Feedforward neural networks: Backpropagation/Gradient Descent

Stefanos Kollias

Machine Learning

Outline

1. Recap from previous lectures
2. Multi-layer perceptron
3. Generalized delta rule (Backpropagation)
4. Practical considerations of MLP
5. Worked examples

Outline

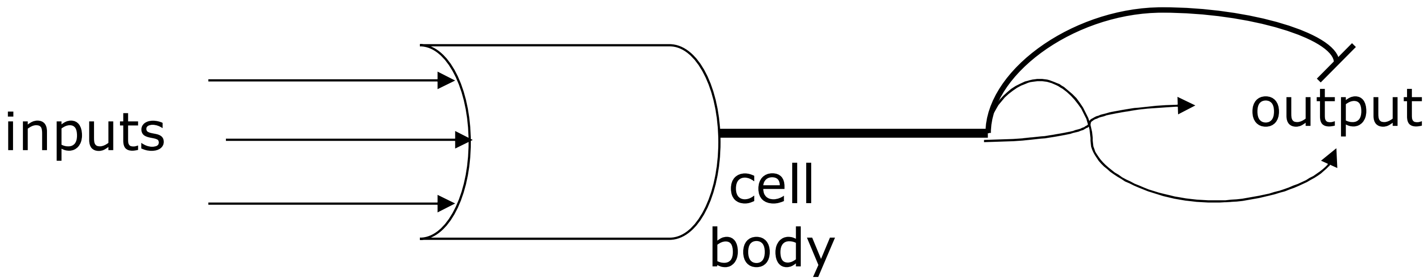
1. Recap from previous lectures
2. Multi-layer perceptron
3. Generalized delta rule (Backpropagation)
4. Practical considerations of MLP
5. Worked examples

Recap from first lecture

- **Unit:** neuron or node, basic information processing structure in neural networks
- **Connection:** a conduit through which information flows between members of a network.
- **Activation:** how actively a neuron sends an action potential (firing rate)
- **Connection weight:** the strength or weakness of a connection
- **Activation function:** a mathematical formula that "squashes" the COMBINED INPUT into the activation value range, usually between 0 and 1
- **How to estimate a unit's output:**
  Step 1: Estimate the combine input
  Step 2: Squash it

Recap: McCulloch-Pitt Neuron

In analogy to a biological neuron, we can think of a virtual neuron that crudely mimics the biological neuron and performs analogous computation.



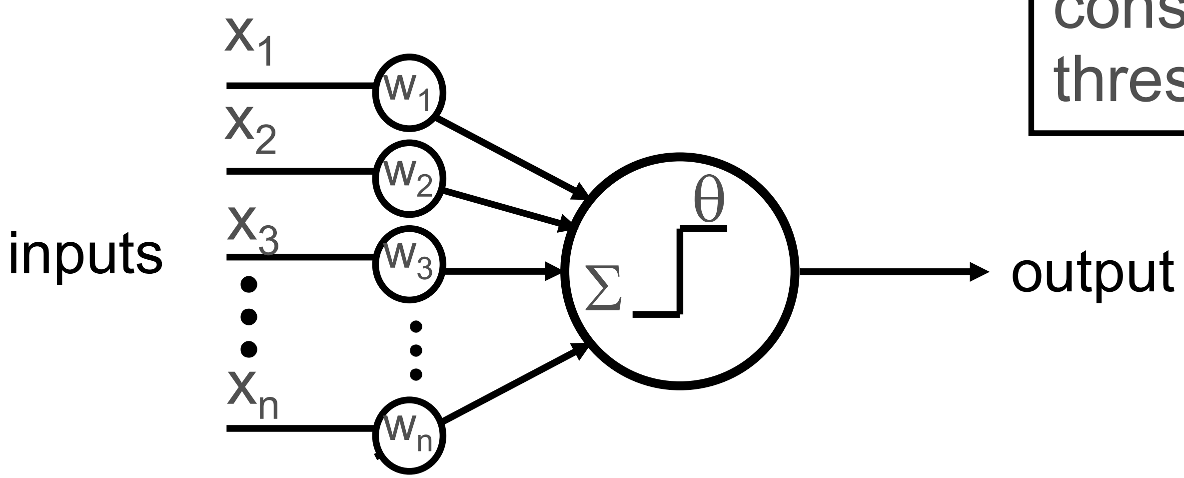
Just like biological neurons, this artificial neuron neuron will have:

- Inputs (like biological dendrites) carry signal to cell body.
- A body (like the soma), sums over inputs to compute output, and
- outputs (like synapses on the axon) transmit the output downstream

Recap: MCP properties

- Inputs x are binary: 0,1
- Each input has an assigned weight w
- Weighted inputs are summed Σ in the cell body.
- Neuron fires if sum exceeds (or equals) activation threshold θ.
- If the neuron fires, the output =1
- Otherwise, the output=0

The "computation" consists of "adders" and a threshold.



$$\left(\text{inputs} * \text{weights} \right)_{\text{over all } i}$$

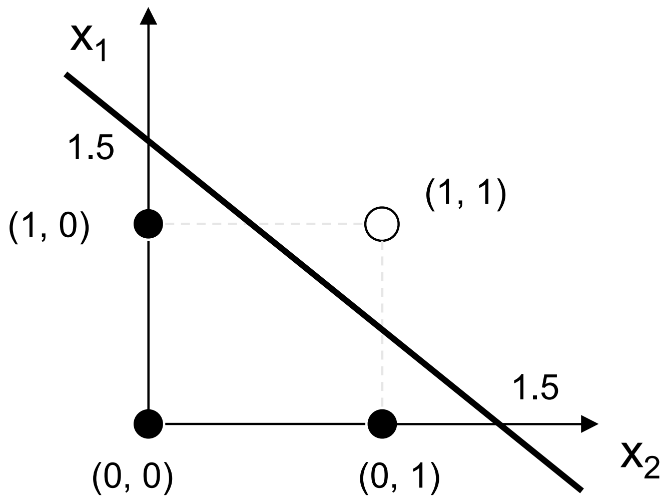
$$\begin{array}{l} 1 \ \text{ if } \ \Sigma \geq \theta \\ 0 \ \text{ if } \ \Sigma < \theta \end{array}$$

