UNIVERSITY OF LEEDS

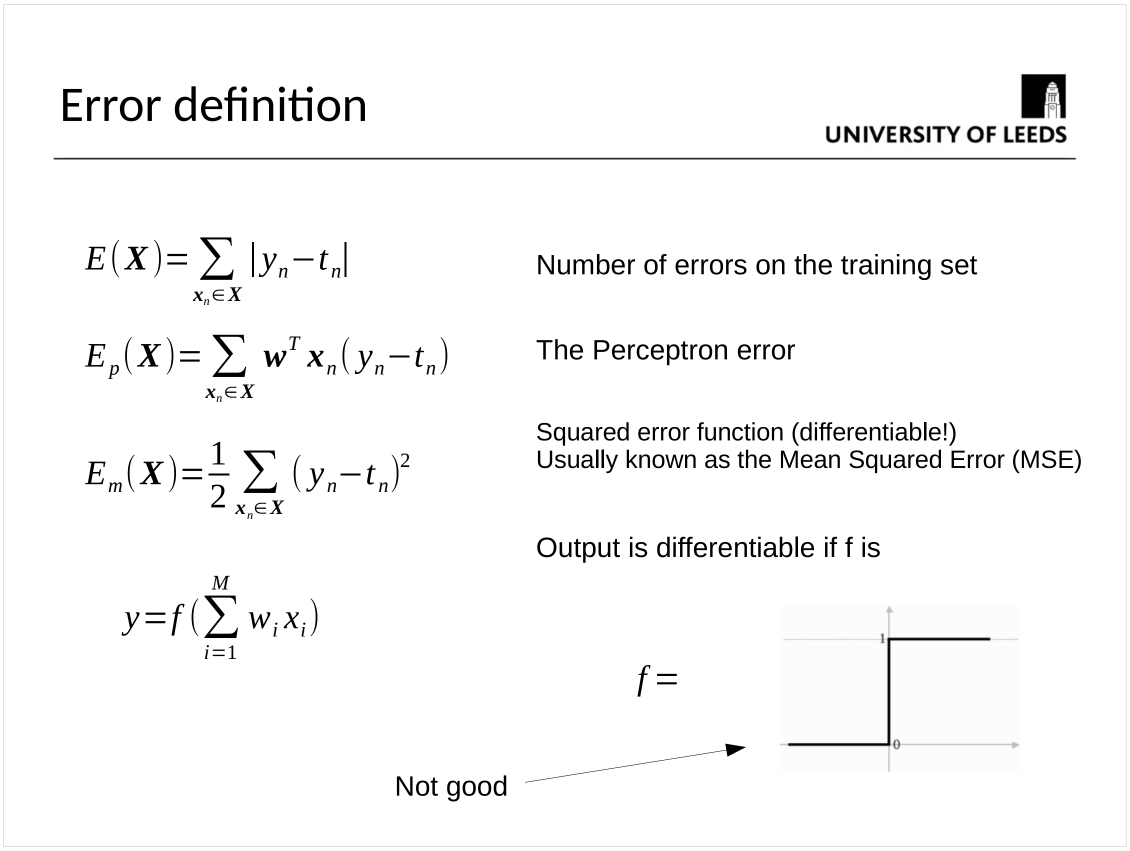
Class: Machine Learning

Multi-Layer Neural Networks

Instructor: Matteo Leonetti

Learning outcomes

- Define an appropriate error to minimise for Feed-forward neural networks.
- Derive the update rule of the weights of the NN, through backpropagation.
- Apply NNs to real-world data sets



Error definition

UNIVERSITY OF LEEDS

$$E(\boldsymbol{X})=\sum_{\boldsymbol{x}_n\in\boldsymbol{X}}|y_n-t_n|$$

Number of errors on the training set

$$E_p(\boldsymbol{X})=\sum_{\boldsymbol{x}_n\in\boldsymbol{X}}\boldsymbol{w}^T\boldsymbol{x}_n(y_n-t_n)$$

The Perceptron error

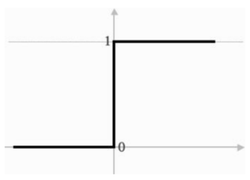
$$E_m(\boldsymbol{X})=\frac{1}{2}\sum_{\boldsymbol{x}_n\in\boldsymbol{X}}(y_n-t_n)^2$$

Squared error function (differentiable!)
Usually known as the Mean Squared Error (MSE)

$$y=f\left(\sum_{i=1}^{M}w_i x_i\right)$$

Output is differentiable if f is

$f=$



Not good

In the definition of the perceptron error, we took advantage of the fact that the perceptron implements a linear decision boundary, and we noted that the value of the linear function evaluated on any point was proportional to the distance of the point from the hyperplane.

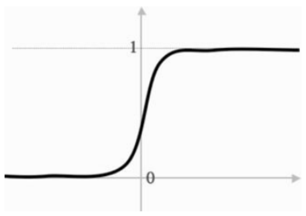
The function implemented by the multi-layer perceptron, however, is not linear, and quite a bit more complicated. Consequently, we need a more general definition of the error.

Going back to the original idea of using the number of mistakes, we can do something similar, but differentiable, such as the mean squared error (MSE)

The MSE is differentiable (indeed, it's just a quadratic function) with respect to the parameters, as long as the activation function is also differentiable. So far we used a step function for the activation, which is not good in this case.

A different activation function

UNIVERSITY OF LEEDS



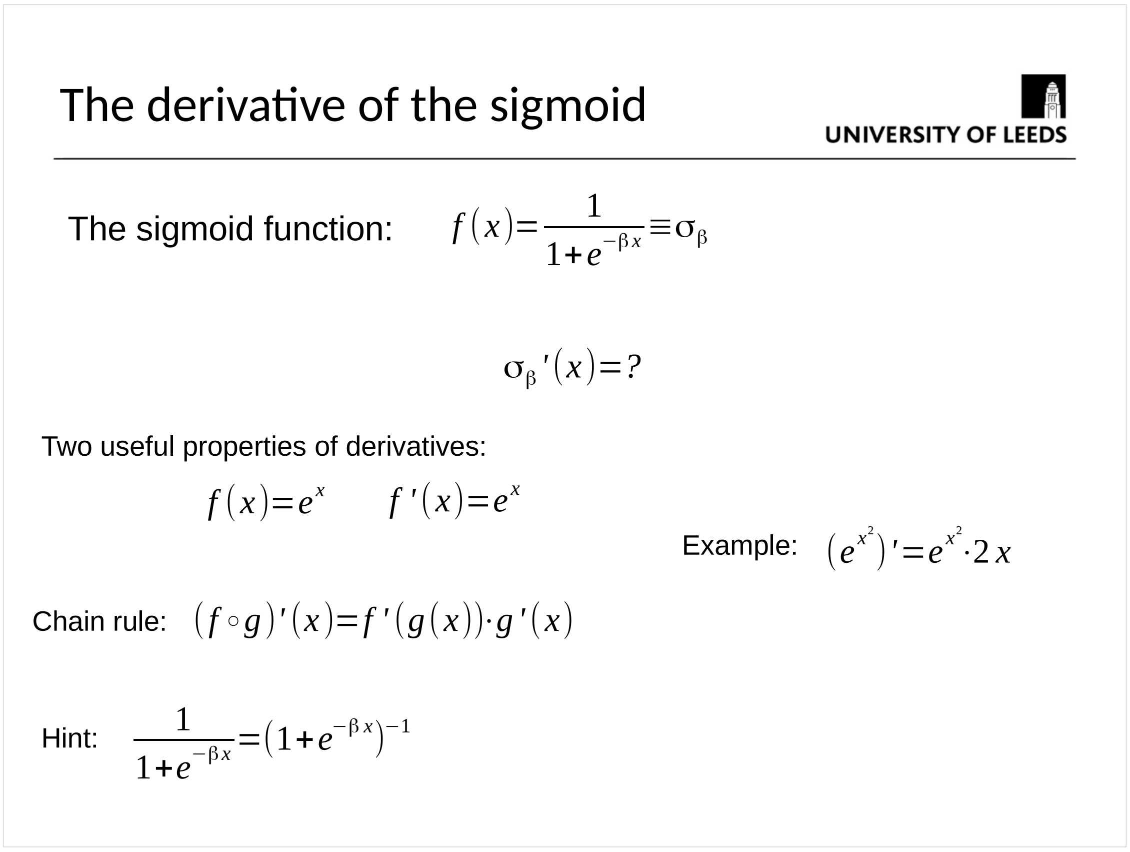
The sigmoid function: $f(x) = \frac{1}{1+e^{-\beta x}} \equiv \sigma_\beta$

$$\sigma_\beta'(x) = ?$$

We substitute the step function with a function called sigmoid.

The sigmoid is similar to the step function in shape, but it’s rounded at the edges, so that it is differentiable everywhere.

