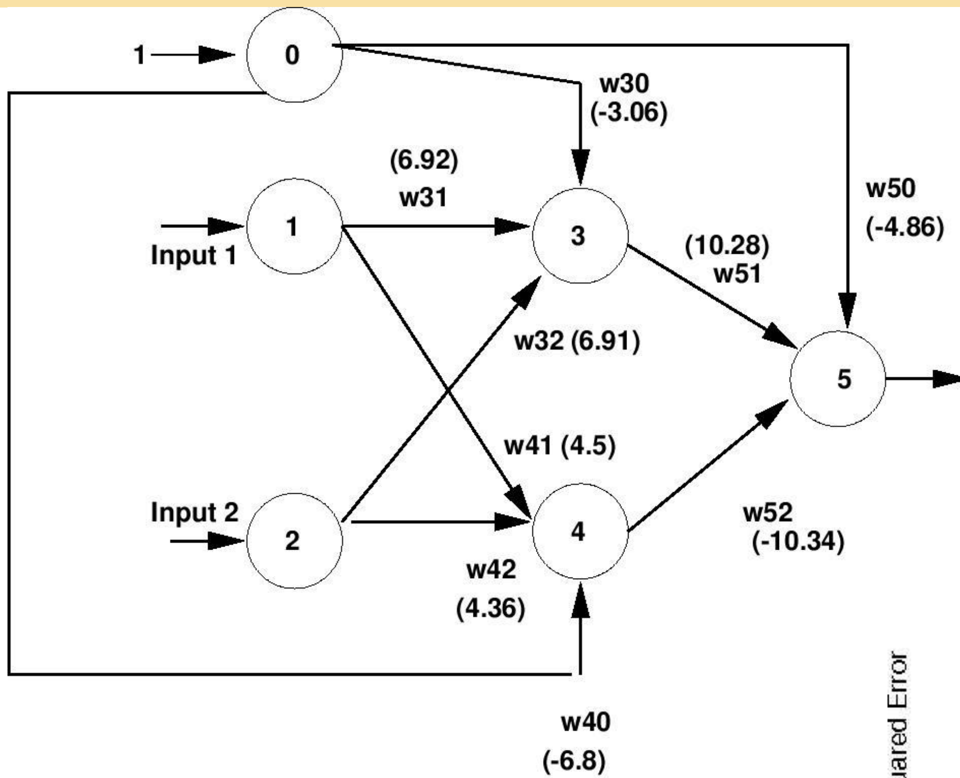
A 1-HIDDEN LAYER NET



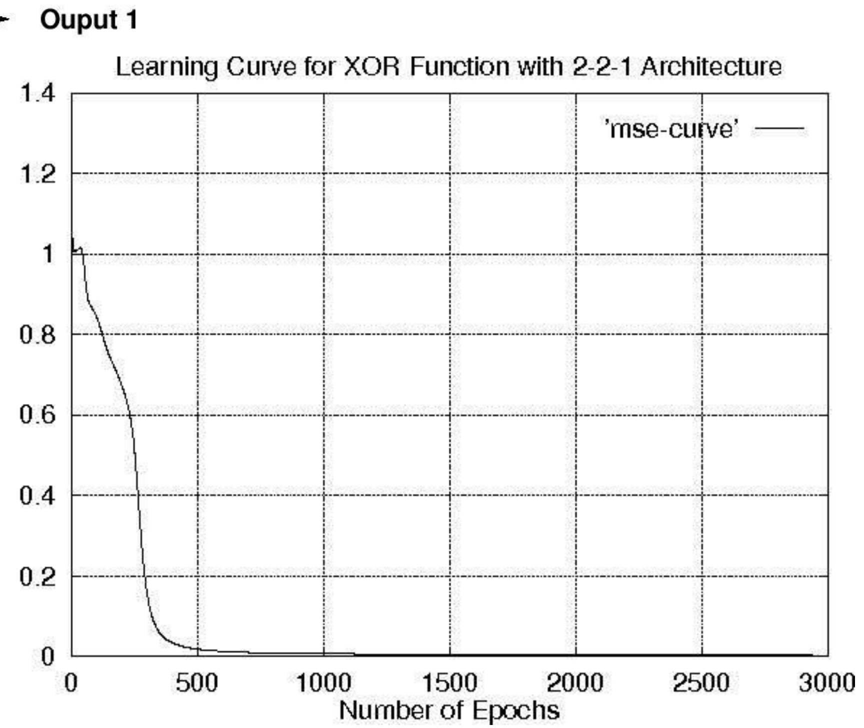
Representational Power

- <u>A single sigmoid neuron:</u>
  - has the same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR
- <u>Any Boolean Formula</u>
  - Consider a formula in disjunctive normal form:
    $(x_1 \wedge \neg x_2) \vee (x_2 \wedge x_4) \vee (\neg x_3 \wedge x_5)$
    Each AND can be represented by a hidden unit and the OR can be represented by the output unit. Arbitrary boolean functions require exponentially-many hidden units, however.
- <u>Bounded functions</u>
  - Suppose we make the output linear: $\hat{y} = \sigma\left(\sum_k w_{7k} x_k\right)$ of hidden units (from previous slide). It can be proved that any bounded continuous function can be approximated to arbitrary accuracy with enough hidden units. [Cybenko 1989; Hornik et al. 1989]
- <u>Arbitrary Functions</u>
  - Any function can be approximated to arbitrary accuracy with two hidden layers of sigmoid units and a linear output unit. [Cybenko 1988]

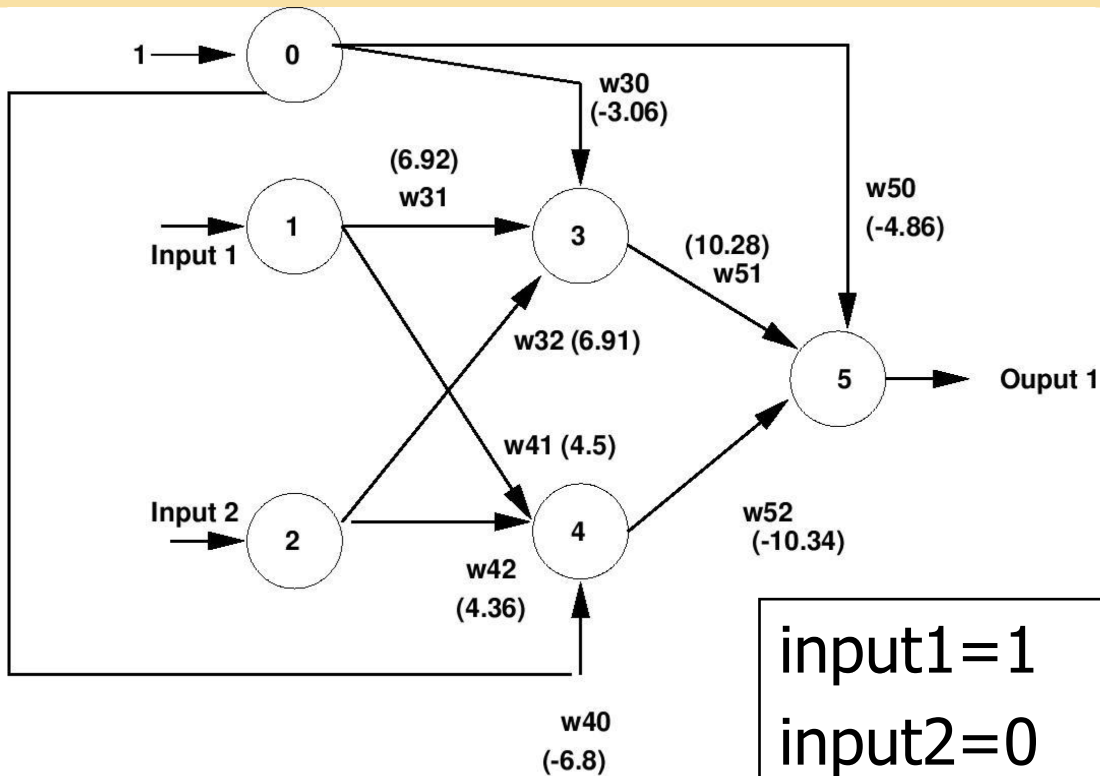
Example: A network representing the XOR function



Input1	Input2	x3	x4	Ouput 1
0	0	0.04	0.001	0.011
0	1	0.98	0.08	0.99
1	0			
1	1			



Example: A network representing the XOR function



Input1	Input2	x3	x4	Ouput 1
0	0	0.04	0.001	0.011
0	1	0.98	0.08	0.99
1	0			
1	1			

input1=1

input2=0

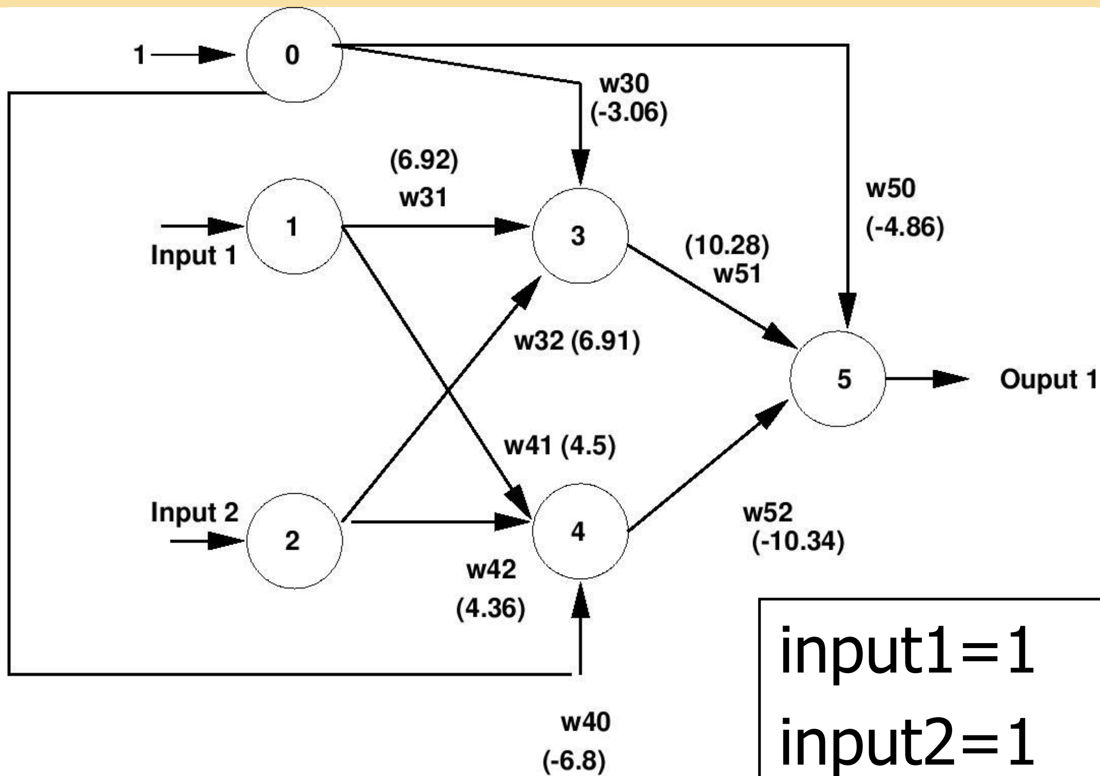
x3=σ(w30+w31) =0.979

x4=σ(w40+w41) =0.091

Output1=σ(w50+x3*w51 + x4*w2)