Recap: Linear separable problems

We can now plot the decision boundary of AND logic gate

AND
w1=1, w2=1, θ=1.5

AND		
x_1	x_2	out
0	0	0
0	1	0
1	0	0
1	1	1



slope

intercept

$$\sum_i w \cdot x_i = \theta \Leftrightarrow w_1 x_1 + w_2 x_2 = \theta \Leftrightarrow x_2 = -\left(\frac{w_1}{w_2}\right) x_1 + \left(\frac{\theta}{w_2}\right)$$

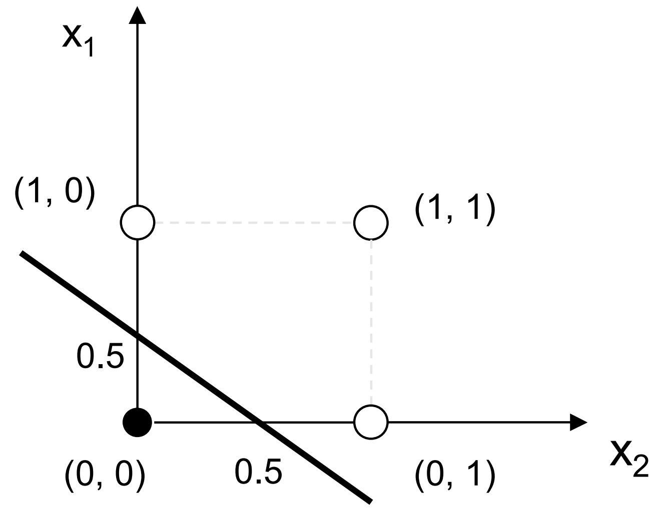
Recap: Linear separable problems

We can now plot the decision boundary of OR logic gate

OR

w1=1, w2=1, θ=0.5

OR		
x_1	x_2	out
0	0	0
0	1	1
1	0	1
1	1	1



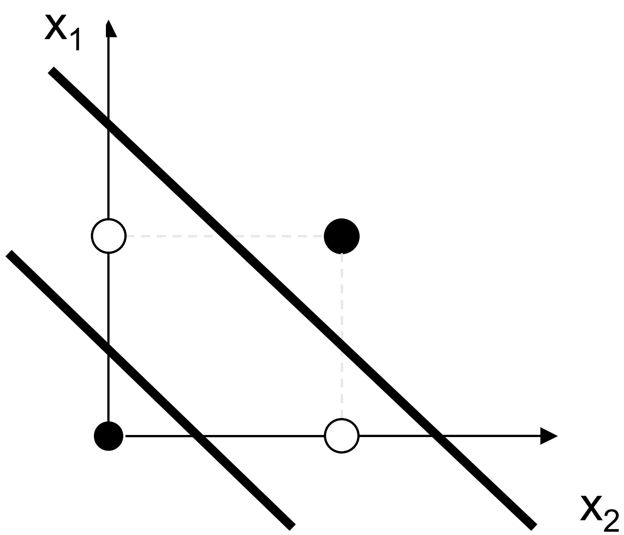
slope intercept

$$\sum_i w \cdot x_i = \theta \Leftrightarrow w_1 x_1 + w_2 x_2 = \theta \Leftrightarrow x_2 = -\left(\frac{w_1}{w_2}\right) x_1 + \left(\frac{\theta}{w_2}\right)$$

Recap: Non-linearly separable problems: 'XOR' gate

XOR		
x_1	x_2	out
0	0	0
0	1	1
1	0	1
1	1	0

Geometric representation

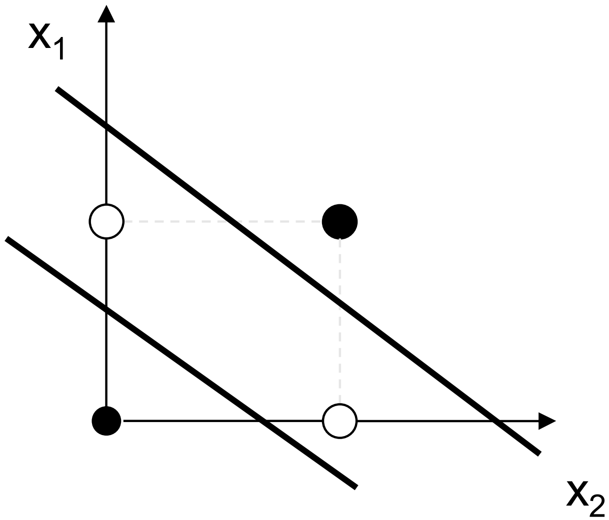


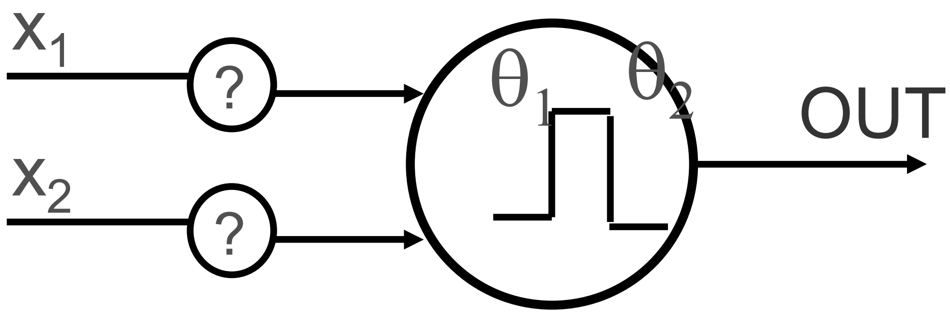
Solution: needs two lines to separate the data into two classes

Recap: Solving the 'XOR' problem: Change the activation function

Need two straight lines to separate the different outputs/decisions:

XOR		
x_1	x_2	out
0	0	0
0	1	1
1	0	1
1	1	0





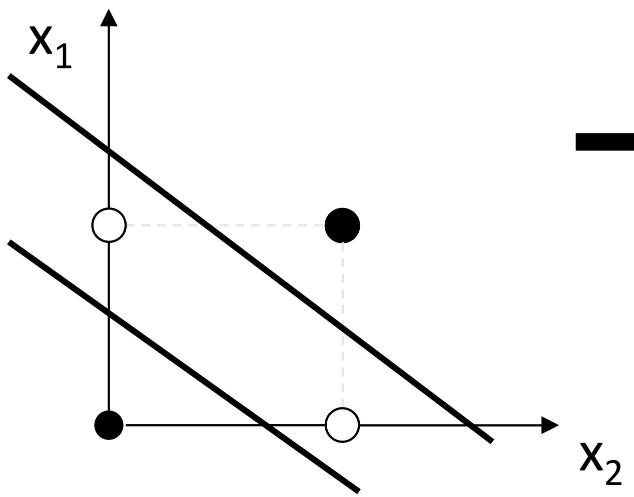
$\theta_1 = ?, \theta_2 = ?$

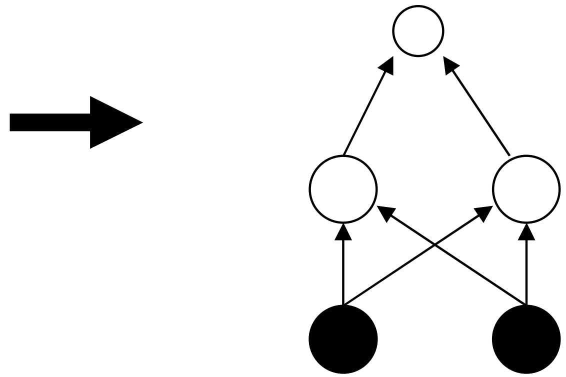
Find w_1, w_2, θ_1 and θ_2 that satisfies the XOR gate

Recap: Another solution to the 'XOR' problem

Recall that it is not possible to find weights that enable Single Layer Perceptrons to deal with non-linearly separable problems like XOR

XOR		
x_1	x_2	out
0	0	0
0	1	1
1	0	1
1	1	0





The proposed solution was to use a more complex network that is able to generate more complex decision boundaries. That network is the **Multi-Layer Perceptron**.

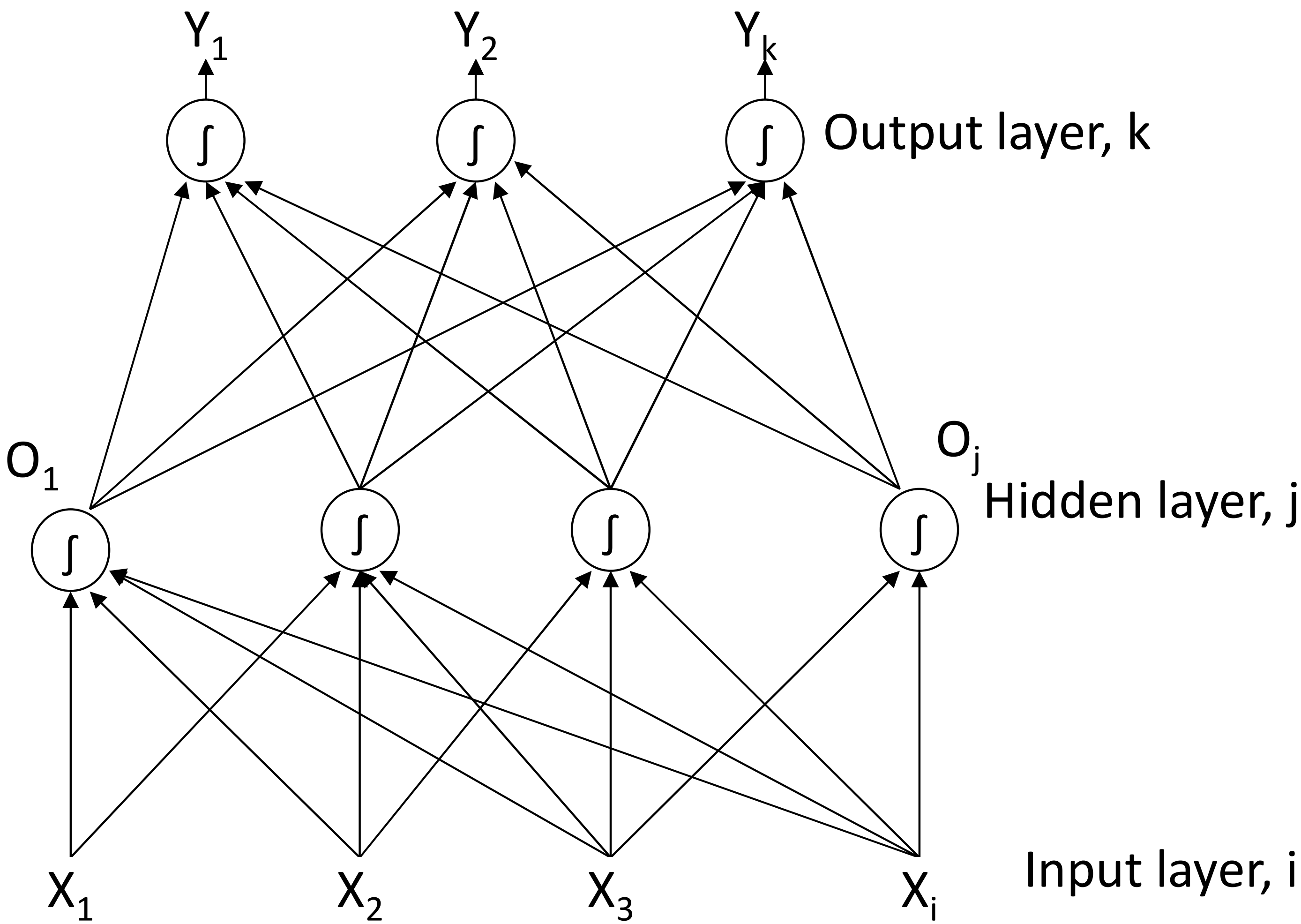
Outline

1. Recap from previous lectures
2. Multi-layer perceptron
3. Generalized delta rule (Backpropagation)
4. Practical considerations of MLP
5. Worked examples