What is its derivative?



The derivative of the sigmoid

UNIVERSITY OF LEEDS

The sigmoid function: $f(x)=\frac{1}{1+e^{-\beta x}}\equiv\sigma_\beta$

$$\sigma_\beta{}'(x)=?$$

Two useful properties of derivatives:

$$f(x)=e^x \qquad f'(x)=e^x$$

Example: $\left(e^{x^2}\right)'=e^{x^2}\cdot 2x$

Chain rule: $(f\circ g)'(x)=f'(g(x))\cdot g'(x)$

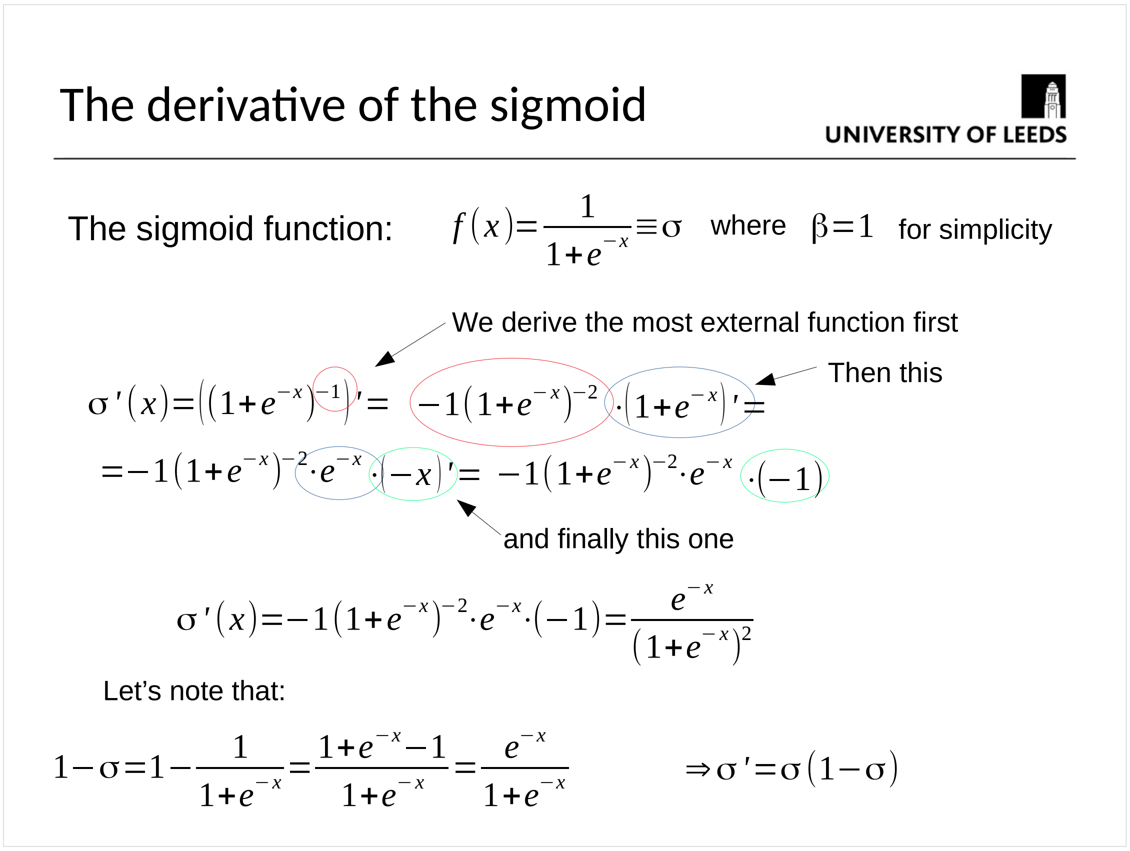
Hint: $\frac{1}{1+e^{-\beta x}}=\left(1+e^{-\beta x}\right)^{-1}$

In order to compute the derivative of the sigmoid, we need to remember a couple of properties of derivatives.

The first one is that the derivative of the exponential is the exponential itself.

The second one is called the “chain” rule: the derivative of the composition of two functions is the product of the derivative of the most external function with the derivative of the internal one.

Now, all we need to compute the derivative of the sigmoid is to note that $1/x = x^{-1}$, and we know how to derive exponents!



The derivative of the sigmoid

UNIVERSITY OF LEEDS

The sigmoid function: $f(x)=\frac{1}{1+e^{-x}}\equiv\sigma$ where $\beta=1$ for simplicity

We derive the most external function first

Then this

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

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

and finally this one

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

Let's note that:

$$1-\sigma=1-\frac{1}{1+e^{-x}}=\frac{1+e^{-x}-1}{1+e^{-x}}=\frac{e^{-x}}{1+e^{-x}} \qquad \Rightarrow\sigma'=\sigma(1-\sigma)$$

The derivative of the sigmoid can be conveniently expressed in terms of the sigmoid itself!

Here I have circled with the same colour the original function and its derivative. At every step one more function in the composition is derived thanks to the chain rule.

The same thing with β, FYI

UNIVERSITY OF LEEDS

The sigmoid function: $h(x)=\frac{1}{1+e^{-\beta x}}\equiv\sigma_\beta$

We derive the most external function first

Then this

$$\sigma_\beta{}'(x)=\left(\left(1+e^{-\beta x}\right)^{-1}\right)'=-1\left(1+e^{-\beta x}\right)^{-2}\cdot\left(1+e^{-\beta x}\right)'=$$

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

and finally this one

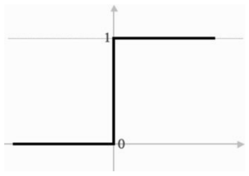
$$\sigma_\beta{}'(x)=-1\left(1+e^{-\beta x}\right)^{-2}\cdot e^{-\beta x}\cdot(-\beta)=\frac{\beta e^{-\beta x}}{\left(1+e^{-\beta x}\right)^{2}}$$

Let's note that:

$$1-\sigma_\beta=1-\frac{1}{1+e^{-\beta x}}=\frac{1+e^{-\beta x}-1}{1+e^{-\beta x}}=\frac{e^{-\beta x}}{1+e^{-\beta x}}\qquad\Rightarrow\sigma_\beta{}'=\beta\sigma_\beta\left(1-\sigma_\beta\right)$$

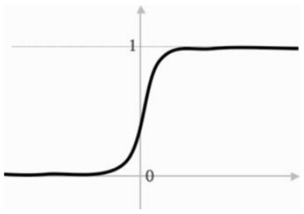
A different activation function

UNIVERSITY OF LEEDS



before

$$y(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}) = \begin{cases} 1 & \text{if} \quad \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} > 0 \\ 0 & \text{if} \quad \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} \leq 0 \end{cases}$$

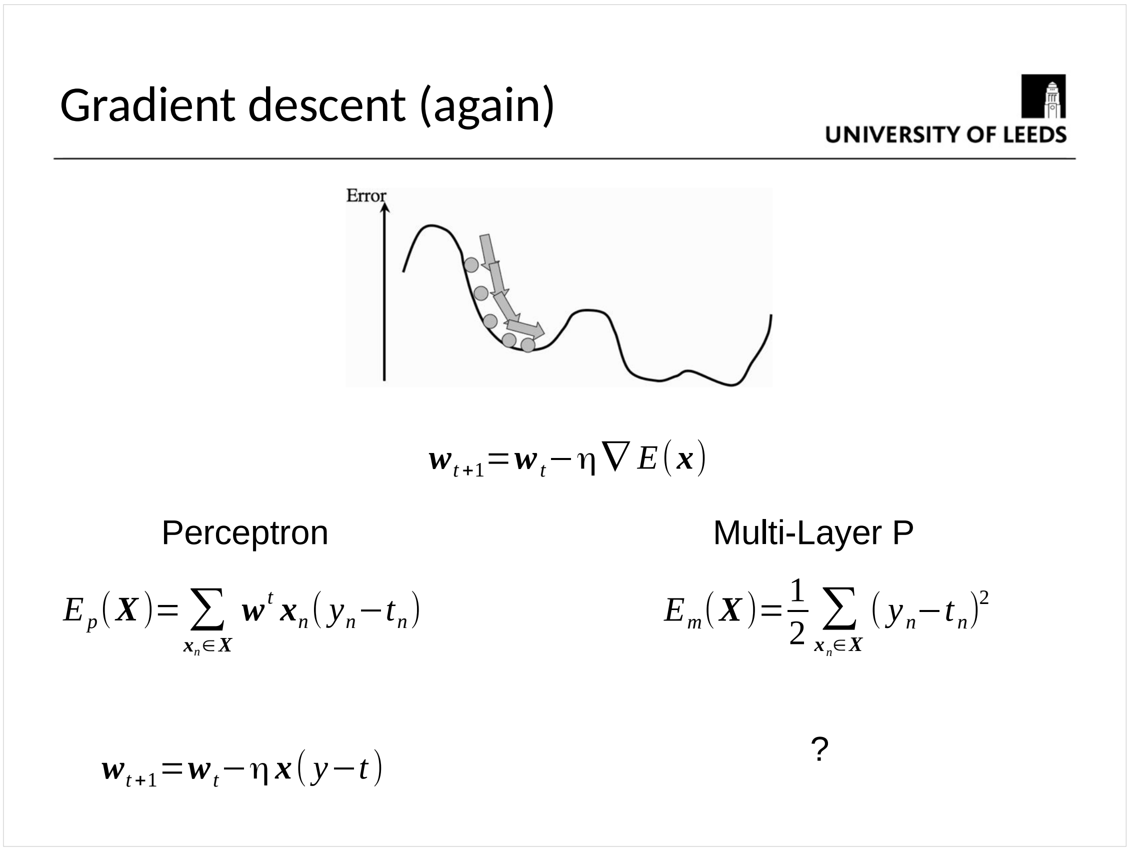


after

$$y(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}) = \frac{1}{1+e^{-\beta \boldsymbol{w}^{T}\boldsymbol{x}}}$$