=0.986

Example: A network representing the XOR function



Input1	Input2	x3	x4	Ouput 1
0	0	0.04	0.001	0.011
0	1	0.98	0.08	0.99
1	0			
1	1			

input1=1

input2=1

x3=σ(w30+w31+w32) =0.999

x4=σ(w40+w41+w42) =0.887

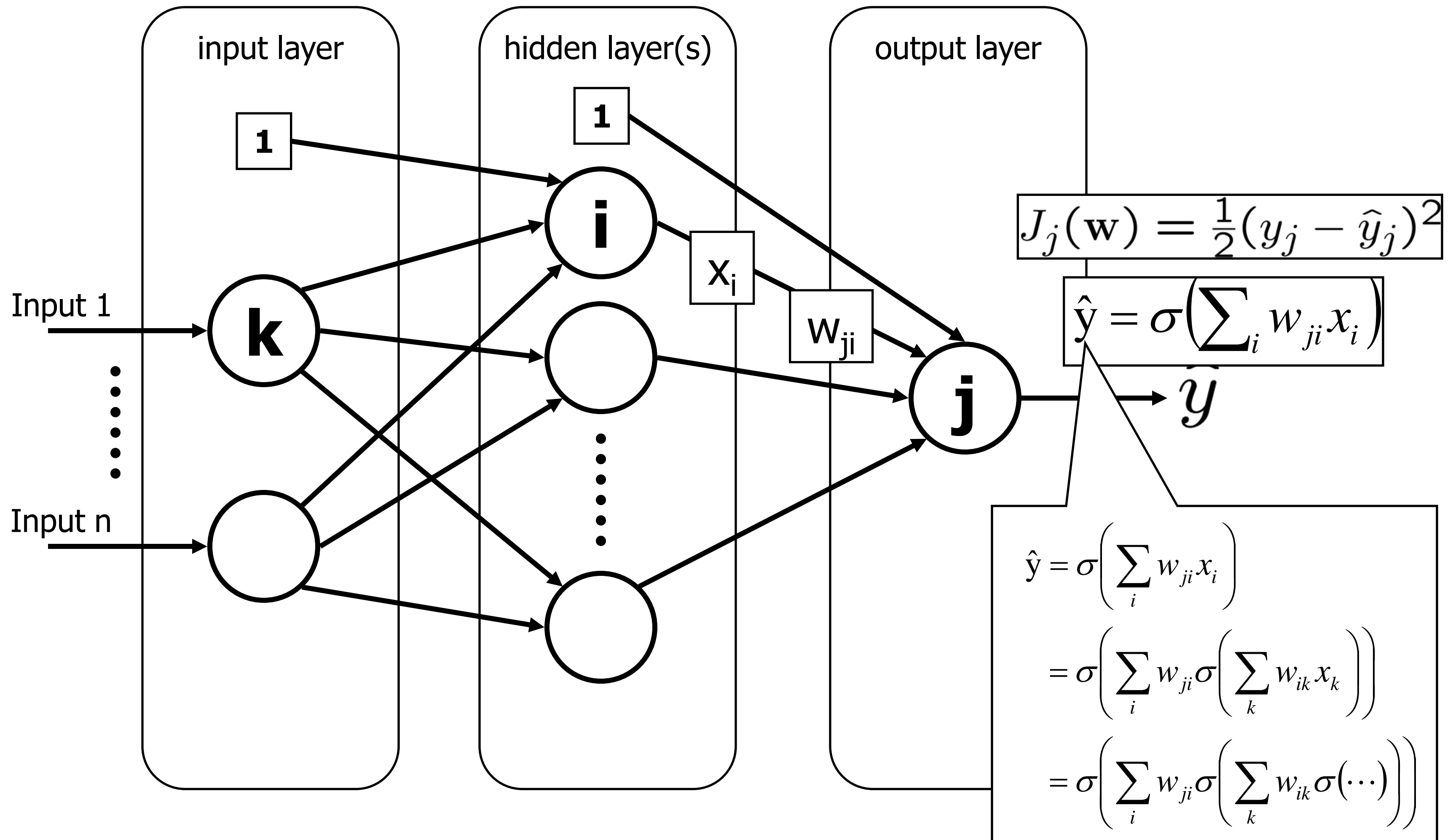
Output1=σ(w50+x3*w51 + x4*w2)

=0.023

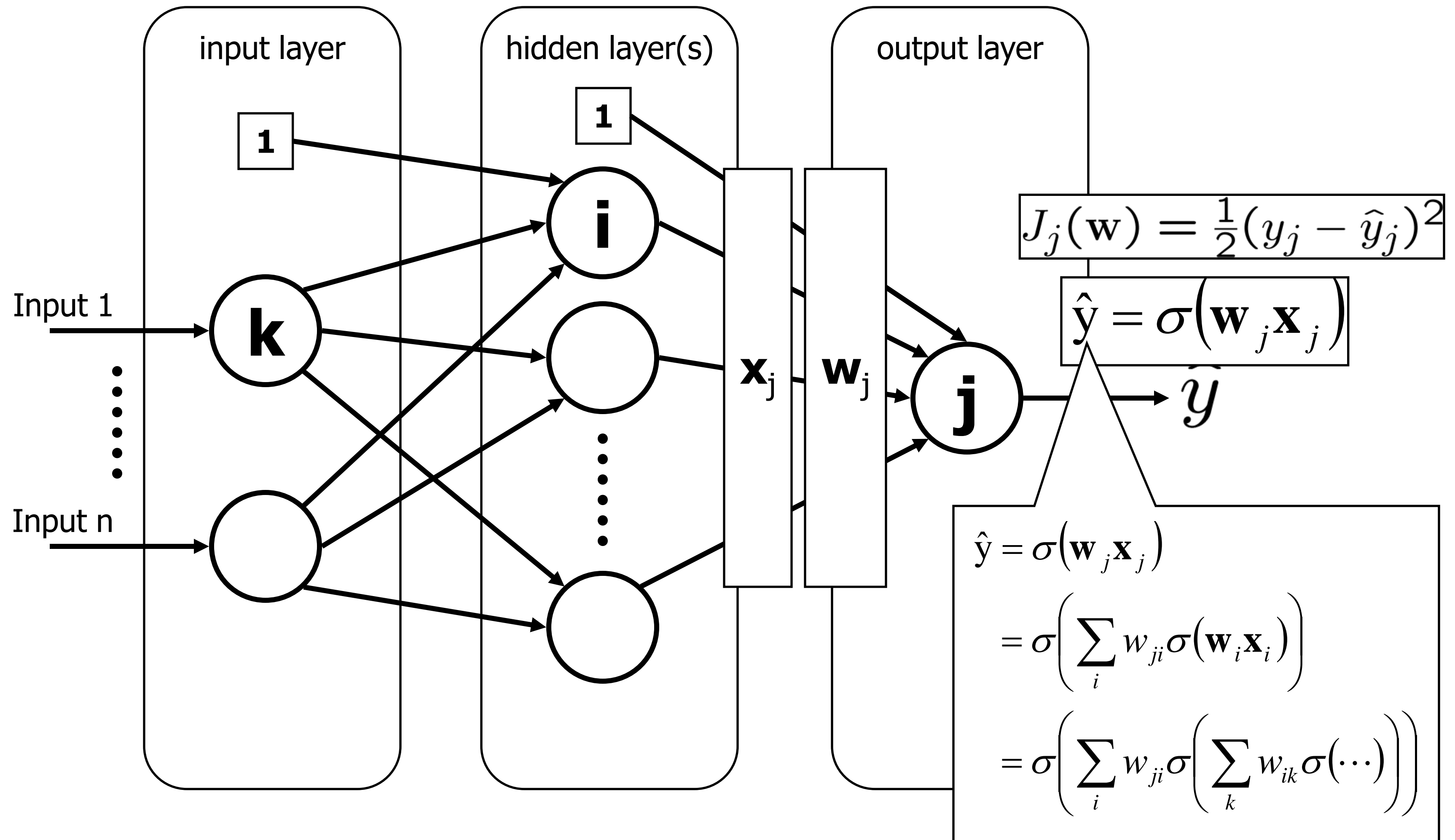
Learning in feed-forward neural nets

- Usually, the network structure (units and interconnections) is specified by the designer
- The learning problem is finding a *good set of weights*
- The answer: *gradient descent*, because the form of the hypothesis formed by the networks, h_w, is
  - Differentiable! Because of the choice of sigmoid units
  - Very complex! Hence, direct computation of the optimal weights is not possible

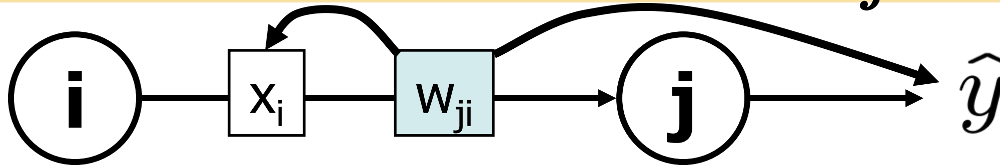
Some Terminology and Notation



Some Terminology and Notation

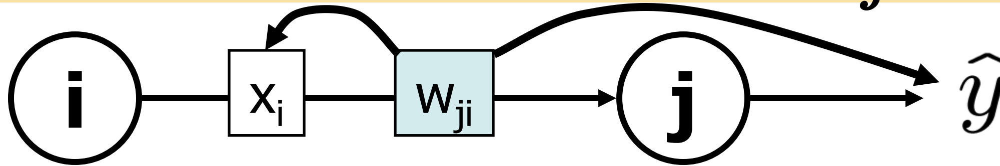


Derivation: Output Unit (updating w_{ji})



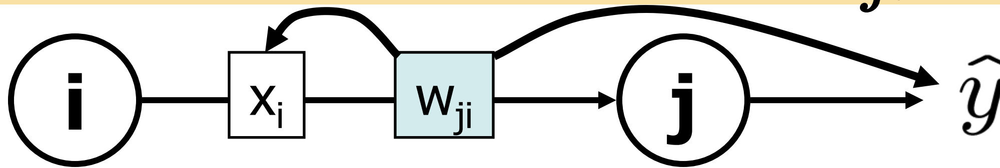
$$\frac{\partial J(\mathbf{w})}{\partial w_{ji}}$$

Derivation: Output Unit (updating w_{ji})



$$\frac{\partial J(\mathbf{w})}{\partial w_{ji}} = \left(\frac{\partial \hat{y}}{\partial w_{ji}}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

Derivation: Output Unit (updating w_{ji})

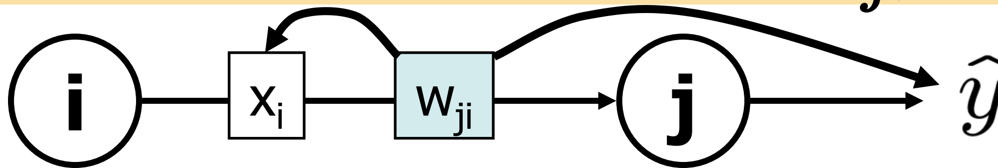


$$\frac{\partial J(\mathbf{w})}{\partial w_{ji}} = \left(\frac{\partial \hat{y}}{\partial w_{ji}}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \hat{y}}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

Remember: $\mathbf{w_j} \cdot \mathbf{x_j} = \sum_i w_{ji} \cdot x_i$

Derivation: Output Unit (updating w_{ji})



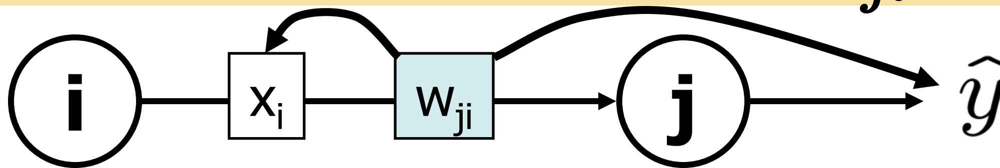
Remember: $\mathbf{w_j} \cdot \mathbf{x_j} = \sum_i w_{ij} \cdot x_i$

$$\frac{\partial J(\mathbf{w})}{\partial w_{ji}} = \left(\frac{\partial \hat{y}}{\partial w_{ji}}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \hat{y}}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \sigma(\mathbf{w}_j \mathbf{x}_j)}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial \frac{1}{2}(y-\hat{y})^2}{\partial \hat{y}}\right)$$

Derivation: Output Unit (updating w_{ji})



Remember: $\mathbf{w}_j \cdot \mathbf{x}_j = \sum_i w_{ij} \cdot x_i$

$$\frac{\partial J(\mathbf{w})}{\partial w_{ji}} = \left(\frac{\partial \hat{y}}{\partial w_{ji}}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

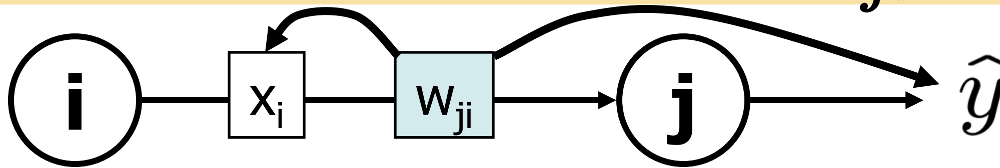
$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \hat{y}}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

Because $\sigma'(x) = \sigma(x)(1-\sigma(x))$

$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \sigma(\mathbf{w}_j \mathbf{x}_j)}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial \frac{1}{2}(y-\hat{y})^2}{\partial \hat{y}}\right)$$

$$= (x_i)\left(\sigma(\mathbf{w}_j \mathbf{x}_j)(1-\sigma(\mathbf{w}_j \mathbf{x}_j))\right)\left(2 \cdot \tfrac{1}{2}(\hat{y}-y)\right)$$