Multi-Layer Perceptron (MLP)

$$Y_k = f(\sum_j w_{jk} \cdot O_j)$$

$$O_j = f(\sum_i w_{ij} \cdot X_i)$$



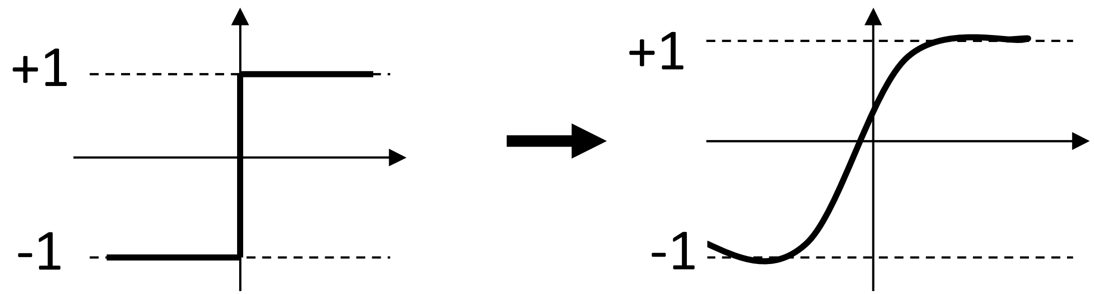
Generalized Delta Rule

Recall the PLR/Delta rule: Adjust neuron weights to reduce error at neuron's output:

$$w = w_{old} + \eta \delta x \qquad \text{where} \qquad \delta = y_{target} - y$$

Main problem: How to adjust the weights in the hidden layer, so they reduce the error in the output layer, when there is no specified target response in the hidden layer?

Solution: Alter the non-linear Perceptron (discrete threshold) activation function to make it differentiable and hence, help derive Generalized DR for MLP training.



Threshold function Sigmoid function

Sigmoid Function Properties

- Approximates the threshold function
- Smoothly differentiable everywhere
- Positive slope

$$y = f(a) = \frac{1}{1 + e^{-a}}$$

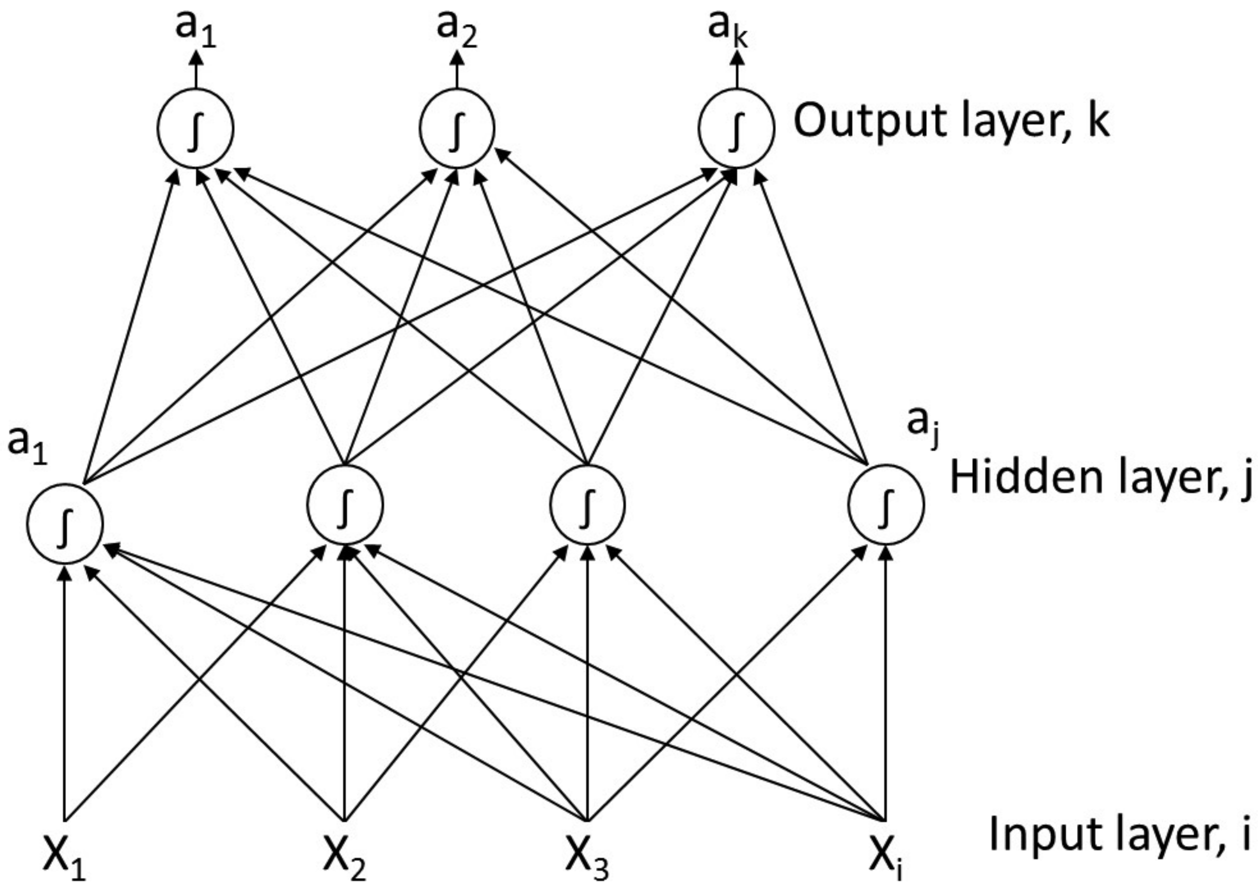
- Derivative of sigmoidal function is:

$$f'(a) = f(a) \cdot (1 - f(a))$$

Outline

1. Recap from previous lectures
2. Multi-layer perceptron
3. Generalized delta rule (Backpropagation)
4. Practical considerations of MLP
5. Worked examples

Derivation of backpropagation rule



k: output layer
j: hidden layer
i: input layer

w_{kj}: weight from hidden to output layer

w_{ji}: weight from input to hidden layer

a: output
t: target output
net: combined input

Calculus review

1. Chain rule: $$\frac{d(e^u)}{dx} = e^u \frac{du}{dx}$$

2. $$\frac{d(g+h)}{dx} = \frac{dg}{dx} + \frac{dh}{dx}$$

3. $$\frac{d(g^n)}{dx} = ng^{n-1}\frac{dg}{dx}$$

Gradient descent on error

$$E = \frac{1}{2}\sum_k (t_k - a_k)^2$$

Total error in the network

$$\Delta W \propto -\frac{\partial E}{\partial W}$$

Adjust network weights to reduce overall error

$$\Delta w_{kj} \propto -\frac{\partial E}{\partial w_{kj}}$$

$$\Delta w_{kj} = -\varepsilon \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial net_k} \frac{\partial net_k}{\partial w_{kj}}$$

via chain rule

Derivative of the error w.r.t. activation

Using $$\frac{d(g^n)}{dx} = ng^{n-1}\frac{dg}{dx}$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial(\frac{1}{2}(t_k - a_k)^2)}{\partial a_k} = -(t_k - a_k)$$