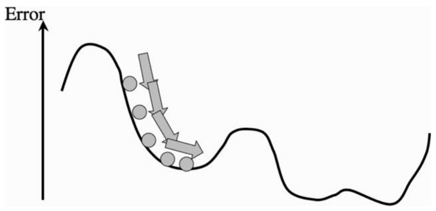
$$\sigma_\beta{}'(x) = \beta \frac{e^{-\beta x}}{(1+e^{-\beta x})^2} = \beta\,\sigma_\beta(x)(1-\sigma_\beta(x))$$

From now on, we are going to consider β =1, so that it will not appear in our equations anymore.



Gradient descent (again)

UNIVERSITY OF LEEDS



$$\boldsymbol{w}_{t+1}=\boldsymbol{w}_t-\eta\nabla E(\boldsymbol{x})$$

Perceptron

$$E_p(\boldsymbol{X})=\sum_{\boldsymbol{x}_n\in\boldsymbol{X}}\boldsymbol{w}^t\boldsymbol{x}_n(y_n-t_n)$$

$$\boldsymbol{w}_{t+1}=\boldsymbol{w}_t-\eta\,\boldsymbol{x}(y-t)$$

Multi-Layer P

$$E_m(\boldsymbol{X})=\frac{1}{2}\sum_{\boldsymbol{x}_n\in\boldsymbol{X}}(y_n-t_n)^2$$

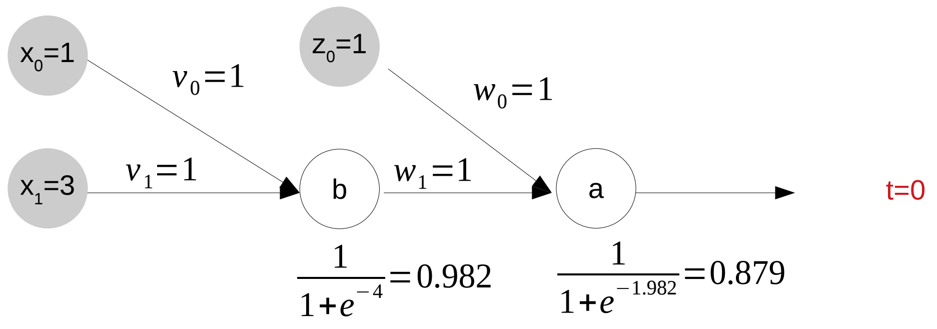
?

We are now ready to do gradient descent for the multi-layer perceptron!

What is the update rule for the weights of an MLP?

Example

UNIVERSITY OF LEEDS

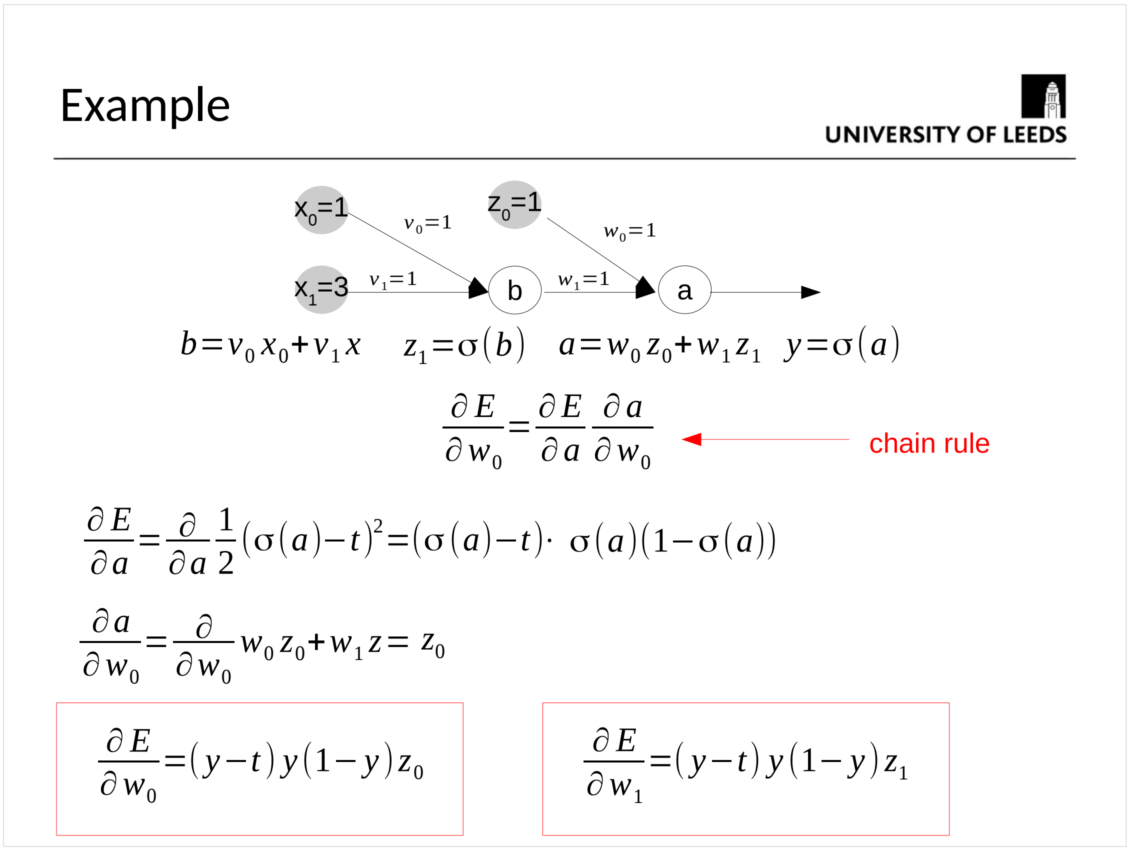


Let’s start with a simple example. This MLP taskes one variable in input, plus the bias.

The output of the MLP is 0.879, that is, close to 1, but let’s imagine that the target value for the input 3 is 0.

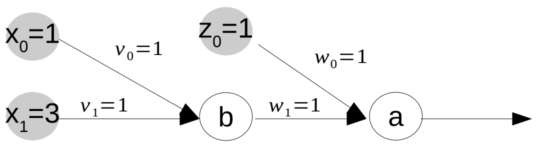
Clearly the weights must be reduced, to lower the output. Which weights should be reduced, and by how much?

The gradient of the error with respect to the weights will give us the direction of maximum improvement of the error.



Example

UNIVERSITY OF LEEDS



$$b=v_0 x_0+v_1 x \qquad z_1=\sigma(b) \qquad a=w_0 z_0+w_1 z_1 \qquad y=\sigma(a)$$

$$\frac{\partial E}{\partial w_0}=\frac{\partial E}{\partial a}\frac{\partial a}{\partial w_0}$$ ← chain rule

$$\frac{\partial E}{\partial a}=\frac{\partial}{\partial a}\frac{1}{2}(\sigma(a)-t)^2=(\sigma(a)-t)\cdot\ \sigma(a)(1-\sigma(a))$$

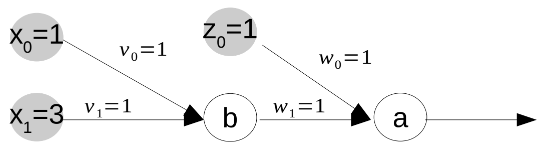
$$\frac{\partial a}{\partial w_0}=\frac{\partial}{\partial w_0}w_0 z_0+w_1 z= z_0$$

$$\boxed{\frac{\partial E}{\partial w_0}=(y-t)y(1-y)z_0} \qquad \boxed{\frac{\partial E}{\partial w_1}=(y-t)y(1-y)z_1}$$

We can use the chain rule to decompose the derivative with respect to the weights w_0 and w_1 in two parts: the derivative with respect to the liner component *a*, and the derivative of *a* with respect to the particular weight.