Derivation: Output Unit (updating w_{ji})



Remember: $\mathbf{w_j} \cdot \mathbf{x_j} = \sum_i w_{ij} \cdot x_i$

$$\frac{\partial J(\mathbf{w})}{\partial w_{ji}} = \left(\frac{\partial \hat{y}}{\partial w_{ji}}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

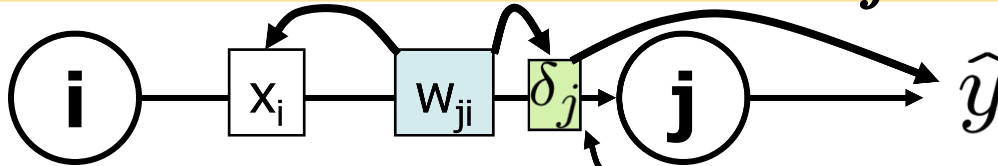
$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \hat{y}}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \sigma(\mathbf{w}_j \mathbf{x}_j)}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial \frac{1}{2}(y - \hat{y})^2}{\partial \hat{y}}\right)$$

$$= (x_i)\left(\sigma(\mathbf{w}_j \mathbf{x}_j)(1 - \sigma(\mathbf{w}_j \mathbf{x}_j))\right)\left(2 \cdot \tfrac{1}{2}(\hat{y} - y)\right)$$

$$= (x_i)(\hat{y})(1 - \hat{y})(\hat{y} - y)$$

Derivation: Output Unit (updating w_{ji})



$$\frac{\partial J(\mathbf{w})}{\partial w_{ji}} = \left(\frac{\partial \hat{y}}{\partial w_{ji}}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \hat{y}}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial J(\mathbf{w})}{\partial \hat{y}}\right)$$

$$= \left(\frac{\partial \mathbf{w}_j \mathbf{x}_j}{\partial w_{ji}}\right)\left(\frac{\partial \sigma(\mathbf{w}_j \mathbf{x}_j)}{\partial \mathbf{w}_j \mathbf{x}_j}\right)\left(\frac{\partial \frac{1}{2}(y-\hat{y})^2}{\partial \hat{y}}\right)$$

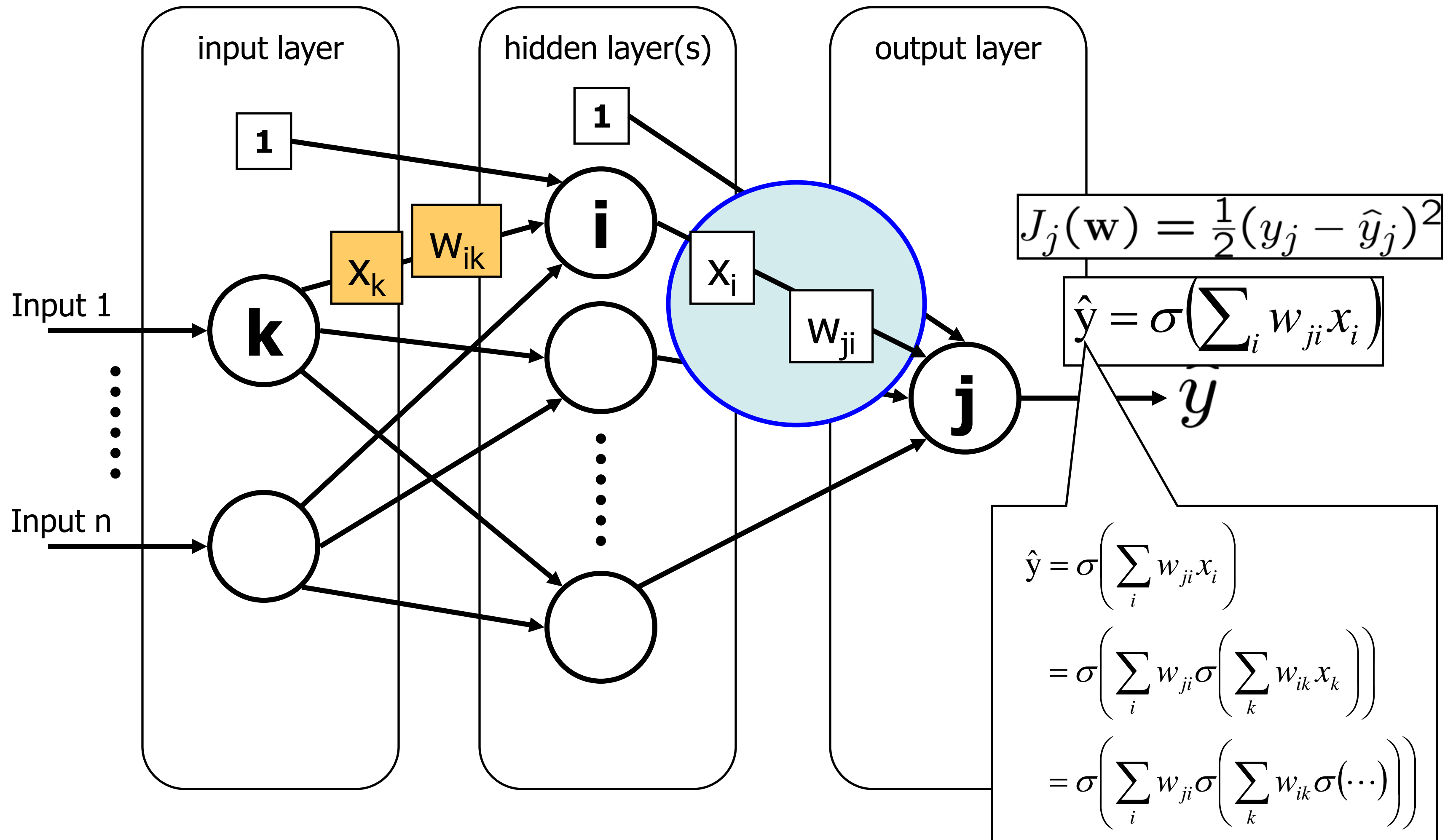
$$= (x_i)\left(\sigma(\mathbf{w}_j \mathbf{x}_j)(1-\sigma(\mathbf{w}_j \mathbf{x}_j))\right)\left(2 \cdot \tfrac{1}{2}(\hat{y}-y)\right)$$

$$= (x_i)(\hat{y})(1-\hat{y})(\hat{y}-y)$$

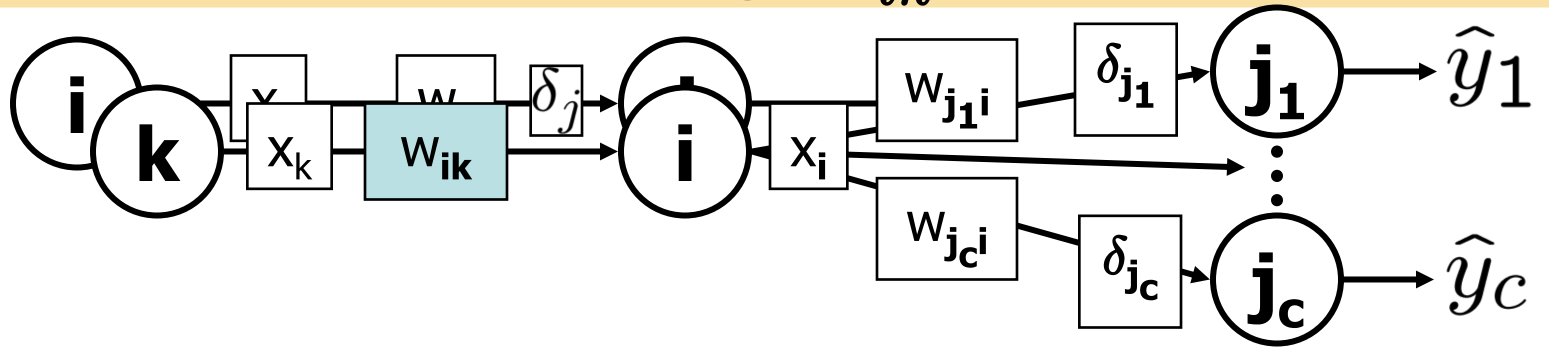
Define 'sensitivity':

$$\delta_j = (\hat{y}-y)\hat{y}(1-\hat{y})$$

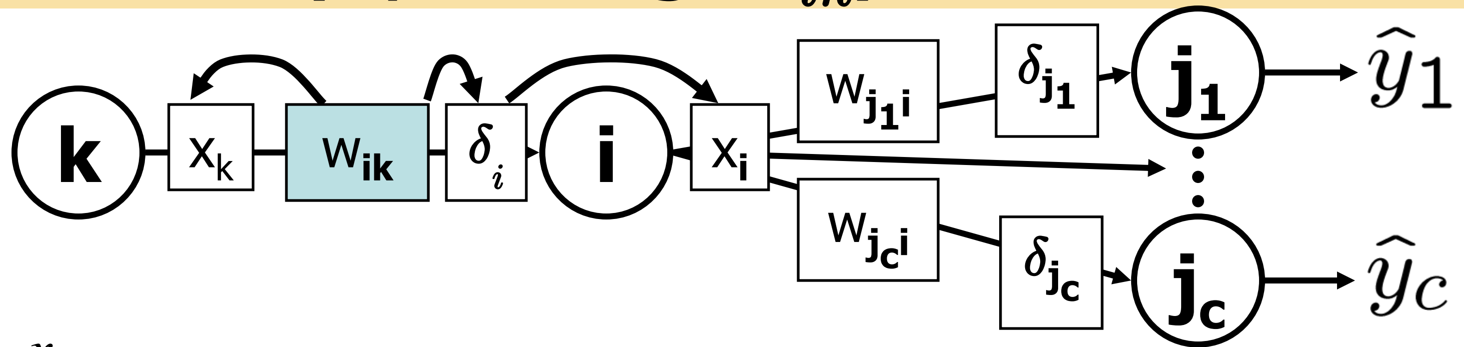
Some Terminology and Notation



Derivation: Non-Output Units (updating w_{ik})

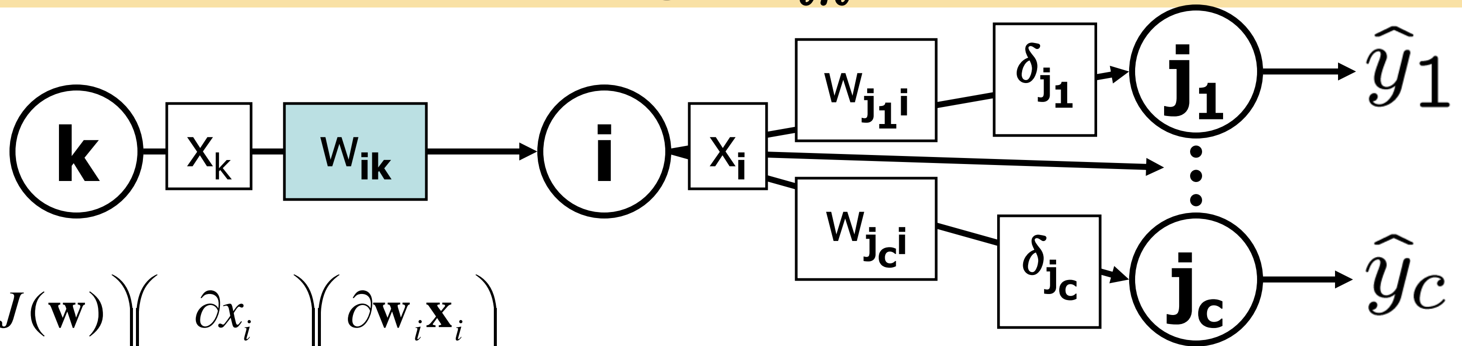


Derivation: Non-Output Units (updating w_{ik})



$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \delta_i \cdot x_k$$

Derivation: Non-Output Units (updating w_{ik})



