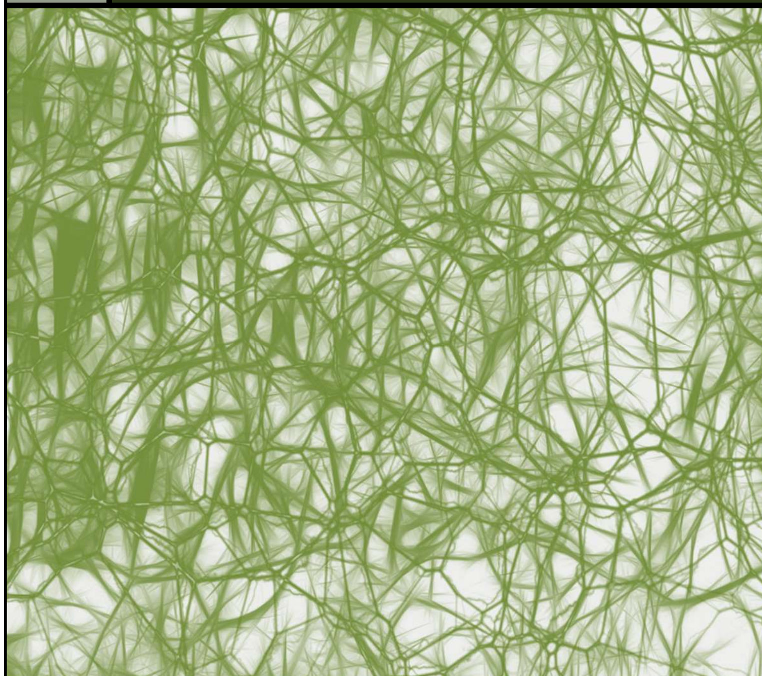
Department of Computer Science
University of Bristol