Example

UNIVERSITY OF LEEDS



$b=v_0x_0+v_1x \qquad z_1=\sigma(b) \qquad a=w_0z_0+w_1z_1 \qquad y=\sigma(a)$

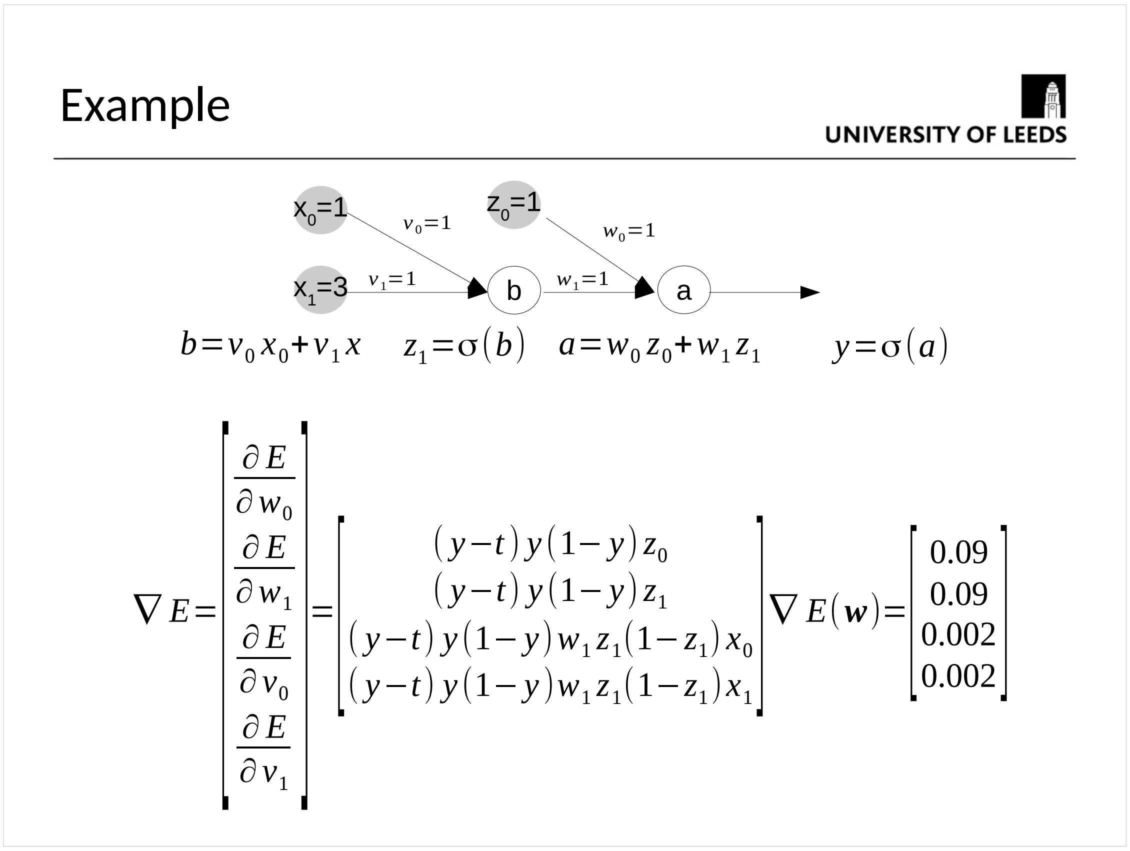
$$a=w_0z_0+w_1\sigma(b)$$

$$\frac{\partial E}{\partial v_0}=\frac{\partial E}{\partial a}\frac{\partial a}{\partial b}\frac{\partial b}{\partial v_0} \qquad \frac{\partial E}{\partial a}=(y-t)y(1-y)$$ from before

$$\frac{\partial a}{\partial b}=\frac{\partial}{\partial b}w_0z_0+w_1\sigma(b)=w_1\sigma(b)(1-\sigma(b))=w_1z_1(1-z_1)$$

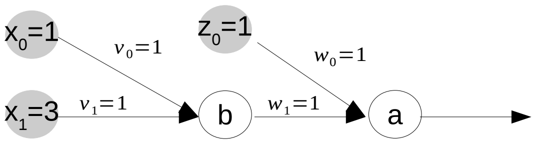
$$\frac{\partial}{\partial v_0}v_0x_0+v_1x_1=x_0$$

$$\boxed{\frac{\partial E}{\partial v_0}=(y-t)y(1-y)w_1z_1(1-z_1)x_0} \qquad \boxed{\frac{\partial E}{\partial v_1}=(y-t)y(1-y)w_1z_1(1-z_1)x_1}$$



Example

UNIVERSITY OF LEEDS



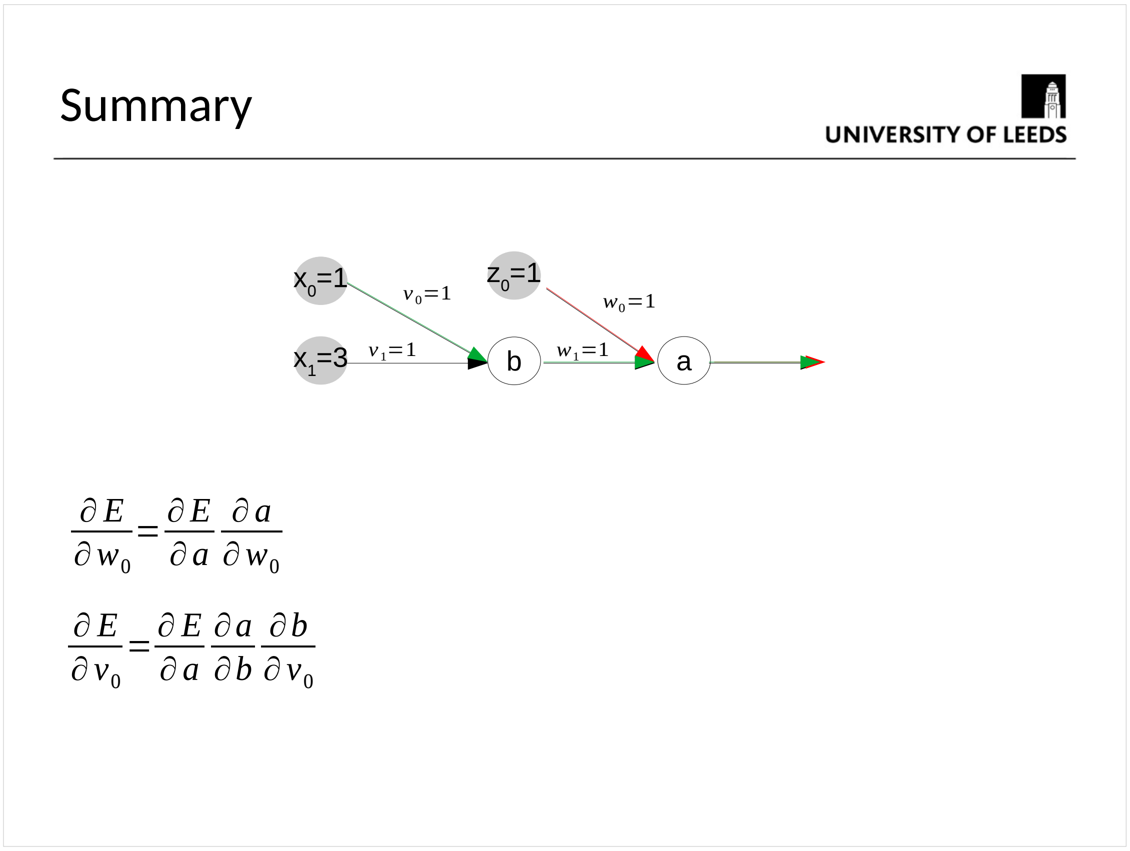
$$b=v_0 x_0+v_1 x \qquad z_1=\sigma(b) \qquad a=w_0 z_0+w_1 z_1 \qquad y=\sigma(a)$$

$$\nabla E=\begin{bmatrix}\frac{\partial E}{\partial w_0}\\ \frac{\partial E}{\partial w_1}\\ \frac{\partial E}{\partial v_0}\\ \frac{\partial E}{\partial v_1}\end{bmatrix}=\begin{bmatrix}(y-t)y(1-y)z_0\\ (y-t)y(1-y)z_1\\ (y-t)y(1-y)w_1 z_1(1-z_1)x_0\\ (y-t)y(1-y)w_1 z_1(1-z_1)x_1\end{bmatrix}\nabla E(\boldsymbol{w})=\begin{bmatrix}0.09\\ 0.09\\ 0.002\\ 0.002\end{bmatrix}$$

Recall that we are minimizing, therefore the gradient must be mutiplied by -1 (we follow the anti-gradient).