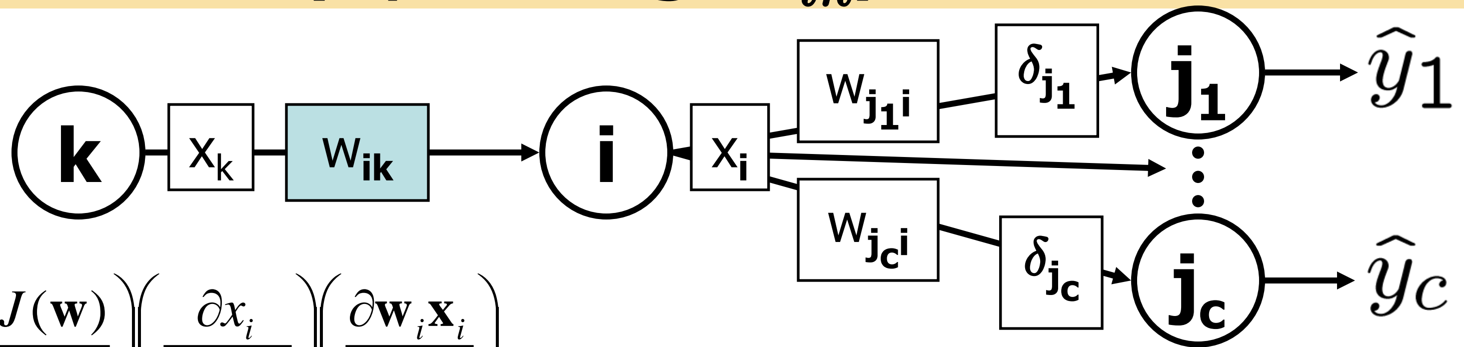
$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \left(\frac{\partial J(\mathbf{w})}{\partial x_i}\right)\left(\frac{\partial x_i}{\partial \mathbf{w}_i \mathbf{x}_i}\right)\left(\frac{\partial \mathbf{w}_i \mathbf{x}_i}{\partial w_{ik}}\right)$$

Remember: $\mathbf{w_i} \cdot \mathbf{x_i} = \sum_k w_{ik} \cdot x_k$

$x_i = \sigma(\mathbf{w_i x_i})$, so this equals $(x_i)(1-x_i)$

This is just x_k

Derivation: Non-Output Units (updating w_{ik})



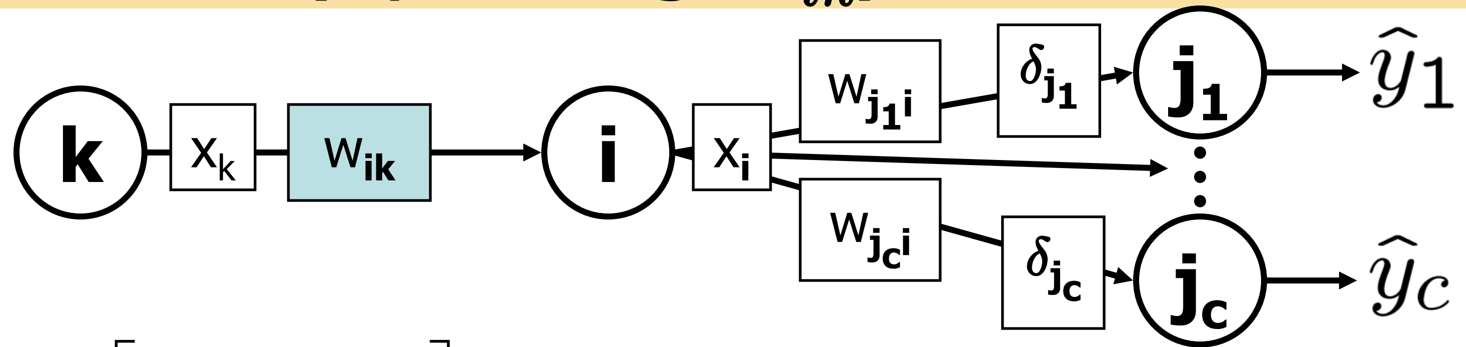
$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \left(\frac{\partial J(\mathbf{w})}{\partial x_i}\right)\left(\frac{\partial x_i}{\partial \mathbf{w}_i \mathbf{x}_i}\right)\left(\frac{\partial \mathbf{w}_i \mathbf{x}_i}{\partial w_{ik}}\right)$$

$$= \left(\frac{\partial J(\mathbf{w})}{\partial x_i}\right)(x_i)(1 - x_i)\cdot(x_k)$$

Remember: $\mathbf{w_i} \cdot \mathbf{x_i} = \sum_k w_{ik} \cdot x_k$

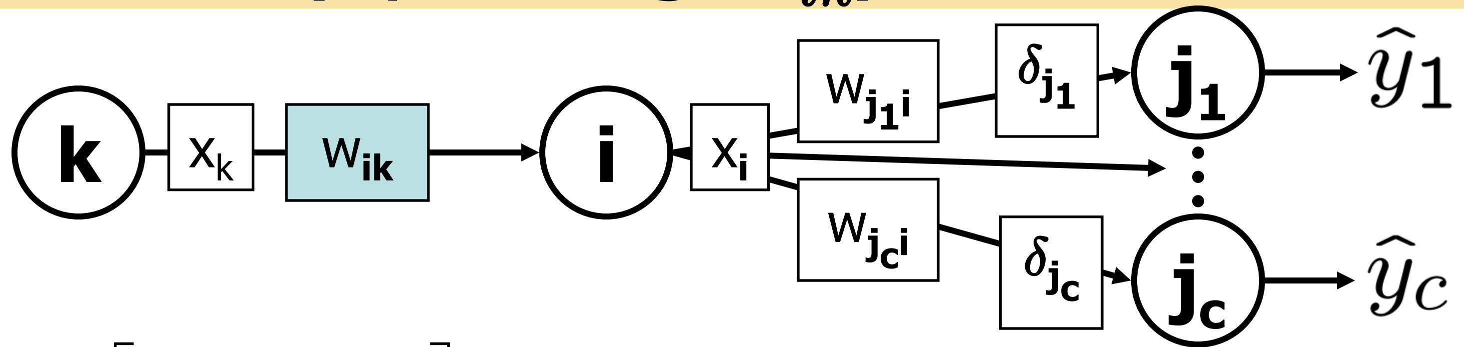
Let's focus in on this.

Derivation: Non-Output Units (updating w_{ik})



$$\frac{\partial J(\mathbf{w})}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\frac{1}{2}\sum_{k=1}^{c}\left(y-\hat{y}\right)^2\right]$$

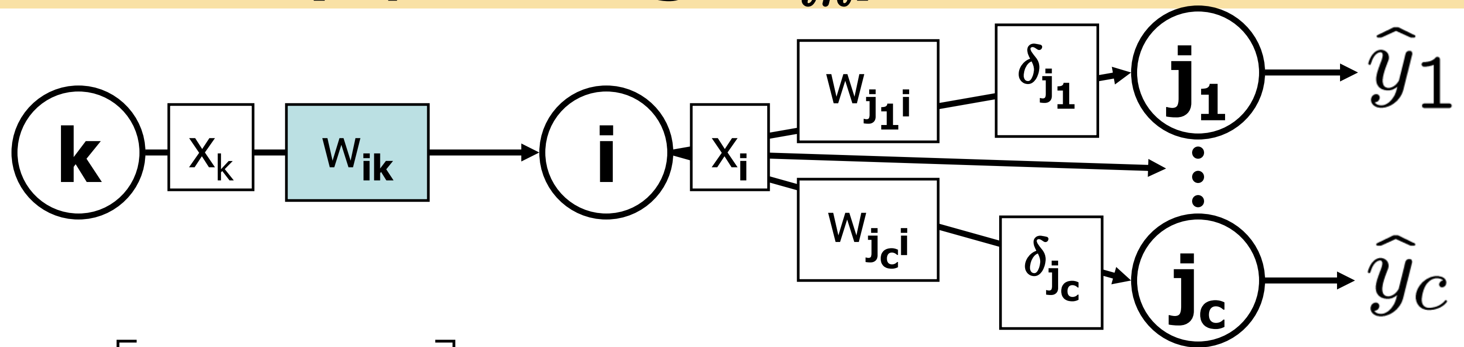
Derivation: Non-Output Units (updating w_{ik})



$$\frac{\partial J(\mathbf{w})}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\frac{1}{2}\sum_{k=1}^{c}\left(y-\hat{y}\right)^2\right]$$

$$= \sum_{k=1}^{c}\left[\left(\hat{y}_k - y_k\right)\cdot\left(\frac{\partial \hat{y}_k}{\partial x_i}\right)\right]$$

Derivation: Non-Output Units (updating w_{ik})



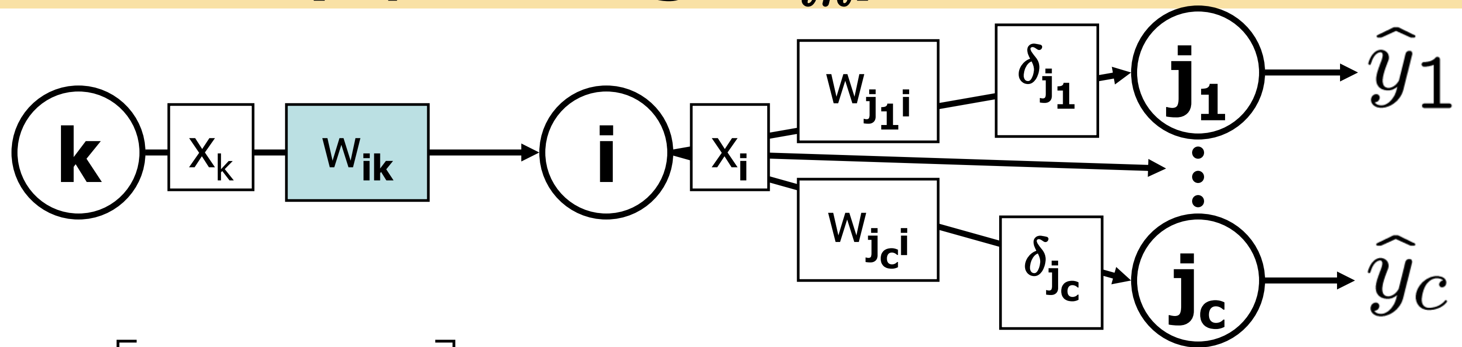
$$\frac{\partial J(\mathbf{w})}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\frac{1}{2}\sum_{k=1}^{c}(y-\hat{y})^2\right]$$

$$= \sum_{k=1}^{c}\left[(\hat{y}_k - y_k)\cdot\left(\frac{\partial \hat{y}_k}{\partial x_i}\right)\right]$$

$$= \sum_{k=1}^{c}\left[(\hat{y}_k - y_k)\cdot\left(\frac{\partial \hat{y}_k}{\partial \mathbf{w}_{j_k}\mathbf{x}_{j_k}}\right)\cdot\left(\frac{\partial \mathbf{w}_{j_k}\mathbf{x}_{j_k}}{\partial x_i}\right)\right]$$

We solved this before: δ_{j_k}

Derivation: Non-Output Units (updating w_{ik})



$$\frac{\partial J(\mathbf{w})}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\frac{1}{2}\sum_{k=1}^{c}(y-\hat{y})^2\right]$$

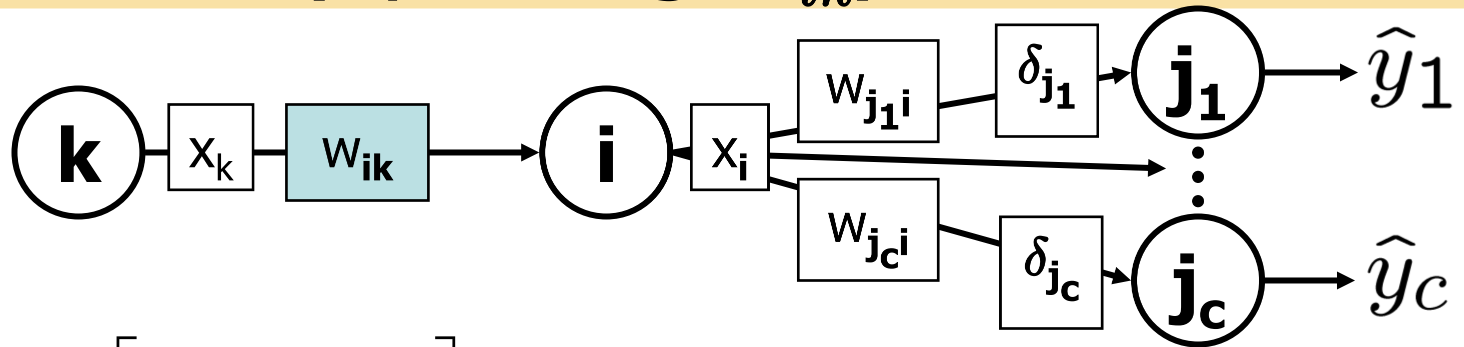
$$= \sum_{k=1}^{c}\left[(\hat{y}_k - y_k)\cdot\left(\frac{\partial \hat{y}_k}{\partial x_i}\right)\right]$$

$$= \sum_{k=1}^{c}\left[(\hat{y}_k - y_k)\cdot\left(\frac{\partial \hat{y}_k}{\partial \mathbf{w}_{j_k}\mathbf{x}_{j_k}}\right)\cdot\left(\frac{\partial \mathbf{w}_{j_k}\mathbf{x}_{j_k}}{\partial x_i}\right)\right]$$