Derivative of activation w.r.t. net input

$$\frac{\partial a_k}{\partial net_k} = \frac{\partial (1 + e^{-net_k})^{-1}}{\partial net_k} = \frac{e^{-net_k}}{(1 + e^{-net_k})^2}$$

Notice:

$$1 - \frac{1}{1 + e^{-net_k}} = \frac{e^{-net_k}}{1 + e^{-net_k}}$$

Rewriting in terms of the activation function

$$a_k(1 - a_k)$$

Derivative of net input w.r.t. weight

$$\frac{\partial net_k}{\partial w_{kj}} = \frac{\partial (w_{kj} a_j)}{\partial w_{kj}} = a_j$$

Weight change rule for a hidden to output weight

- Substituting everything back

$$\Delta w_{kj} = \varepsilon \overbrace{(t_k - a_k)a_k(1 - a_k)}^{\delta_k} a_j$$

$$\Delta w_{kj} = \varepsilon \delta_k a_j$$

Weight change rule for an input to hidden weight

$$\Delta w_{ji} \propto -[\sum_k \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial net_k}\frac{\partial net_k}{\partial a_j}]\frac{\partial a_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

via chain rule

$$= \varepsilon[\sum_k \overbrace{(t_k - a_k)a_k(1-a_k)}^{\delta_k} w_{kj}]a_j(1-a_j)a_i$$

$$= \varepsilon\,\overbrace{[\sum_k \delta_k w_{kj}]a_j(1-a_j)}^{\delta_j}\,a_i$$

$$\Delta w_{ji} = \varepsilon\delta_j a_i$$

Backpropagation rule

So, the weight change from the input layer unit *i* to hidden layer unit *j* is:

$$\boxed{\Delta w_{ji} = \varepsilon \delta_j a_i}$$

where

$$\boxed{\varepsilon \overbrace{[\sum_k \delta_k w_{kj}] a_j (1 - a_j)}^{\delta_j} a_i}$$

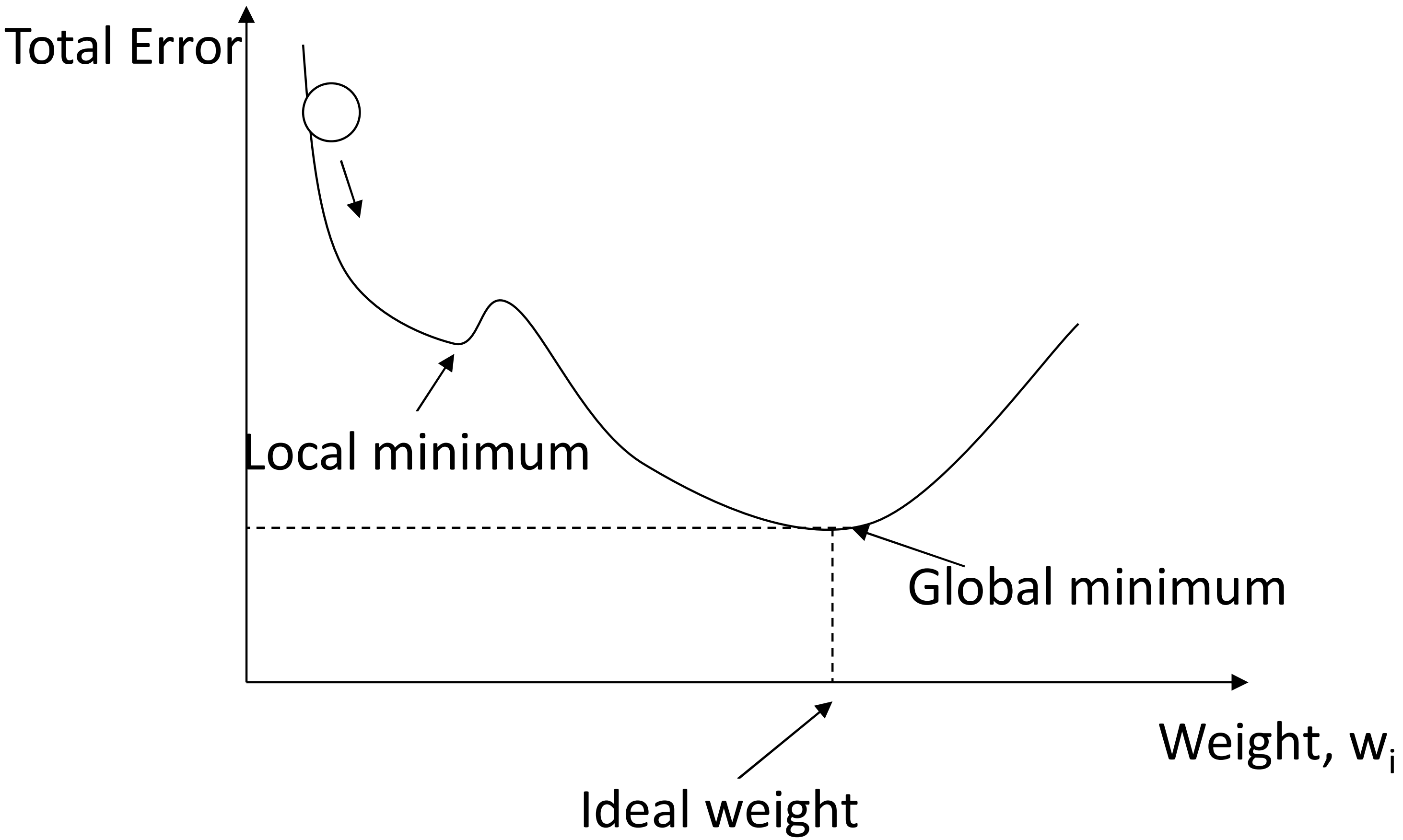
The weight change from the hidden layer unit *j* to the output layer unit *k* is:

$$\boxed{\Delta w_{kj} = \varepsilon \delta_k a_j}$$

where

$$\boxed{\varepsilon \overbrace{(t_k - a_k) a_k (1 - a_k)}^{\delta_k} a_j}$$

Graphical Representation of GDR



Training a two-layer feed forward network

1. Take the set of training patterns you wish the network to learn
2. Set up the network with **N** input units fully connected to **M** hidden non-linear hidden units via connections with weights w_{ij}, which in turn are fully connected to **P** output units via connections with weights w_{jk}
3. Generate random initial weights, e.g. from range [-wt, +wt]
4. Select appropriate error function $E(w_{jk})$ and learning rate η
5. Apply the weight update equation $\Delta w_{jk}=-\eta\partial E(w_{jk})/\partial w_{jk}$ to each weight w_{jk} for each training pattern p.
6. Do the same to all hidden layers.
7. Repeat step 5-6 until the network error function is ‘small enough’

Outline

1. Recap from previous lectures
2. Multi-layer perceptron
3. Generalized delta rule (Backpropagation)
4. Practical considerations of MLP
5. Worked examples

Practical Considerations

1. Do we need to pre-process the training data? If so, how?
2. How do we choose the initial weights from which we start the training?
3. How do we choose an appropriate learning rate η?
4. Should we change the weights after each training pattern, or after the whole set?
5. Are some activation/transfer functions better than others?
6. How do we avoid local minima in the error function?
7. How do we know when we should stop the training?
8. How many hidden units do we need?
9. Should we have different learning rates for the different layers?

Pre-processing of training data

- Training data should be representative
  - Not too many examples of one type at the expense of another.
  - If one class of pattern is easy to learn, having large numbers of patterns from that class in the training set will only slow down the over-all learning process.

- Rescale input data if continuous
  - Shift zero of the scale so that the mean value of each input is near zero
  - Normalise so std of values for each input are roughly the same

- On-line training
  - shuffle the order of the training data each epoch.

Choosing the Initial Weight Values

- Never start all weights start from the same values
  - Learning rule will change weights the same way, so all the hidden units will end up doing the same thing and the network will never learn properly.
- We generally start off all the weights with small random values.
  - Take values from a flat distribution around zero [–*smwt*, +*smwt*], or
  - From a Gaussian distribution around zero with standard deviation *smwt*.
- When choosing a value for *smwt* make it as large as you can without saturating any of the sigmoids.
- Train network from a number of different random initial weight sets to make sure performance is independent of initial weight values

Choosing the Learning Rate

- Choosing a good value for the learning rate ε is constrained by two opposing facts:
  1. If ε is too small, it will take too long to get anywhere near the minimum of the error function.
  2. If ε is too large, the weight updates will over-shoot the error minimum and the weights will oscillate, or even diverge.

- Finding the optimal value is very problem and network dependent, so one cannot formulate reliable general prescriptions.

- Try a range of different values (e.g. ε = 0.1, 0.01, 1.0, 0.0001) and use the results as a guide.

Batch Training vs. On-line Training

- **Batch Training:** update the weights after all training patterns have been presented.

- **On-line Training (or Sequential Training):** update all the weights immediately after processing each training pattern.
  - Individual weight changes can be rather erratic.
  - A much lower learning rate ε will be necessary than for batch learning.
  - Each weight has *npatterns* updates per epoch, rather than just one **=>** learning is much quicker
  - Particularly if there is a lot of redundancy in the training data, i.e. many training patterns containing similar information.

Choosing the Transfer Function

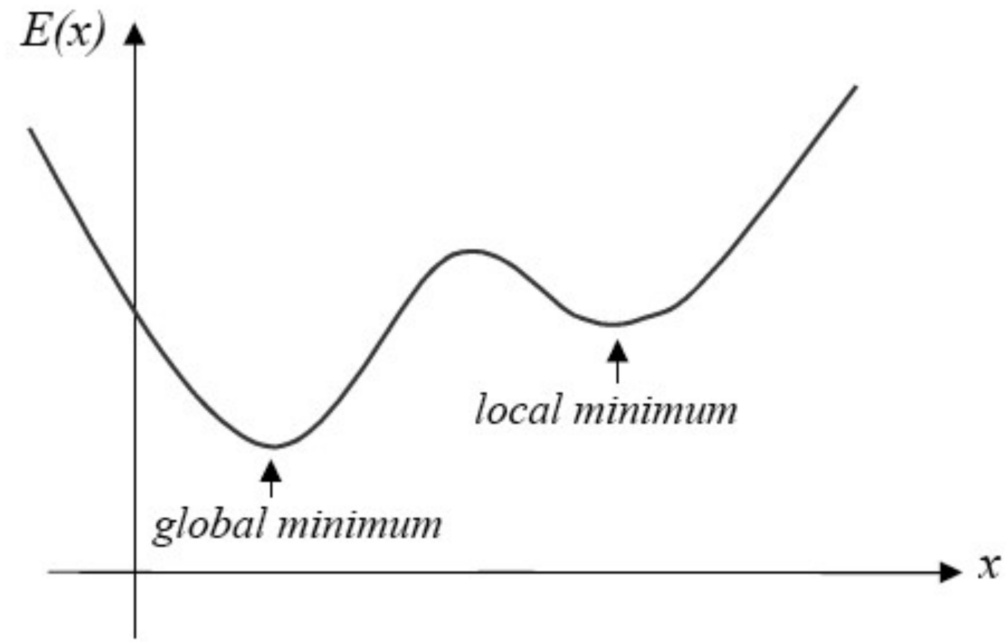
- A differentiable transfer/activation function is important for the gradient descent algorithm to work.

- The logistic function ranges from 0 to 1. There is some evidence that an anti-symmetric transfer function, i.e. one that satisfies $f(-x) = -f(x)$, enables the gradient descent algorithm to learn faster.

- When outputs are continuous real values, then sigmoidal transfer functions no longer makes sense. Thus, a simple linear transfer function $f(x) = x$ is appropriate.

Local Minima

Cost functions can quite easily have more than one minimum:



- If we start off in the vicinity of the local minimum, we may end up at the local minimum rather than the global minimum.
- Starting with a range of different initial weight sets increases our chances of finding the global minimum.