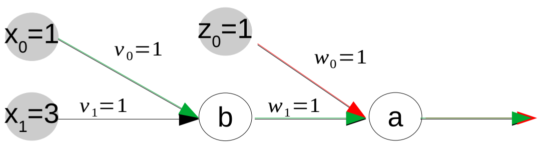
Note how the most external layer affects the error the most, while the inner layers act through the externals ones. The more layers we have, the more the effect of the first layers is small.

Neural Networks suffer from the problem of *vanishing* gradients: after a certain number of layers (depending on the architecture, but in the order of 10) the gradient of the weights near the inputs is close to zero, that is, they barely affect the error.



Summary

UNIVERSITY OF LEEDS



$$\frac{\partial E}{\partial w_0}=\frac{\partial E}{\partial a}\frac{\partial a}{\partial w_0}$$

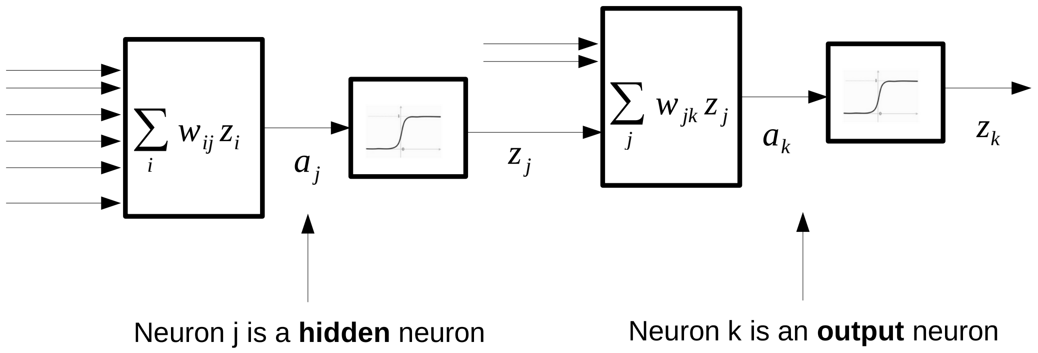
$$\frac{\partial E}{\partial v_0}=\frac{\partial E}{\partial a}\frac{\partial a}{\partial b}\frac{\partial b}{\partial v_0}$$

The chain rule is the key to back-propagation!

The path to a weight determines the chain.

Backpropagation of errors, notation

UNIVERSITY OF LEEDS


$\sum_i w_{ij} z_i$
a_j
z_j
$\sum_j w_{jk} z_j$
a_k
z_k
Neuron j is a **hidden** neuron
Neuron k is an **output** neuron

$$\ldots \qquad a_j = \sum_{i=1}^{N} w_{ij} z_i \qquad z_j = f(a_j) \qquad a_k = \sum_{j=1}^{M} w_{jk} z_j \qquad z_k = f(a_k)$$

We do the same thing as in the example, but in a more general form.

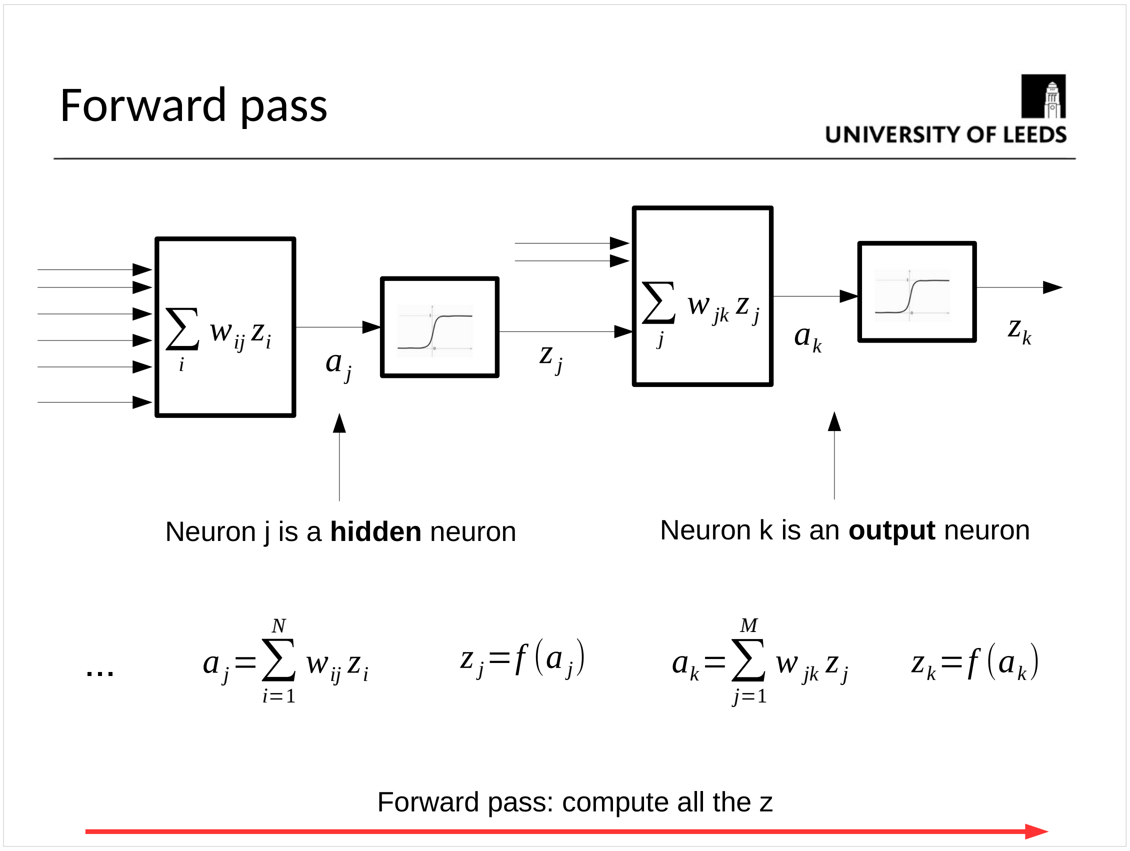
This is the structure of an MLP with one hidden layer, and one output layer. The network may have many hidden layers, but we only look at one, because they are all treated equally, and the update rules are the same.

This diagram must be read from right to left. The output of the right-most neuron is the activation function applied to its input.

The input of the activation function is, as we know, the sum of all the stimuli multiplied by their corresponding weight.

The input of the right-most neuron is the output of a hidden neuron. The process repeats for as many hidden neurons as there are.

The input of the left-most neurons is the input of the whole network.