We solved this before: δ_{j_k}

This is just $w_{j_k i}$

Derivation: Non-Output Units (updating w_{ik})



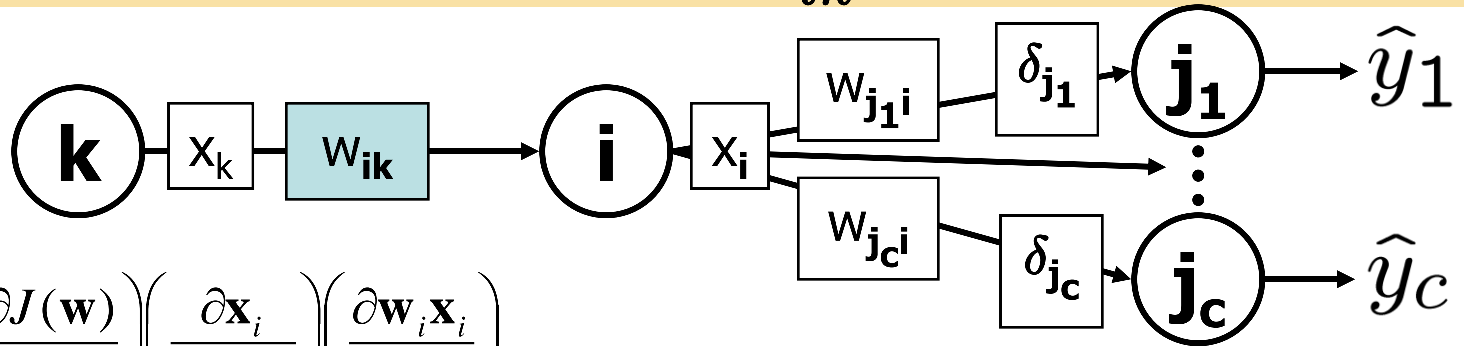
$$\frac{\partial J(\mathbf{w})}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\frac{1}{2}\sum_{k=1}^{c}(y-\hat{y})^2\right]$$

$$= \sum_{k=1}^{c}\left[(\hat{y}_k - y_k)\cdot\left(\frac{\partial \hat{y}_k}{\partial x_i}\right)\right]$$

$$= \sum_{k=1}^{c}\left[(\hat{y}_k - y_k)\cdot\left(\frac{\partial \hat{y}_k}{\partial \mathbf{w}_{j_k}\mathbf{x}_{j_k}}\right)\cdot\left(\frac{\partial \mathbf{w}_{j_k}\mathbf{x}_{j_k}}{\partial x_i}\right)\right]$$

$$= \sum_{k=1}^{c}\left[\delta_{j_k}\cdot w_{j_k i}\right]$$

Derivation: Non-Output Units (updating w_{ik})



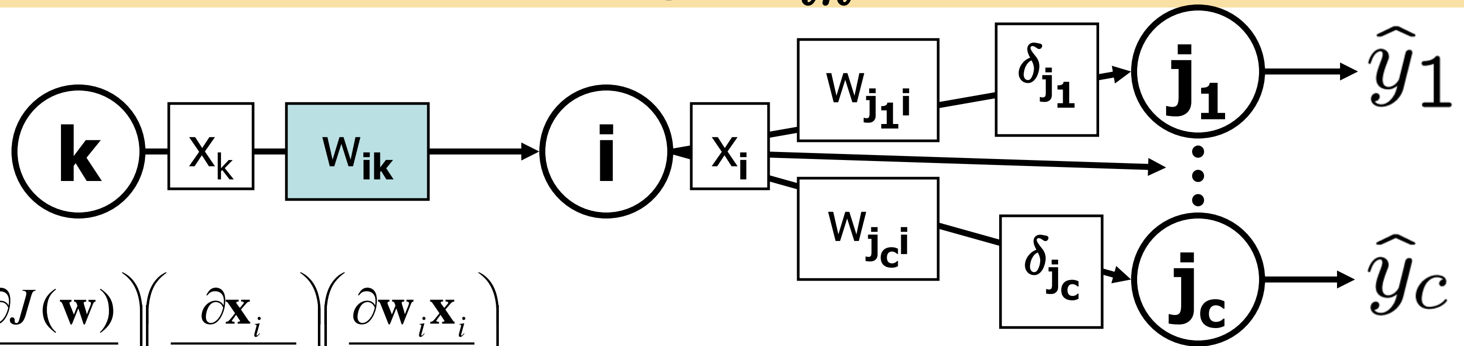
$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{x}_i}\right)\left(\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}_i \mathbf{x}_i}\right)\left(\frac{\partial \mathbf{w}_i \mathbf{x}_i}{\partial w_{ik}}\right)$$

$$= \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{x}_i}\right)(x_i)(1 - x_i)\cdot(x_k)$$

Now plug back in…

$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \left(\sum_{k=1}^{c} \delta_{j_k} \cdot w_{j_k i}\right)(x_i)(1 - x_i)\cdot(x_k)$$

Derivation: Non-Output Units (updating w_{ik})



$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{x}_i}\right)\left(\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}_i \mathbf{x}_i}\right)\left(\frac{\partial \mathbf{w}_i \mathbf{x}_i}{\partial w_{ik}}\right)$$

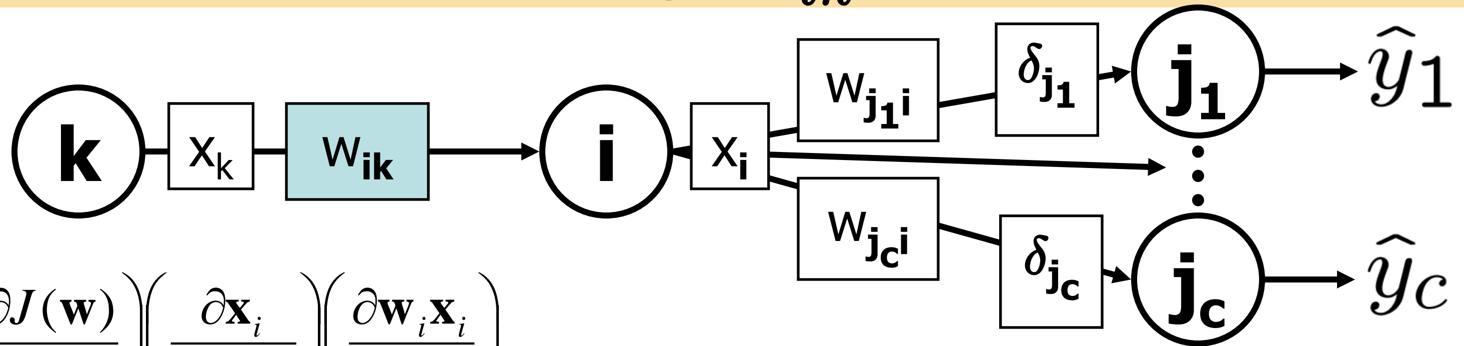
$$= \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{x}_i}\right)(x_i)(1 - x_i) \cdot (x_k)$$

Now plug back in…

$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \left(\sum_{k=1}^{c} \delta_{j_k} \cdot w_{j_k i}\right)(x_i)(1 - x_i) \cdot (x_k)$$

Define $\delta_{\mathrm{i}} = \left(\sum_{k=1}^{c} \delta_{j_k} \cdot w_{j_k i}\right)(x_i)(1 - x_i)$

Derivation: Non-Output Units (updating w_{ik})



$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{x}_i}\right)\left(\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}_i \mathbf{x}_i}\right)\left(\frac{\partial \mathbf{w}_i \mathbf{x}_i}{\partial w_{ik}}\right)$$

$$= \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{x}_i}\right)(x_i)(1 - x_i)\cdot(x_k)$$

Now plug back in…

$$\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \left(\sum_{k=1}^{c} \delta_{j_k} \cdot w_{j_k i}\right)(x_i)(1 - x_i)\cdot(x_k)$$

Define $\delta_{\mathrm{i}} = \left(\sum_{k=1}^{c} \delta_{j_k} \cdot w_{j_k i}\right)(x_i)(1 - x_i)$

and rewrite as $\frac{\partial J(\mathbf{w})}{\partial w_{ik}} = \delta_{\mathrm{i}} \cdot x_k$

The Backpropagation Algorithm

- Just gradient descent over all weights in the network.
- Pick a training example and:
  1. Forward Pass.
     1. Compute x_i and $\hat{y}_j$ for hidden units i and output units j
     2. Compute Errors. Compute $\varepsilon_j = (\hat{y}_j - y_j)$ for each output unit j
  2. Backward pass:
     1. Compute Output Sensitivities. $\delta_j = (\hat{y}_j - y_j)\, \hat{y}_j(1 - \hat{y}_j)$
     2. Compute Hidden Sensitivities. $\delta_i = x_i(1 - x_i) \sum_j w_{ji}\, \delta_j$
     3. Compute Gradients:

$$\frac{\partial J}{\partial w_{ik}} = \delta_i x_k$$ for input-to-hidden weights.

$$\frac{\partial J}{\partial w_{ji}} = \delta_j x_i$$ for hidden-to-output weights.

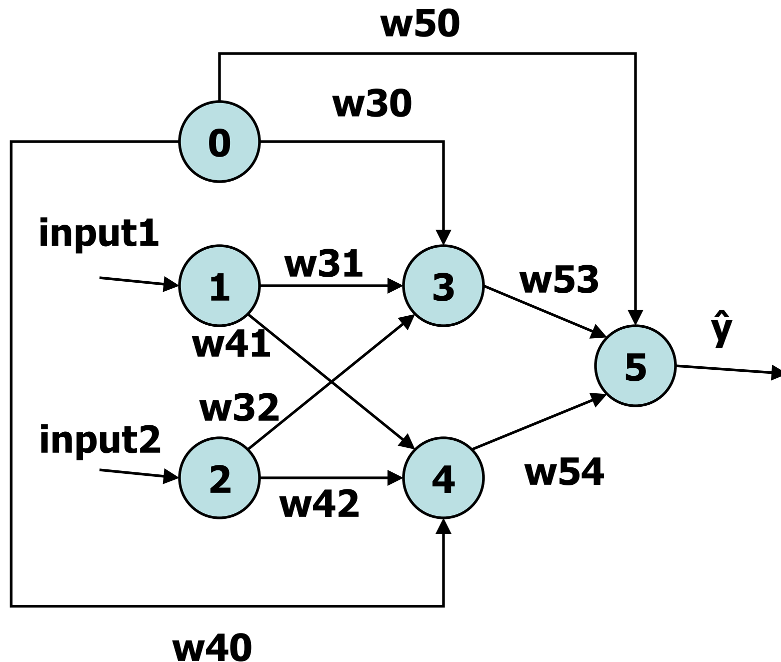
- Take Gradient Step.

$$w_{ji} \leftarrow w_{ji} - \eta \delta_j x_i$$
$$w_{ik} \leftarrow w_{ik} - \eta \delta_i x_k$$

The Backpropagation Algorithm (2)

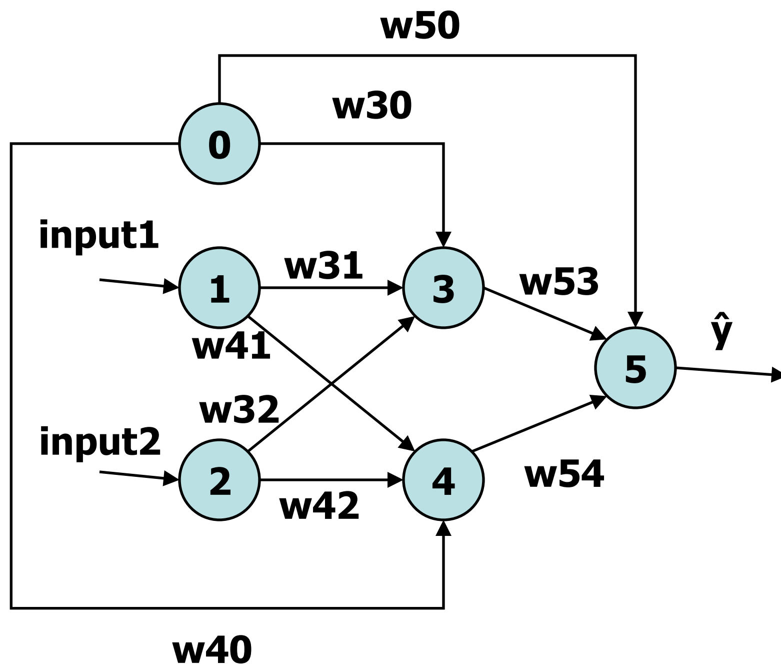
- The previous version corresponds to incremental (stochastic) gradient descent
- An analogous batch version can be used as well:
  - Loop through the training data, accumulating weight changes
  - Update weights
- One pass through the data set is called **epoch**
- Algorithm can be easily generalized to predict probabilities, instead of minimizing sum-squared error