When to Stop Training

- The Sigmoid(x) function reach its extreme values of 0 and 1 when $x = \pm\infty$.
- Network achieves its binary targets when at least some of its weights reach $\pm\infty$.
- Given finite gradient descent step sizes, our networks will never reach their binary targets.
- Even if we off-set the targets (to 0.1 and 0.9 say) we will generally require an infinite number of increasingly small gradient descent steps to achieve those targets.
- Clearly, if the training algorithm can never actually reach the minimum, we have to stop the training process when it is ‘near enough’.
  - Stop the training when the sum squared error function becomes less than a particular small value (0.2 say).

How Many Hidden Units?

- Best number of hidden units depends on
  - Number of training patterns
  - Numbers of input and output units
  - Amount of noise in the training data
  - Complexity of the function or classification to be learned
  - Type of hidden unit activation function
  - Training algorithm

- Too few hidden units will generally leave high training and generalisation errors due to under-fitting.
- Too many hidden units will result in low training errors, but will make the training unnecessarily slow, and will result in poor generalisation unless some other technique (such as *regularisation*) is used to prevent over-fitting.
- A sensible strategy is to try a range of numbers of hidden units and see which works best.

Different Learning Rates for Different Layers?

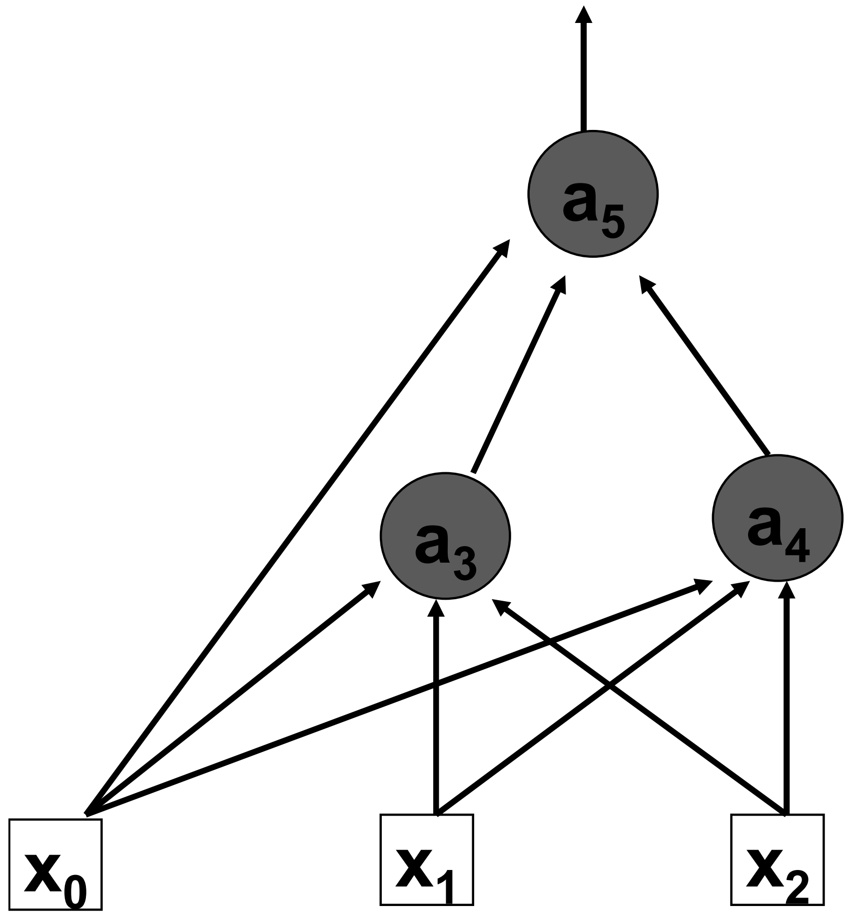
- A network usually learns most efficiently if all its neurons are learning at roughly the same speed.
- A number of factors affect the choices:
  - Later network layers (nearer the outputs) have larger local gradients (*deltas*) than the earlier layers (nearer the inputs).
  - Activations of units with many connections feeding into or out of them change faster than units with fewer connections.
  - Activations required for linear units will be different for Sigmoidal units.
  - Empirical evidence showed better to have different learning rates η for the thresholds/biases.
- In practice, it is often quicker to just use the same rates η for all the weights and thresholds, rather than spending time trying to work out appropriate differences.
- **Solution:** use evolutionary strategies to determine good learning rates.

Outline

1. Recap from previous lectures
2. Multi-layer Perceptron
3. Generalized Delta Rule (a.k.a. Backpropagation algorithm)
4. Practical considerations of MLP
5. Worked examples

Example

- Three-layer feedforward neural network
- Layer 1: 3 units
- Layer 2: 2 units
- Layer 3: 1 unit
- Connectivity: all-to-all



…(2)

Input patterns: $x_1 = 1$, $x_2 = 0$

Bias input: $x_0 = 1$

Weights: $w_{13} = 3$, $w_{14} = 6$, $w_{03} = 1$, $w_{04} = -6$, $w_{23} = 4$, $w_{24} = 5$, $w_{05} = -3.93$, $w_{35} = 2$, $w_{45} = 4$