Forward pass

UNIVERSITY OF LEEDS

Neuron j is a **hidden** neuron

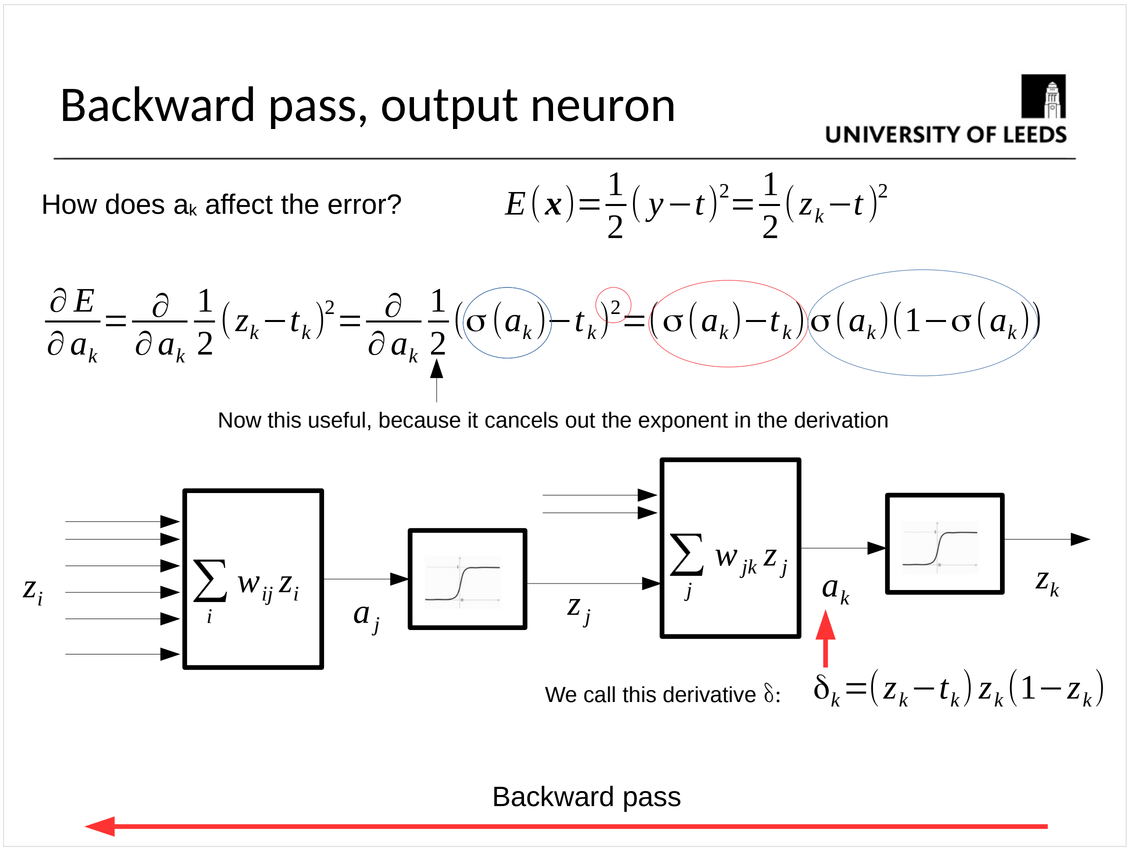
Neuron k is an **output** neuron

$$\ldots \qquad a_j = \sum_{i=1}^{N} w_{ij} z_i \qquad z_j = f(a_j) \qquad a_k = \sum_{j=1}^{M} w_{jk} z_j \qquad z_k = f(a_k)$$

Forward pass: compute all the z

The backpropagation of errors (usually just referred to just as backpropagation, or backprop), is a 2-pass algorithms.

The forward pass consists simply in computing the output of all the neurons, that is, all the “z”s.



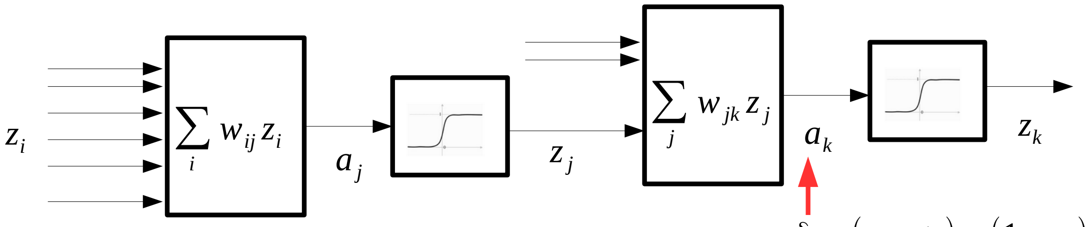
Backward pass, output neuron

UNIVERSITY OF LEEDS

How does a_k affect the error? $E(\boldsymbol{x})=\frac{1}{2}(y-t)^2=\frac{1}{2}(z_k-t)^2$

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

Now this useful, because it cancels out the exponent in the derivation



We call this derivative δ: $\delta_k=(z_k-t_k)z_k(1-z_k)$

Backward pass

In the backward pass, the error is “backpropagated” layer by layer.

We first see what happens to the error as the input of the activation function changes.

To do that, we compute the derivative of the error with respect to the input of the sigmoid, a_k.

The output of the whole network is the output z_k of an output neuron. The output z_k is the sigmoid of its input, and we know how to derive the sigmoid!

Backward pass, output neuron

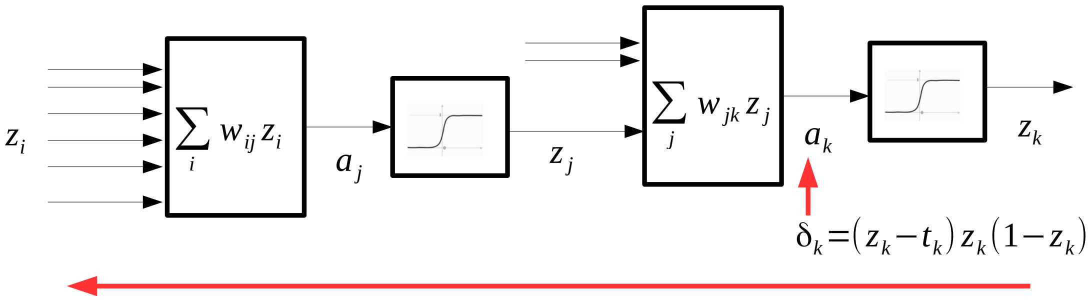
UNIVERSITY OF LEEDS

One step backward, inside the box: how does w_{jk} affect the error?

$$E(\mathbf{x}) = \frac{1}{2}(z_k - t)^2 = \frac{1}{2}(\sigma(a_k) - t)^2 \qquad a_k = \sum_j w_{jk} z_j$$

We apply the chain rule again: $\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{jk}} = \delta_k \cdot ?$

$$\frac{\partial a_k}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} w_{0k} z_0 + w_{1k} z_1 + w_{2k} z_2 + \cdots + w_{jk} z_j = ?$$



If we apply the chain rule to the error, the derivative with respect to one of the weights of an output neuron is the derivative of the output with respect to a_k, multiplied by the derivative of the a_k with respect to the weight.

Since the input of the sigmoid is just the sum of the stimuli multiplied by their weight, the derivative with respect to a single weight is very easy!

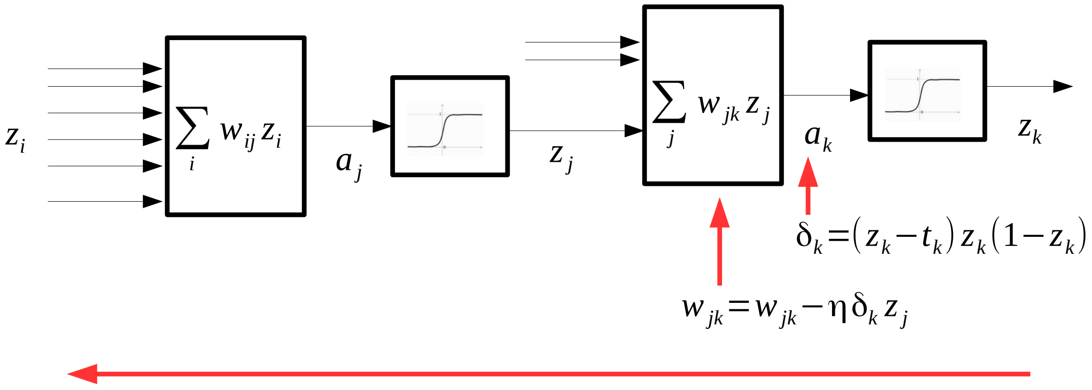
Backward pass, output neuron

UNIVERSITY OF LEEDS

One step backward, inside the box: how does w_{jk} affect the error?

We apply the chain rule again: $\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial w_{jk}} = \delta_k z_j$

$$\frac{\partial a_k}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} w_{0k} z_0 + w_{1k} z_1 + w_{2k} z_2 + \cdots + w_{jk} z_j = z_j$$