Example: A network representing the XOR function



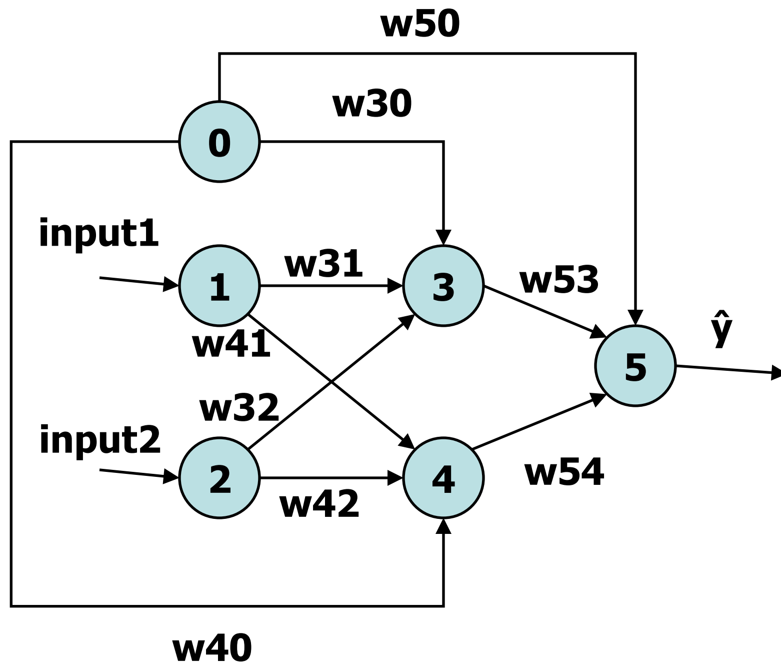
w30=0.1	w31=0.15
w40=-0.1	w32=-0.10
w50=0.05	w41=-0.12
w53=0.18	w42=-0.02
w54=-0.12	
η=0.1	

Example: A network representing the XOR function



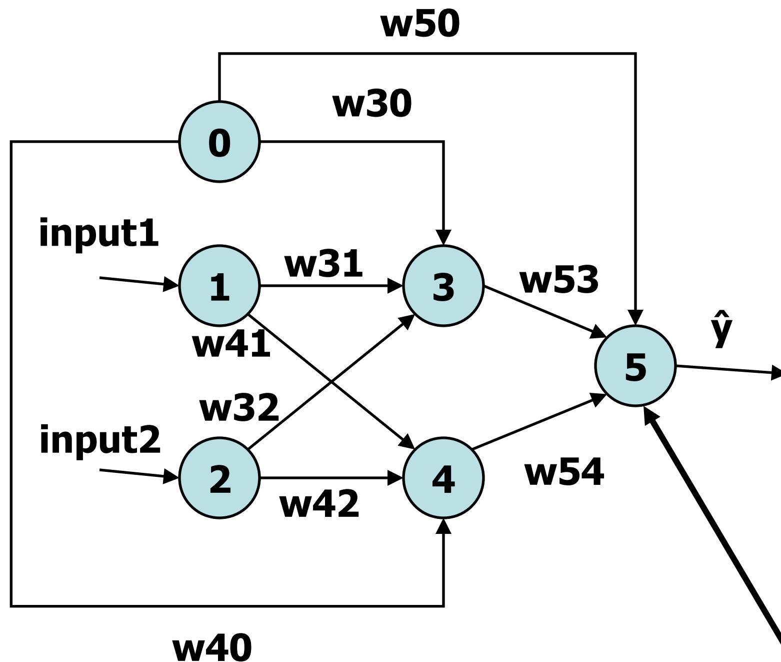
w30=0.1	w31=0.15
w40=-0.1	w32=-0.10
w50=0.05	w41=-0.12
w53=0.18	w42=-0.02
w54=-0.12	
η=0.1	
input1=1	input2=1

Example: A network representing the XOR function



w30=0.1	w31=0.15	
w40=-0.1	w32=-0.10	
w50=0.05	w41=-0.12	
w53=0.18	w42=-0.02	
w54=-0.12		
η=0.1		
input1=1	input2=1	
o3=0.537	o4=0.450	
$\hat{y}$=0.715	y=0	ε=0.715

Example: A network representing the XOR function

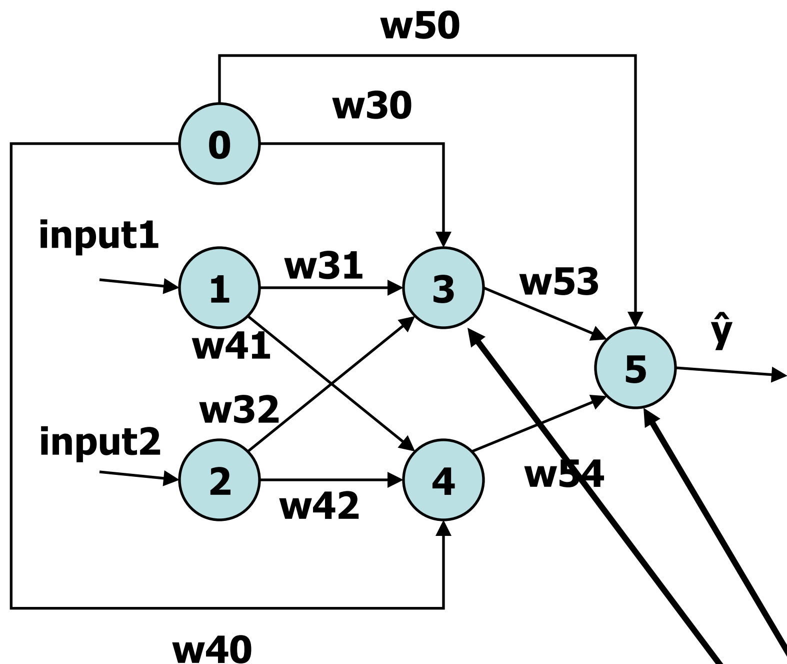


$$\delta_j = (\hat{y} - y)\hat{y}(1 - \hat{y})$$

w30=0.1	w31=0.15	
w40=-0.1	w32=-0.10	
w50=0.05	w41=-0.12	
w53=0.18	w42=-0.02	
w54=-0.12		
η=0.1		
input1=1	input2=1	
o3=0.537	o4=0.450	
ŷ=0.715	y=0	ε=0.715

δ_5=(0.715-0)(0.715)(1-0.715)=0.146

Example: A network representing the XOR function



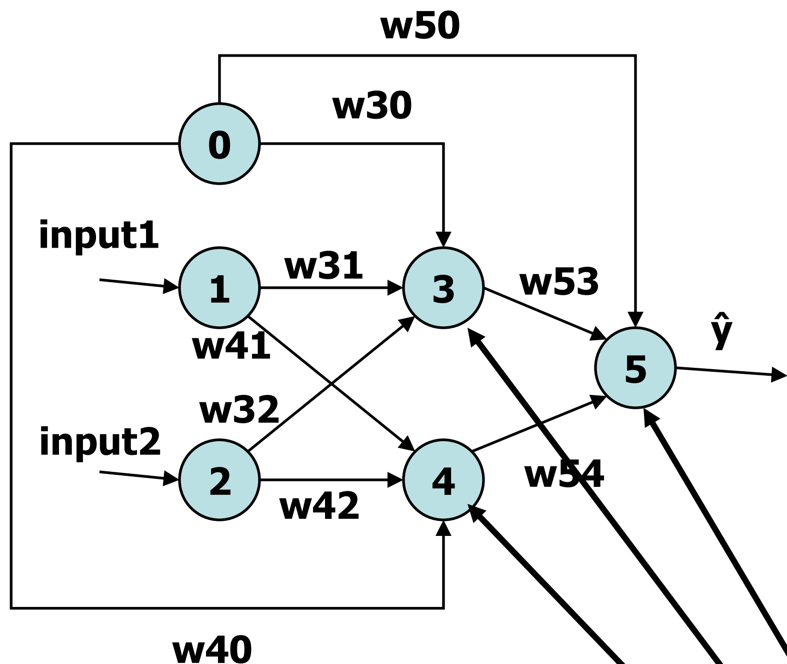
$$\delta_i = \left(\sum_{k=1}^{c} \delta_{j_k} \cdot w_{j_k i}\right) \cdot (o_i)(1 - o_i)$$

w30=0.1	w31=0.15	
w40=-0.1	w32=-0.10	
w50=0.05	w41=-0.12	
w53=0.18	w42=-0.02	
w54=-0.12		
η=0.1		
input1=1	input2=1	
o3=0.537	o4=0.450	
$\hat{y}$=0.715	y=0	ε=0.715

δ_5=(0.715-0)(0.715)(1-0.715)=0.146

δ_3=(0.537)(1-0.537)*0.18*0.146=0.006

Example: A network representing the XOR function



w30=0.1	w31=0.15
w40=-0.1	w32=-0.10
w50=0.05	w41=-0.12
w53=0.18	w42=-0.02
w54=-0.12	

η=0.1

input1=1 input2=1

o3=0.537 o4=0.450

$\hat{y}$=0.715 y=0 ε=0.715

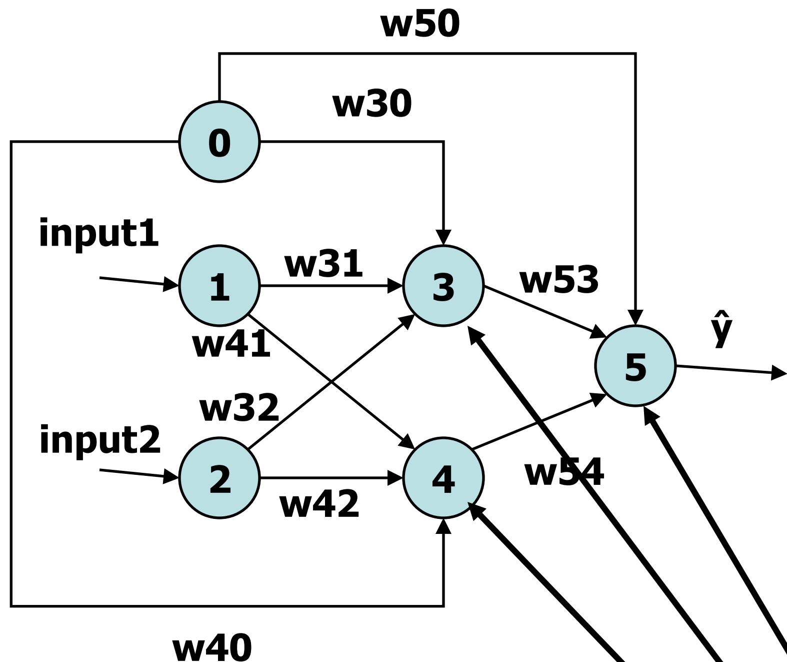
δ_5=(0.715-0)(0.715)(1-0.715)=0.146

δ_3=(0.537)(1-0.537)*0.18*0.146=0.006

δ_4=(0.450)(1-0.450)*-0.12*0.146=-0.004

$$\delta_i = \left(\sum_{k=1}^{c} \delta_{j_k} \cdot w_{j_k i}\right) \cdot (o_i)\,(1 - o_i)$$

Example: A network representing the XOR function



w30=0.1	w31=0.15	
w40=-0.1	w32=-0.10	
w50=0.05	w41=-0.12	
w53=0.18	w42=-0.02	
w54=-0.12		
η=0.1		
input1=1	input2=1	
o3=0.537	o4=0.450	
$\hat{y}$=0.715	y=0	ε=0.715