COMSM0045 – Applied Deep Learning 2020/21

comsm0045-applied-deep-learning.github.io



Lecture 03

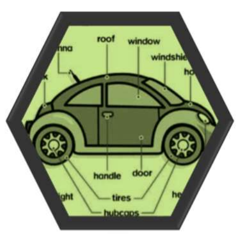
BACKPROPAGATION ALGORITHM

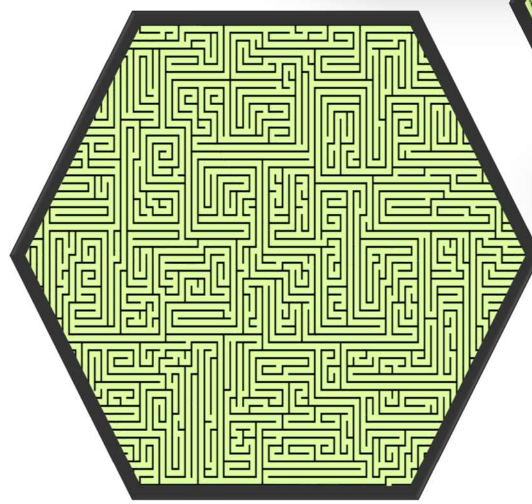
Tilo Burghardt | tilo@cs.bris.ac.uk

24 Slides

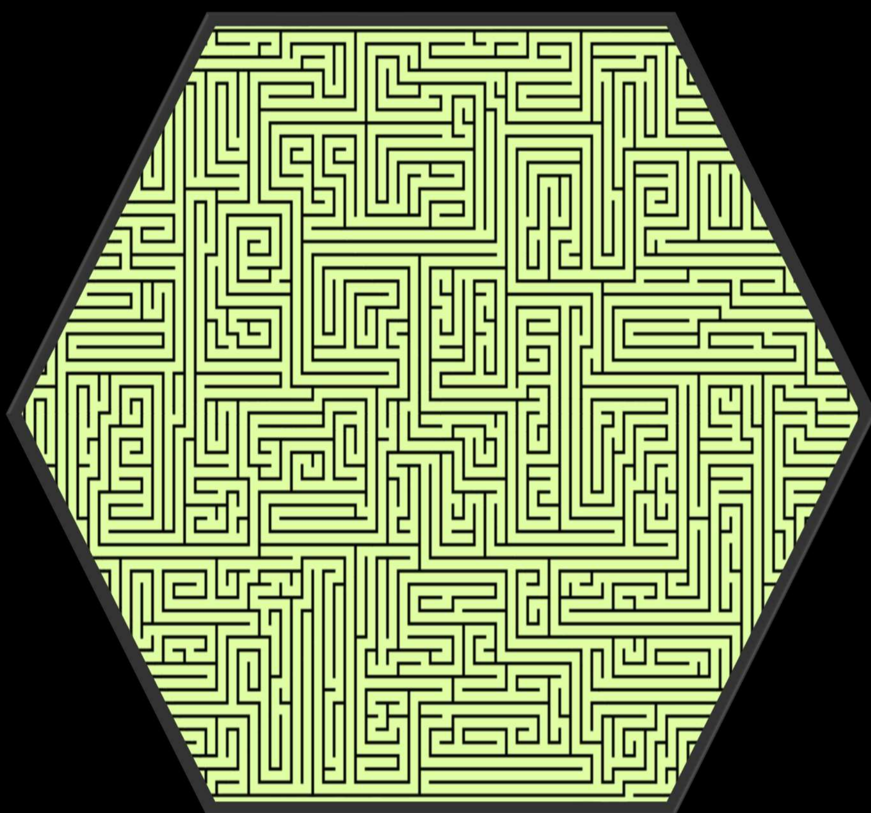
Agenda for Lecture 3

- Recap Auto-Differentiation
- Backpropagation Algorithm
- Activation Functions





Recap: Reverse Auto-Differentiation in Computational Graphs



A General Strategy: Chain Rule and Summing over all Paths

$$a = b * c$$
$$b = d + e$$
$$c = e + 2$$
$$d = 3 + f$$
$$e = f * g$$

Analytical Solution:

$$a = (d+e)(e+2) = de + 2d + e^2 + 2e$$
$$= (3+f)fg + 2(3+f) + (fg)^2 + 2fg$$
$$= 3fg + f^2 g + 6 + 2f + f^2 g^2 + 2fg$$
$$= f^2(g^2 + g) + 5fg + 2f + 6$$
$$\frac{\partial a}{\partial f} = 2fg^2 + 2fg + 5g + 2$$

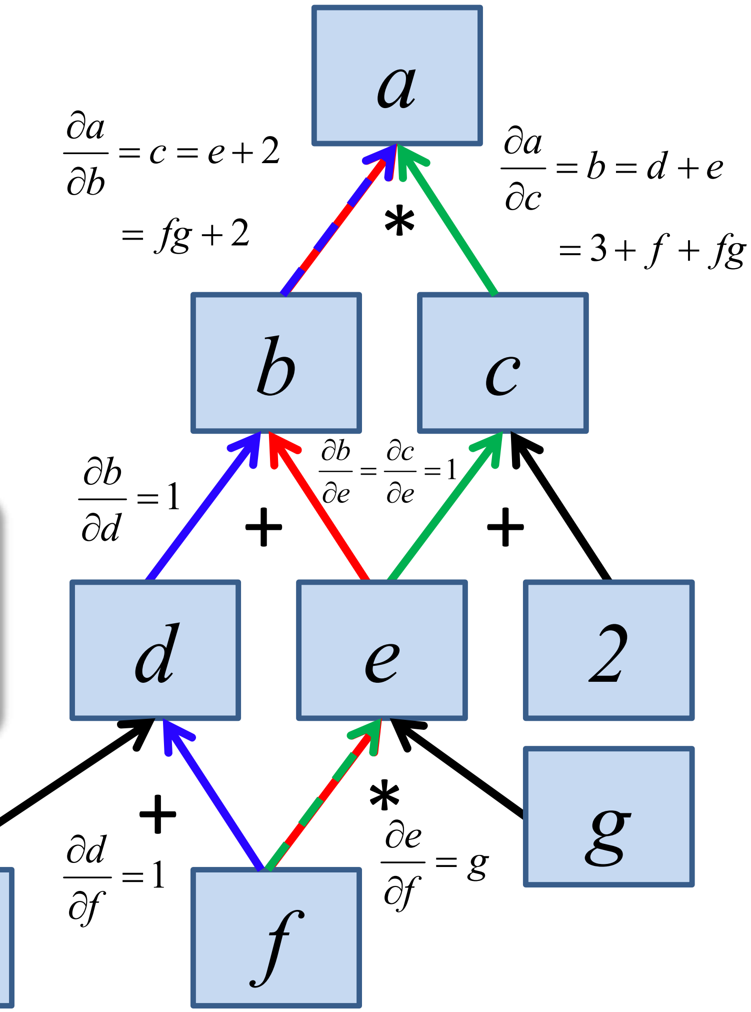
General Approach based on Network Layout:

$$\frac{\partial a}{\partial f} = \underbrace{\frac{\partial a}{\partial b}\frac{\partial b}{\partial e}\frac{\partial e}{\partial f}}_{path1} + \underbrace{\frac{\partial a}{\partial c}\frac{\partial c}{\partial e}\frac{\partial e}{\partial f}}_{path2} + \underbrace{\frac{\partial a}{\partial b}\frac{\partial b}{\partial d}\frac{\partial d}{\partial f}}_{path3}$$

apply chain rule along path

sum over all paths that connect f to a

$$= (fg+2) + (fg+2)g + (3+f+fg)g$$
$$= fg + 2 + fg^2 + 2g + 3g + fg + fg^2$$
$$= 2fg^2 + 2fg + 5g + 2$$