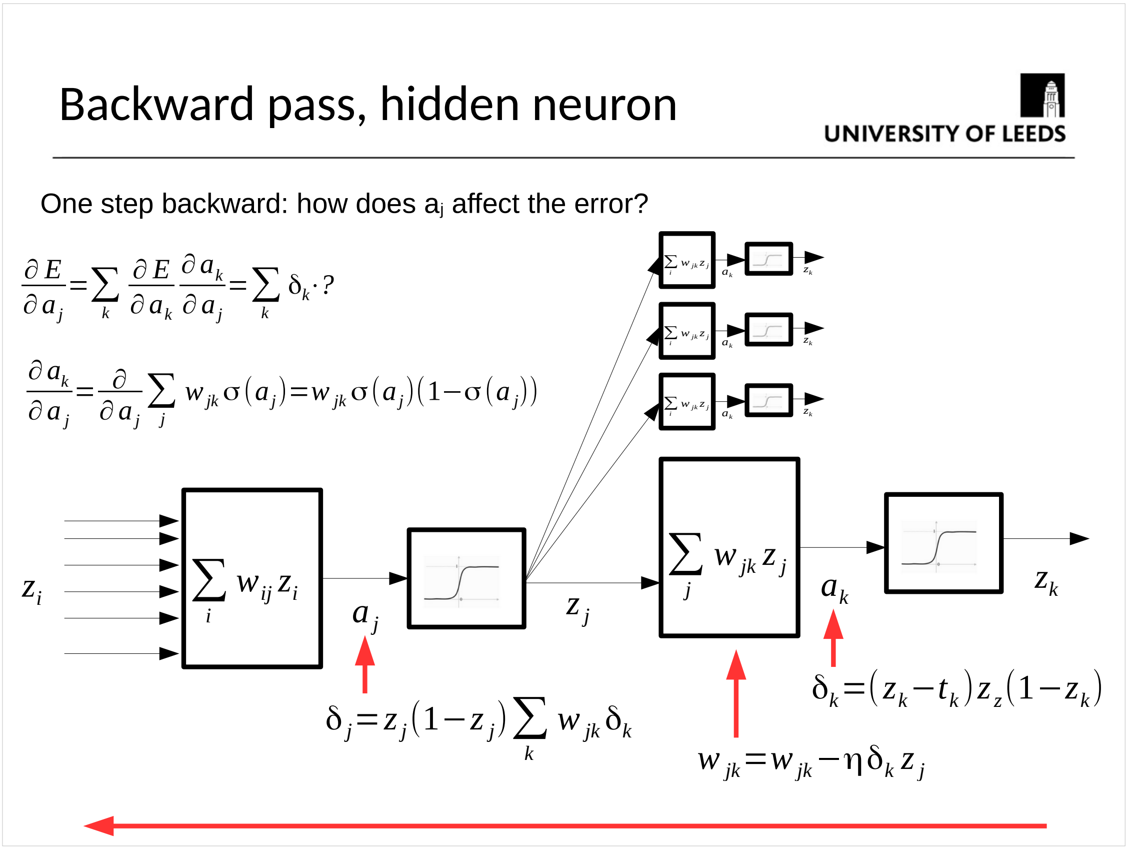


z_i
$\sum_i w_{ij} z_i$
a_j
z_j
$\sum_j w_{jk} z_j$
a_k
z_k
$\delta_k = (z_k - t_k) z_k (1 - z_k)$
$w_{jk} = w_{jk} - \eta \delta_k z_j$

The derivative with respect to a single weight is just the corresponding stimulus (input)!

We have thus computed the gradient with respect to the weights of the output neuron, and we can write the update rule for those weights.

We can now continue towards the weights of the hidden neurons.



Backward pass, hidden neuron

UNIVERSITY OF LEEDS

One step backward: how does a_j affect the error?

$$\frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \cdot ?$$

$$\frac{\partial a_k}{\partial a_j} = \frac{\partial}{\partial a_j}\sum_j w_{jk}\sigma(a_j) = w_{jk}\sigma(a_j)(1-\sigma(a_j))$$

$$\delta_j = z_j(1-z_j)\sum_k w_{jk}\delta_k$$

$$\delta_k = (z_k - t_k) z_z (1 - z_k)$$

$$w_{jk} = w_{jk} - \eta \delta_k z_j$$

The hidden neurons affect the error through the output neurons they are connected to.

Like before, we start by finding the derivative of the error with respect to the input of the sigmoid, a_j.

This can be done by decomposing, thanks to the chain rule, the error in the derivative of the error with respect to a_k (which we computed before, and called δ_k) and the derivative of a_k with respect to a_j.

The derivative of a_k with respect to a_j depends on the outputs of the hidden neuron, which are sigmoids. Luckily we know how to derive a sigmoid!

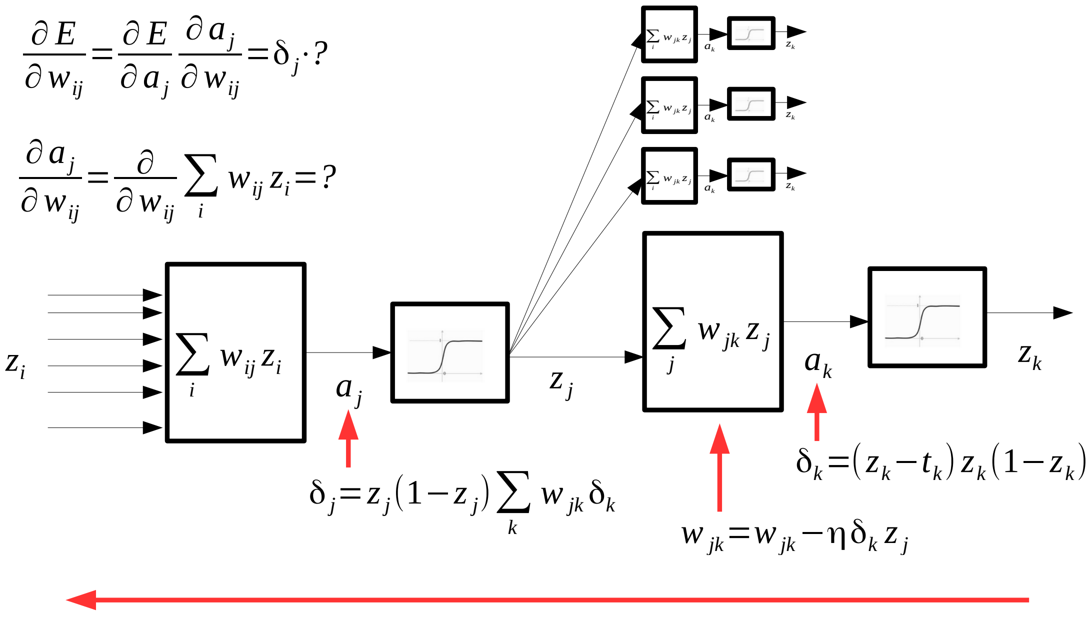
Computing delta

UNIVERSITY OF LEEDS

One step backward, inside the box: how does w_{ij} affect the error?

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} = \delta_j \cdot ?$$

$$\frac{\partial a_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_i w_{ij} z_i = ?$$


z_i
$\sum_i w_{ij} z_i$
a_j
z_j
$\sum_j w_{jk} z_j$
a_k
z_k
$\delta_j = z_j(1-z_j)\sum_k w_{jk}\delta_k$
$w_{jk} = w_{jk} - \eta \delta_k z_j$
$\delta_k = (z_k - t_k) z_k (1-z_k)$

Lastly, we derive the input of the sigmoid, a_j, with respect to the weights connected to it.

Since a_j is just a linear function, this derivative is easy!