δ_5=(0.715-0)(0.715)(1-0.715)=0.146

δ_3=(0.537)(1-0.537)*0.18*0.146=0.006

δ_4=(0.450)(1-0.450)*-0.12*0.146=-0.004

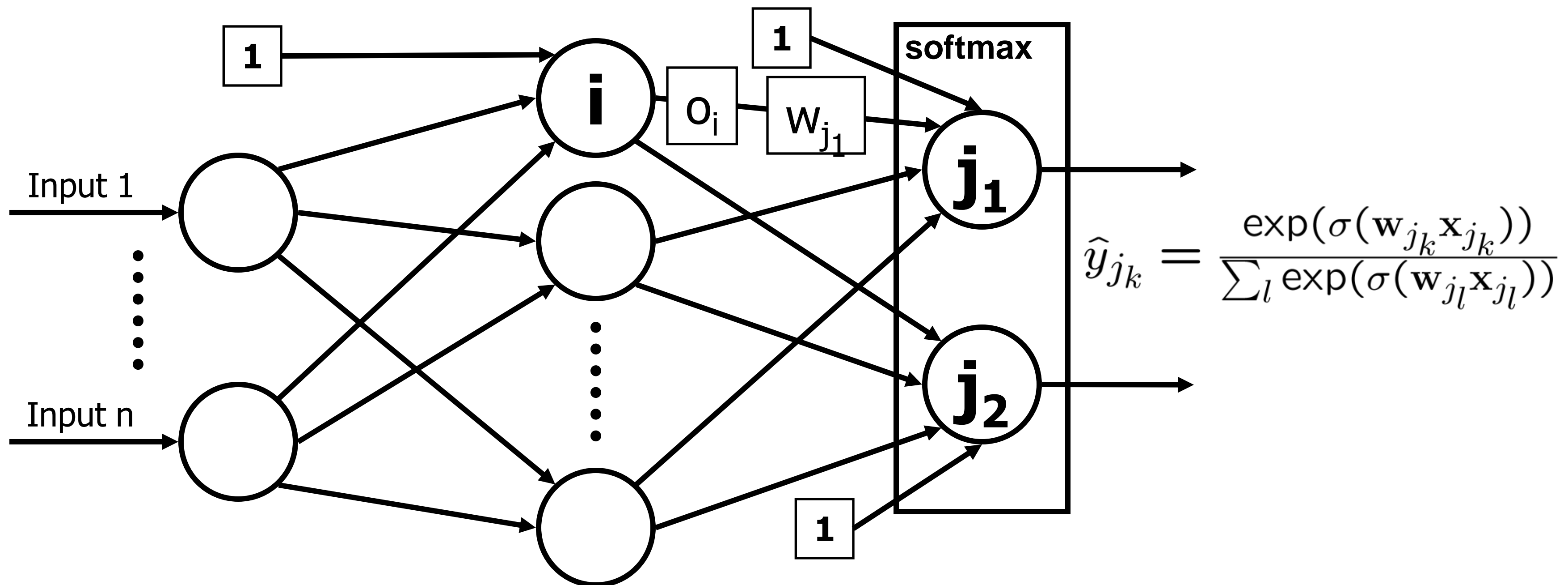
Take gradient step η

$\Delta w_{53}=\eta*\delta_5*o3=0.008 \rightarrow w_{53}=0.172$

$\Delta w_{54}=\eta*\delta_5*o4=0.006 \rightarrow w_{54}=-0.126$

…

Alternative Error Function: Softmax Output Layer



$$\hat{y}_{j_k} = \frac{\exp(\sigma(\mathbf{w}_{j_k}\mathbf{x}_{j_k}))}{\sum_l \exp(\sigma(\mathbf{w}_{j_l}\mathbf{x}_{j_l}))}$$

- The objective function is the negative log likelihood:

$$J(W) = \sum_i \sum_k -I[y_i = k] \log \hat{y}_{j_k}$$

- where I[expr] is the indicator function:
  - I[expr] is 1 if expr is true and 0 otherwise

Proper Initialization

- Start in the “linear” regions
  - keep all weights near zero, so that all sigmoid units are in their linear regions. This makes the whole net the equivalent of one linear threshold unit—a relatively simple function.
- Break symmetry.
  - Ensure that each hidden unit has different input weights so that the hidden units move in different directions.
- Set each weight to a random number in the range

$$[-1, +1] \times \frac{1}{\sqrt{\text{fan-in}}}.$$

where the “fan-in” of weight $w_{v,u}$ is the number of inputs to unit v.

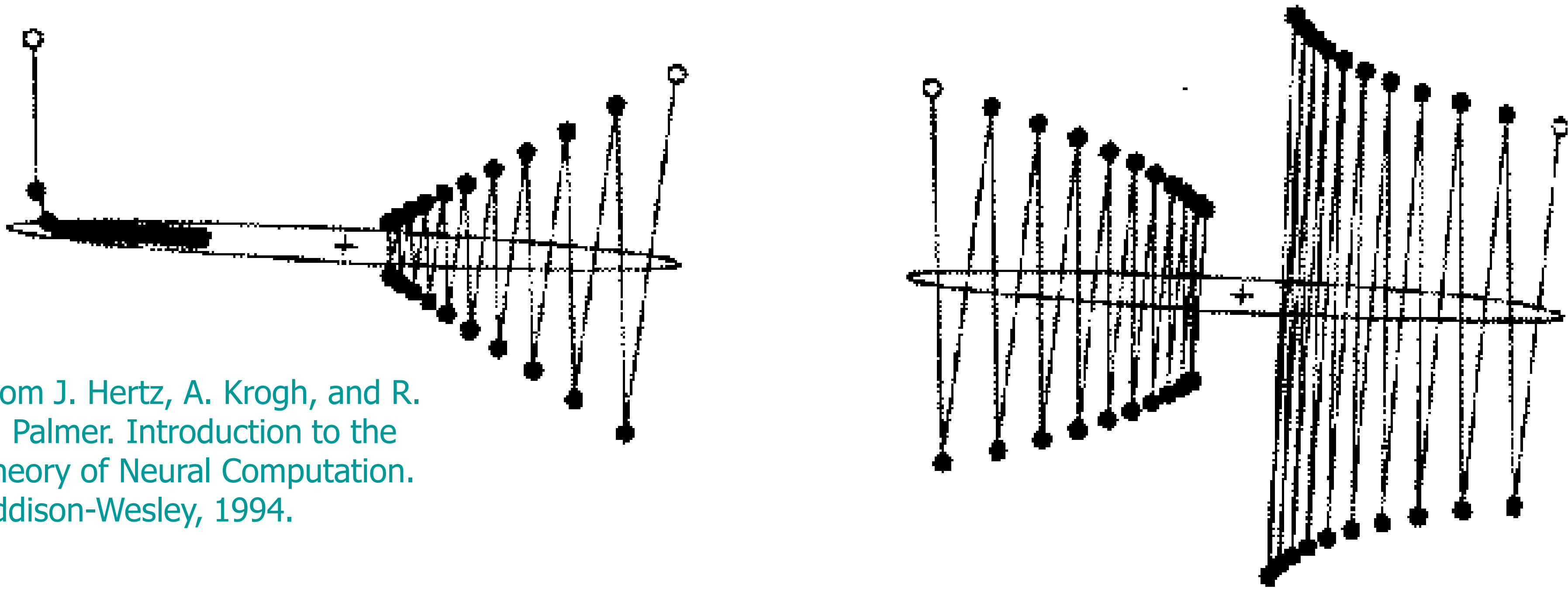
Convergence of backpropagation

- Backpropagation performs gradient descent over *all* the parameters in the network
- Hence, if the learning rate is appropriate, then the algorithm is guaranteed to converge to a *local minimum* of the cost function
  - NOT the global minimum
  - Can be much worse than global minimum
  - There can be MANY local minima (Auer et al, 1997)
- Partial solution: **restarting** = train multiple networks with different initial weights
- In practice, quite often the solution found is very good.

Choosing the learning rate

- This is a subtle art.
  - Too small: can take days instead of minutes to converge
  - Too large: diverges (MSE gets larger and larger while the weights increase and usually oscillate)
  - The learning rate also influences the ability to escape local optima
- Very often, different learning rates are used for units in different layers
- It can be shown that ***each unit*** has its own optimal learning rate
- The "just right" value is hard to find.

Learning-rate problems



From J. Hertz, A. Krogh, and R. G. Palmer. Introduction to the Theory of Neural Computation. Addison-Wesley, 1994.

FIGURE 5.10 Gradient descent on a simple quadratic surface (the left and right parts are copies of the same surface). Four trajectories are shown, each for 20 steps from the open circle. The minimum is at the + and the ellipse shows a constant error contour. The only significant difference between the trajectories is the value of η, which was 0.02, 0.0476, 0.049, and 0.0505 from left to right.

Adding momentum

On the t-th training sample, instead of the update:

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

we do:

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

momentum parameter

The second term is called momentum

Advantages:

- Easy to pass small local minima
- Keeps the weights moving in areas where the erro is flat
- Increases the speed where the gradient stays unchanged

Disadvantages:

- With too much momentum, it can get out of a global maximum!
- One more parameter to tune, and more chances of divergence

Typical values of μ are in the range [0.7, 0.95]

Momentum illustration

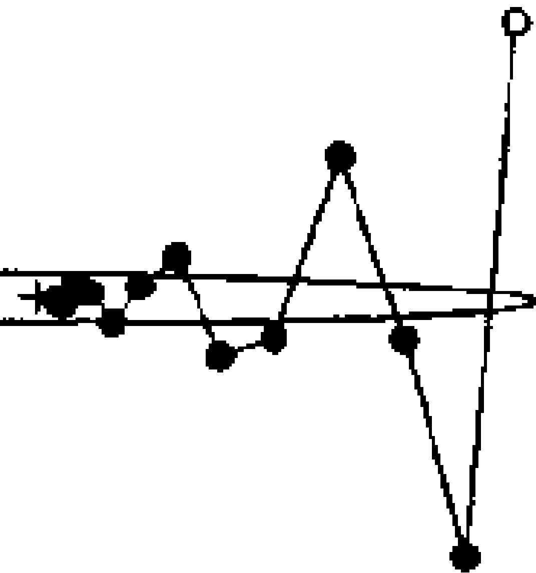


FIGURE 6.3 Gradient descent on the simple quadratic surface of Fig. 5.10. Both trajectories are for 12 steps with $\eta = 0.0476$, the best value in the absence of momentum. On the left there is no momentum ($\alpha = 0$), while $\alpha = 0.5$ on the right.

Adjusting the learning rate: Delta-bar-delta

- Heuristic method, works best in batch mode (though there have been attempts to make it incremental)
- The intuition:
  - If the gradient direction is stable, the learning rate should be increased
  - If the gradient flips to the opposite direction the learning rate should be decreased
- A running average of the gradient and a separate learning rate is kept for each weight
- If the new gradient and the old average have the same sign, increase the learning rate by a constant amount
- If they have opposite sign, decay the learning rate exponentially