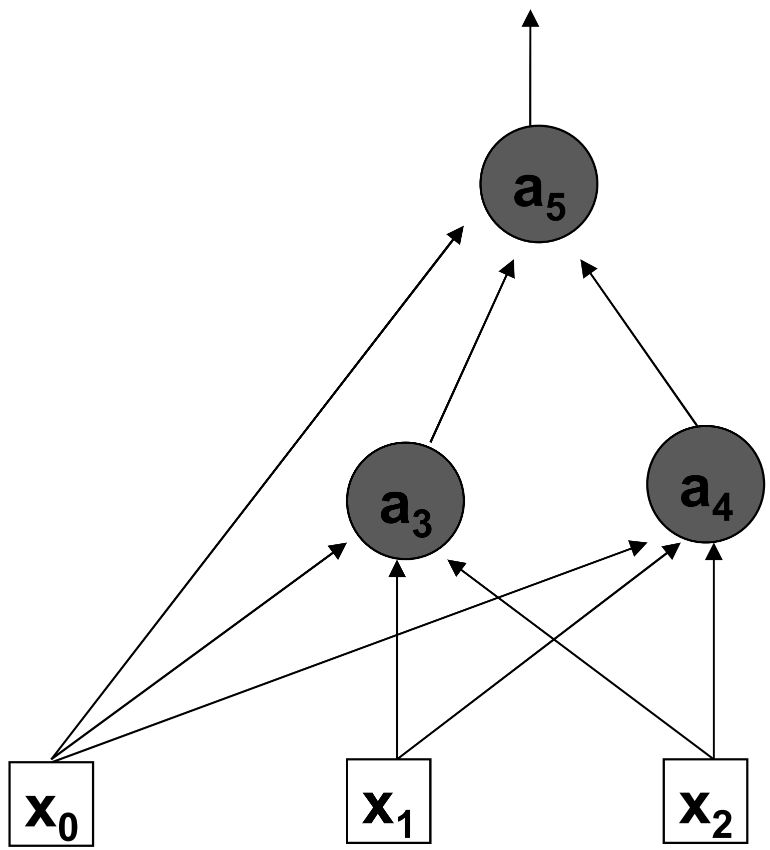
Unit output:

$$y_j = f(a_j) = \frac{1}{1+e^{-a_j}}$$

Combined input:

$$a_j = \sum_i w_{ij} \cdot x_i$$

Target output: $y_{target} = 1$



…(3)

- For unit j = 3:

$$a_3 = 1*1 + 3*1 + 4*0 = 4$$

$$y_3 = f(a_3) = f(4) = 0.982$$

- For unit j =4:

$$a_4 = 1*(-6) + 1*6 + 0*5 = 0$$

$$y_4 = f(a_4) = f(0) = 0.5$$

- For unit j=5

$$a_5 = 1*(-3.93) + 0.982*2 + 4*0.5 = 0.04$$

$$y_3 = f(a_3) = f(0.04) = 0.51$$

So, the error between the NETWORK OUTPUT and the TARGET OUTPUT is: $(y_{target} – y) = (1-0.51) = 0.49$

Weight Update Rule

Generally, weight change from any unit j to unit k by gradient descent (i.e. weight change by small increment in negative direction to the gradient) is now called **Generalized Delta Rule** (**GDR** or **Backpropagation**):

$$\Delta w = w - w_{old} = -\eta \frac{\partial E}{\partial w} = +\eta \delta x$$

So, the weight change from the input layer unit *i* to hidden layer unit *j* is:

$$\Delta w_{ij} = \eta \cdot \delta_j \cdot x_i \quad \text{where} \quad \boxed{\delta_j = o_j (1 - o_j) \sum_k w_{jk} \cdot \delta_k}$$

The weight change from the hidden layer unit *j* to the output layer unit *k* is:

$$\Delta w_{jk} = \eta \cdot \delta_k \cdot o_j \quad \text{where} \quad \boxed{\delta_k = (y_{target,k} - y_k) y_k (1 - y_k)}$$

Backward pass

- $\Delta w_{03}=\eta\delta_3 x_0$ = 0.1*0.0043*1=0.00043
- $\delta_3=y_3(1-y_3)w_{35}\delta_5$=0.982*(1-0.982)*2*(1-0.51)*0.51*(1-0.51)=0.0043
- $\Delta w_{04}=\eta\delta_4 x_0$ = 0.1*0.1225*1=0.01225
- $\delta_4=y_4(1-y_4)w_{45}\delta_5$=0.5*(1-0.5)*4*(1-0.51)*0.51*(1-0.51)=0.1225
- $\Delta w_{13}=\eta\delta_3 x_1$ = 0.1*0.0043*1 = 0.00043
- $\delta_3=y_3(1-y_3)w_{35}\delta_5$=0.982*(1-0.982)*2*(1-0.51)*0.51*(1-0.51)=0.0043
- $\Delta w_{14}=\eta\delta_4 x_1$=0.1*0.1225*1=0.01225
- $\delta_4=y_4(1-y_4)w_{45}\delta_5$=0.5*(1-0.5)*4*(1-0.51)*0.51*(1-0.51)=0.1225

…(2)

- $\Delta w_{23}=\eta\delta_3 x_2$ = 0.1*0.0043*0 = 0
- $\delta_3=y_3(1-y_3)w_{35}\delta_5$=0.982*(1-0.982)*2*(1-0.51)*0.51*(1-0.51)=0.0043
- $\Delta w_{24}=\eta\delta_4 x_2$=0.1*0.1225*0=0
- $\delta_4=y_4(1-y_4)w_{45}\delta_5$=0.5*(1-0.5)*4*(1-0.51)*0.51*(1-0.51)=0.1225
- $\Delta w_{35}=\eta\delta_5 y_3$=0.1*0.1225*0.982=0.012
- $\delta_5=(y_{target}-y_5)y_5(1-y_5)$=(1-0.51)*0.51*(1-0.51)=0.1225
- $\Delta w_{45}=\eta\delta_5 y_4$=0.1*0.1225*0.5=0.0061
- $\delta_5=(y_{target}-y_5)y_5(1-y_5)$=(1-0.51)*0.51*(1-0.51)=0.1225

…(3)

Similarly for all weights w_{ij}:

i	j	w_{ij}	δ_j	y_i	Updated w_{ij}
0	3	**1**	0.0043	1.0	**1.0004**
1	3	**3**	0.0043	1.0	**3.0004**
2	3	**4**	0.0043	0.0	**4.0000**
0	4	**-6**	0.1225	1.0	**-5.9878**
1	4	**6**	0.1225	1.0	**6.0123**
2	4	**5**	0.1225	0.0	**5.0000**
0	5	**-3.92**	0.1225	1.0	**-3.9078**
3	5	**2**	0.1225	0.9820	**2.0120**
4	5	**4**	0.1225	0.5	**4.0061**

Verification that it works!

On **the next forward pass**:
The new activations are:
$y_3 = f(4.0008) = 0.9820$
$y_4 = f(0.0245) = 0.5061$
$y_5 = f(0.0955) = \mathbf{0.5239}$

Thus the **new error**

$(y_{target} - y_5) = (1\text{-}0.5239) = 0.476$

has been reduced by **0.014**
(from **0.490** to **0.476**)