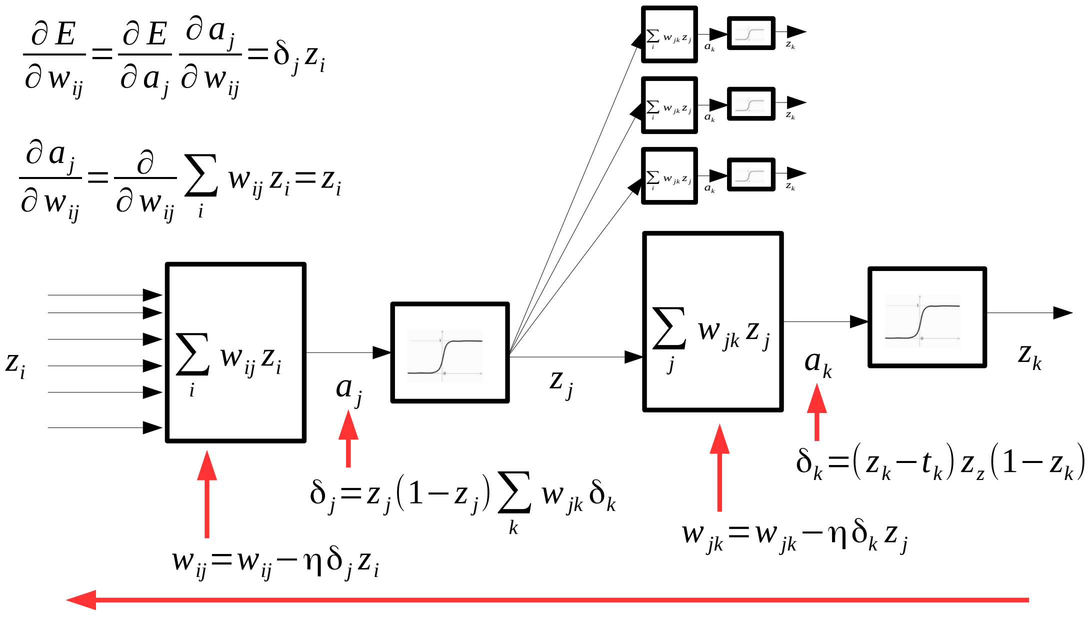
Computing delta

UNIVERSITY OF LEEDS

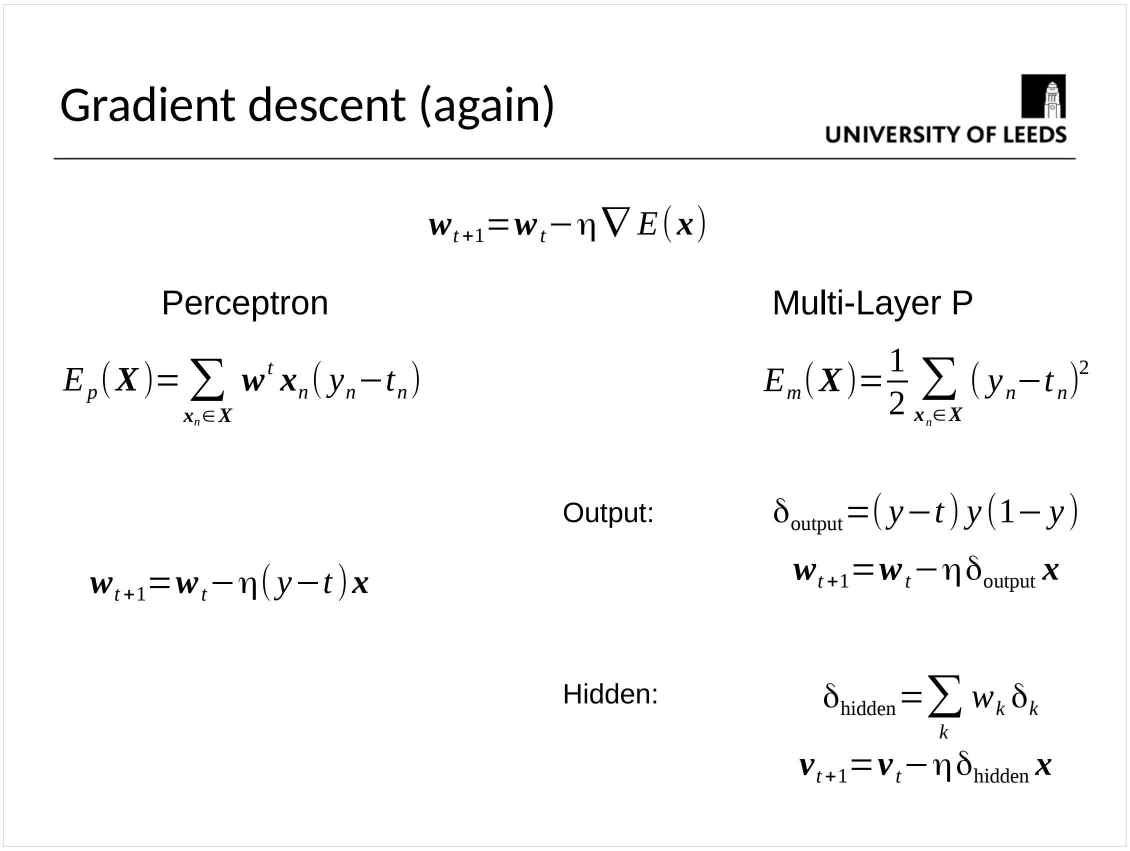
One step backward, inside the box: how does w_{ij} affect the error?

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} = \delta_j z_i$$

$$\frac{\partial a_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\sum_i w_{ij} z_i = z_i$$


z_i
$\sum_i w_{ij} z_i$
a_j
z_j
$\sum_j w_{jk} z_j$
a_k
z_k
$\delta_k = (z_k - t_k) z_z (1 - z_k)$
$\delta_j = z_j (1 - z_j) \sum_k w_{jk} \delta_k$
$w_{jk} = w_{jk} - \eta \delta_k z_j$
$w_{ij} = w_{ij} - \eta \delta_j z_i$

With the last derivative, we have completed the gradient with respect to the weights in a hidden neuron, and we have the corresponding update rule.



Gradient descent (again)

UNIVERSITY OF LEEDS

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla E(\boldsymbol{x})$$

Perceptron

$$E_p(\boldsymbol{X}) = \sum_{\boldsymbol{x}_n \in \boldsymbol{X}} \boldsymbol{w}^t \boldsymbol{x}_n (y_n - t_n)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta (y - t) \boldsymbol{x}$$

Multi-Layer P

$$E_m(\boldsymbol{X}) = \frac{1}{2} \sum_{\boldsymbol{x}_n \in \boldsymbol{X}} (y_n - t_n)^2$$

Output:

$$\delta_{\text{output}} = (y - t)\, y (1 - y)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \delta_{\text{output}} \boldsymbol{x}$$

Hidden:

$$\delta_{\text{hidden}} = \sum_k w_k \delta_k$$

$$\boldsymbol{v}_{t+1} = \boldsymbol{v}_t - \eta \delta_{\text{hidden}} \boldsymbol{x}$$

We set out to find the update rule for the neural network, by minising the MSE.

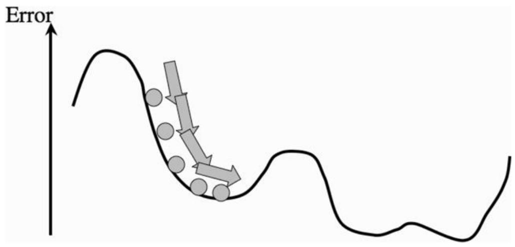
In this notation I focused on the single neurons, so analogously to the perceptron, I represented with y the output of the neuron, and with x its input (even if it comes from a previous neuron). I called w the weights of the output neurons, and v the weights of the hidden neurons.

We can see that the structure of the update rule is similar to the update rule of the perceptron. Both have one component of the gradient, (y-t) in the perceptron and the δ for the MLP , which depends only on the **output**, and one component which is directly the **input**.

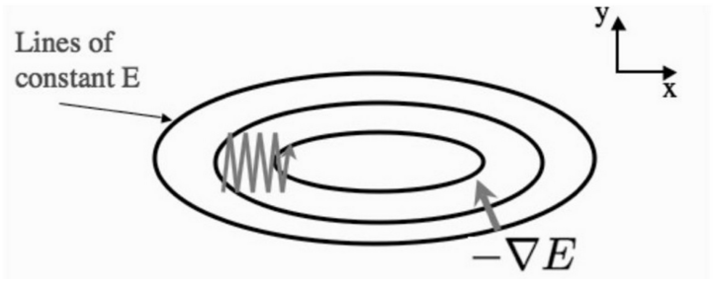
The delta of the hidden neurons depends on the deltas of the neurons they are connected to, forming a chain that depends on the structures of the network.

Local Minima

UNIVERSITY OF LEEDS



Start with weights close to 0: where the decision is actually made



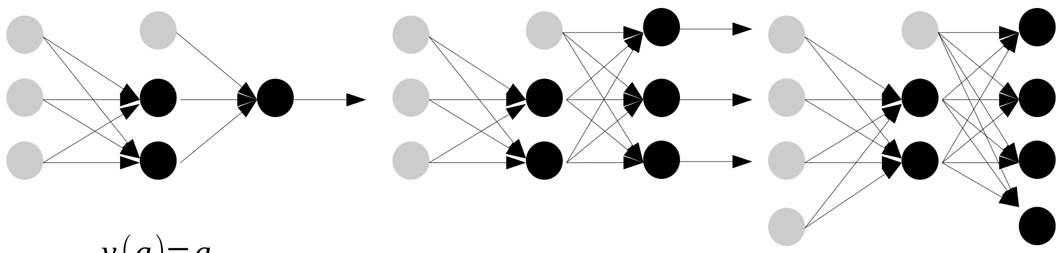
Multiple random restarts

Remember that gradient descent is a local method, so to find different solutions we need to start multiple times from different initial weights.

Using MLPs

UNIVERSITY OF LEEDS

Regression | Classification | Compression (Autoencoder)



$y(a)=a$

Last neuron linear

One output per class, pick highest

Middle “bottleneck” layer

MLPs can be used in different ways. Here we mention three:

For regression, that is, to approximate any function, with a single output, where the output neuron has the identity activation function (otherwise, if it had a sigmoid, the function would be limited to [0,1]).

For classification, where the MLP has one output neuron per class, and we consider the input to be classified into the class with the highest output.

For compression, where the inputs are presented as labels at the output, and the MLP learns to reproduce the inputs. This is an unsupervised learning technique (there are no labels) and the hidden layers learn a “compressed” representation of the input.

Training "recipe"

UNIVERSITY OF LEEDS

Choose features

Normalize (rescale) data: $x' = \frac{x - \bar{x}}{\sigma}$ or $x' = \frac{x - min(x)}{max(x) - min(x)}$

Zero mean, unit variance

in [0,1]

Create training, validation, and test sets

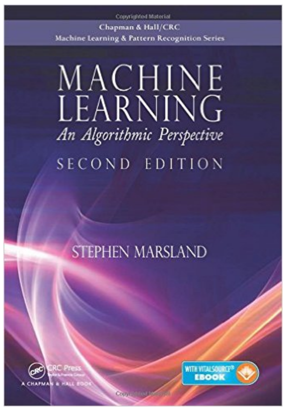
Decide whether you need hidden layers and how big. Try several ones.

Train

Test

UNIVERSITY OF LEEDS

Conclusion


Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series
MACHINE LEARNING
An Algorithmic Perspective
SECOND EDITION
STEPHEN MARSLAND
CRC Press
WITH VITALSOURCE® EBOOK

Chapter 4