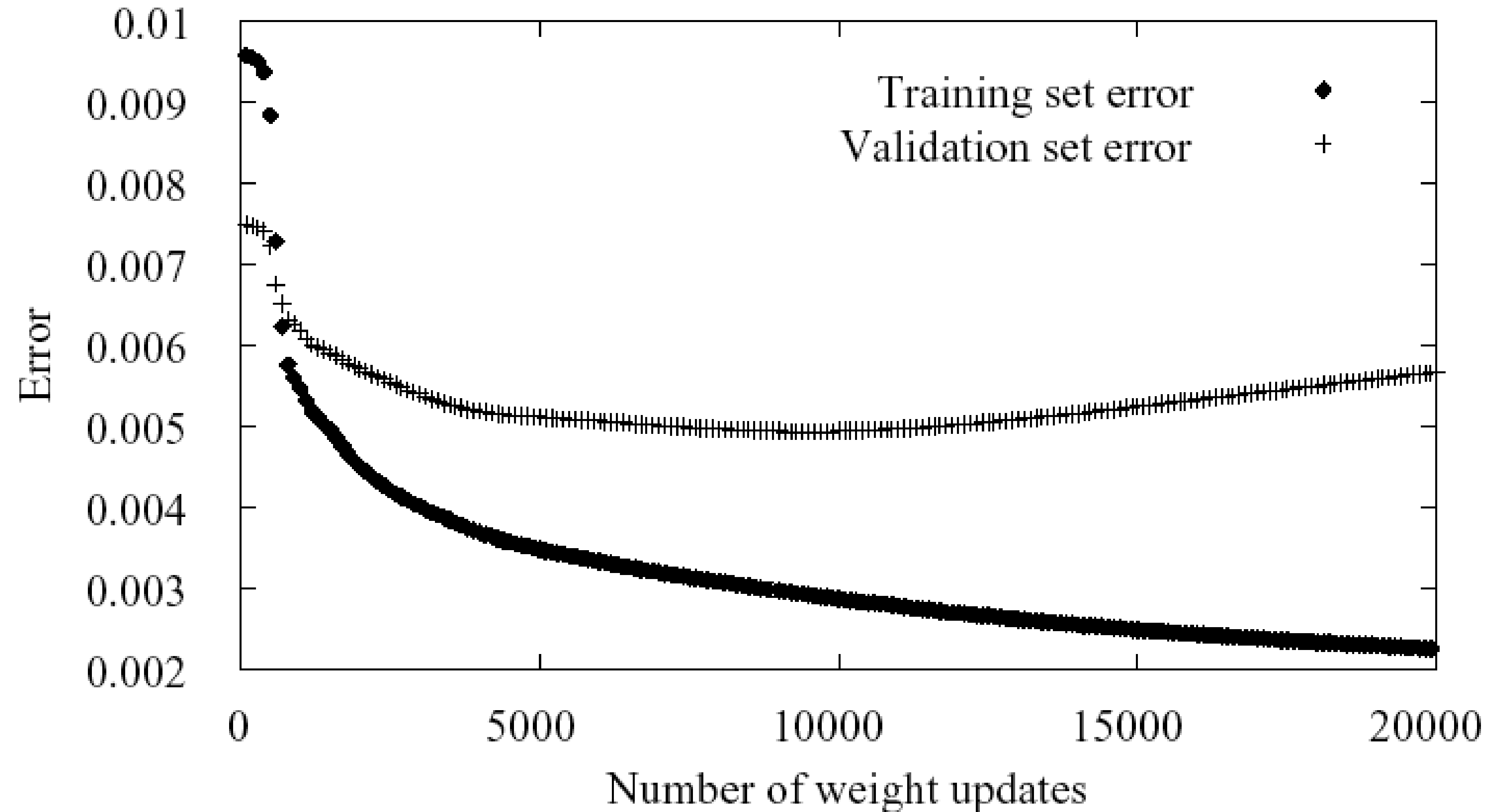
More application-specific tricks

- If there is too little data, it can be perturbed by random noise; this helps escape local minima and gives more robust results
- In image classification and pattern recognition tasks, extra data can be generated, e.g., by applying transformations that make sense
- **Weight sharing** can be used to indicate parameters that should have the same value based on prior knowledge
- In this case, each update is computed separately using backprop, then the tied parameters are updated with an average

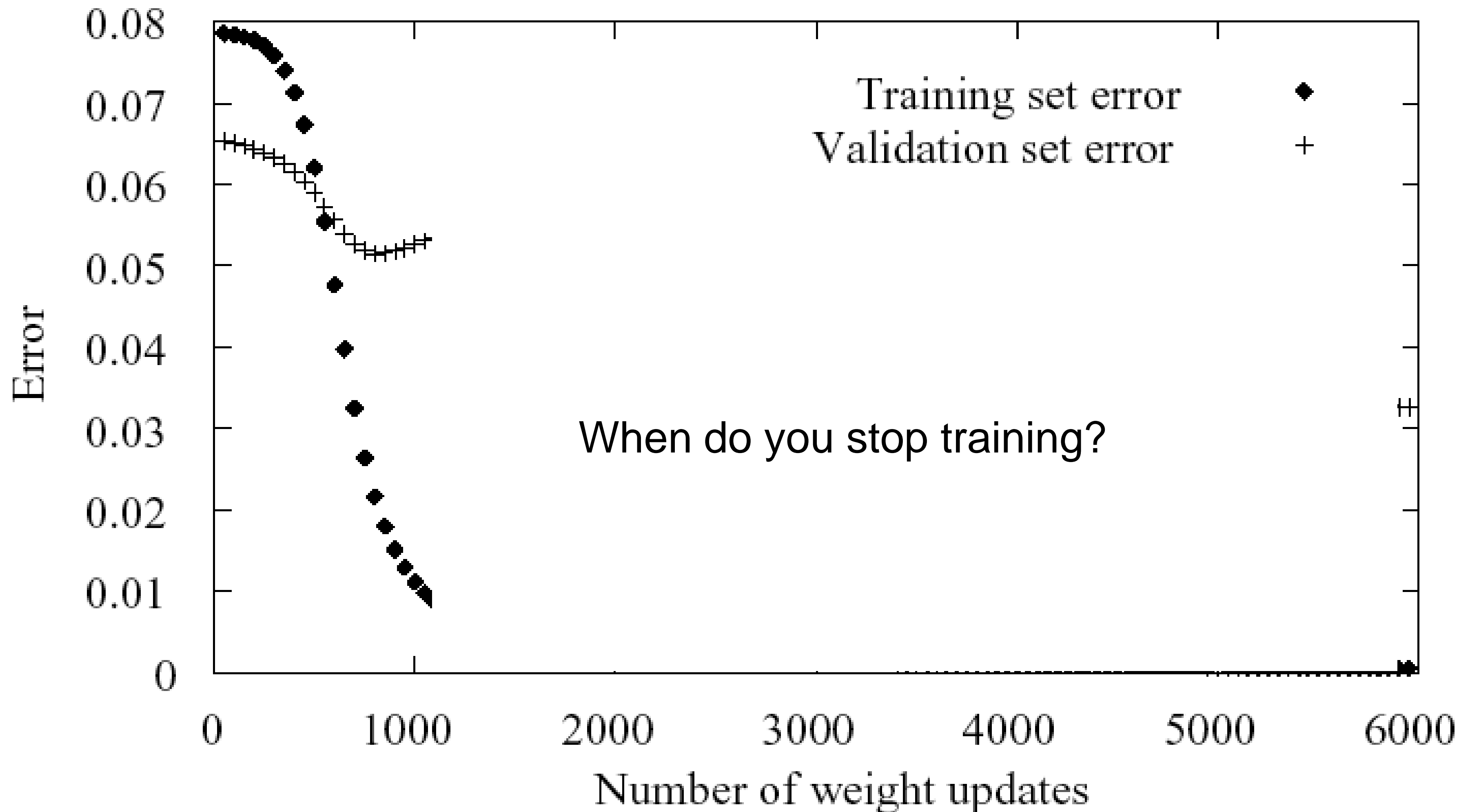
How large should the network be?

- Overfitting occurs if there are too manyu parameters compared to the amount of data available
- Choosing the number of hidden units:
  - Too few hidden units do not allow the concept to be learned
  - Too many lead to slow learning and overfitting
  - If the n inputs are binary, $log\ n$ is a good heuristic choice
- Choosing the number of layers
  - Always start with one hidden layer
  - Never go beyond 2 hidden layers, unless the task structure suggests something different

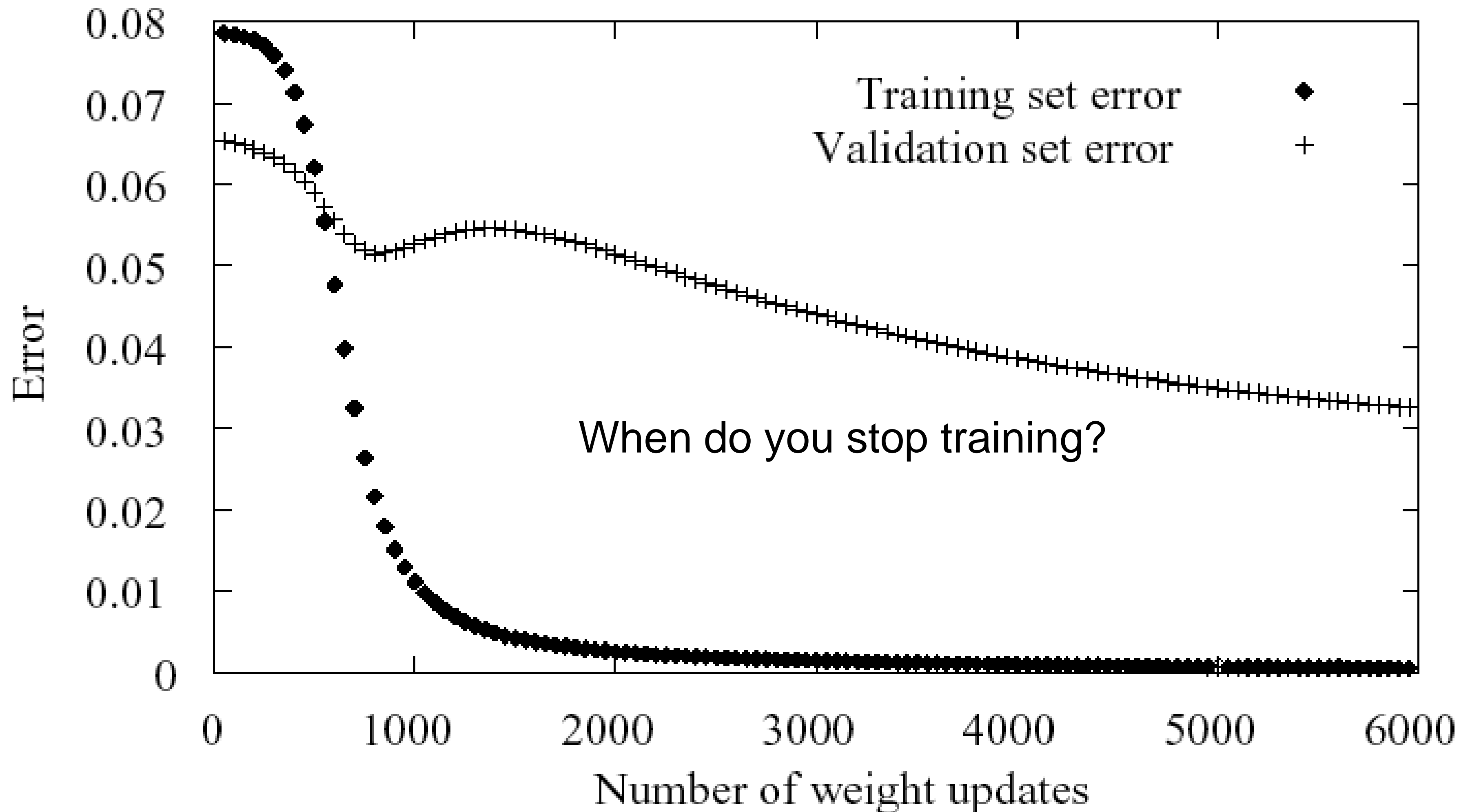
Overfitting in feed-forward networks



Overfitting in feed-forward networks



Overfitting in feed-forward networks



Finding the right network structure

- *Destructive methods* start with a large network and then remove (prune) connections
- *Constructive methods* start with a small network (e.g. 1 hidden unit) and add units as required to reduce error

Destructive methods

- Simple solution: consider removing each weight in turn (by setting it to 9=0), and examine the effect on the error
- Weight decay: give each weight a chance to go to 0, unless it is needed to decrease error:

$$\Delta w_j = -\alpha_j \frac{\partial J}{\partial w_j} - \lambda w_j$$

  where λ is a decay rate
- Optimal brain damage:
  - Train the network to a local optimum
  - Approximate the saliency of each link or unit (i.e., its impact on the performance of the network), using the Hessian matrix
  - Greedily prune the element with the lowest saliency

  This loop is repeated until the error starts to deteriorate

Constructive methods

- Dynamic node creation (Ash):
  - Start with just one hidden unit, train using backprop
  - If the error is still high, add another unit in the same layer and repeat

  Only one layer is constructed
- Meiosis networks (Hanson):
  - Start with just one hidden unit, train using backprop
  - Compute the variance of each weight during training
  - If a unit has one or more weights of high variance, it is split into two units, and the weights are perturbed

  The intuition is that the new units will specialize to different functions.

When to consider neural networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued, or a *vector of values*
- Possibly noisy data
- Training time is unimportant
- Form of target function is unknown
- Human readability of result is unimportant
- The computation of the output based on the input has to be fast

Neural Network Evaluation

Criterion	Perc	Logistic	LDA	Trees	NNbr	Nets
Mixed data	no	no	no	yes	no	no
Missing values	no	no	yes	yes	somewhat	no
Outliers	no	yes	no	yes	yes	yes
Monotone transformations	no	no	no	yes	no	somewhat
Scalability	yes	yes	yes	yes	no	yes
Irrelevant inputs	no	no	no	somewhat	no	no
Linear combinations	yes	yes	yes	no	somewhat	yes
Interpretable	yes	yes	yes	yes	no	no
Accurate	yes	yes	yes	no	no	yes

QUIZ #4

1. Can a sigmoid unit learn a function that an LTU cannot?
2. Why do we use the sigmoid function?
3. Why is it called a “feed-forward” network?
4. Why do we need backpropagation? Why can we not update all weights at once?