Observation of Hierarchical Dependency

Global Structure used so far:

$$\frac{\partial a}{\partial f} = \underbrace{\frac{\partial a}{\partial b}\frac{\partial b}{\partial e}\frac{\partial e}{\partial f}}_{path1} + \underbrace{\frac{\partial a}{\partial c}\frac{\partial c}{\partial e}\frac{\partial e}{\partial f}}_{path2} + \underbrace{\frac{\partial a}{\partial b}\frac{\partial b}{\partial d}\frac{\partial d}{\partial f}}_{path3}$$

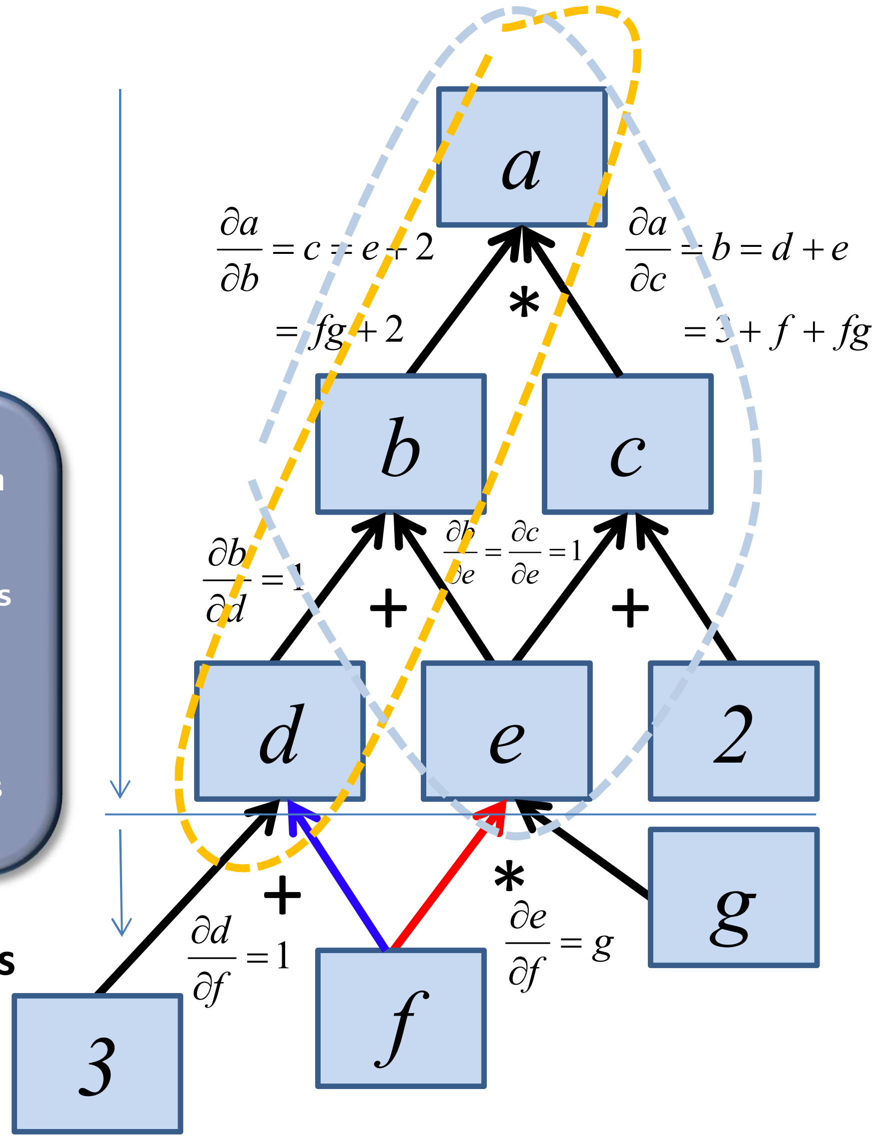
Hierarchical Structure:

$$\frac{\partial a}{\partial f} = \frac{\partial a}{\partial e}\frac{\partial e}{\partial f} + \frac{\partial a}{\partial d}\frac{\partial d}{\partial f}$$

$$\frac{\partial a}{\partial e} = \frac{\partial a}{\partial b}\frac{\partial b}{\partial e} + \frac{\partial a}{\partial c}\frac{\partial c}{\partial e}$$

$$\frac{\partial a}{\partial d} = \frac{\partial a}{\partial b}\frac{\partial b}{\partial d} \quad \ldots$$

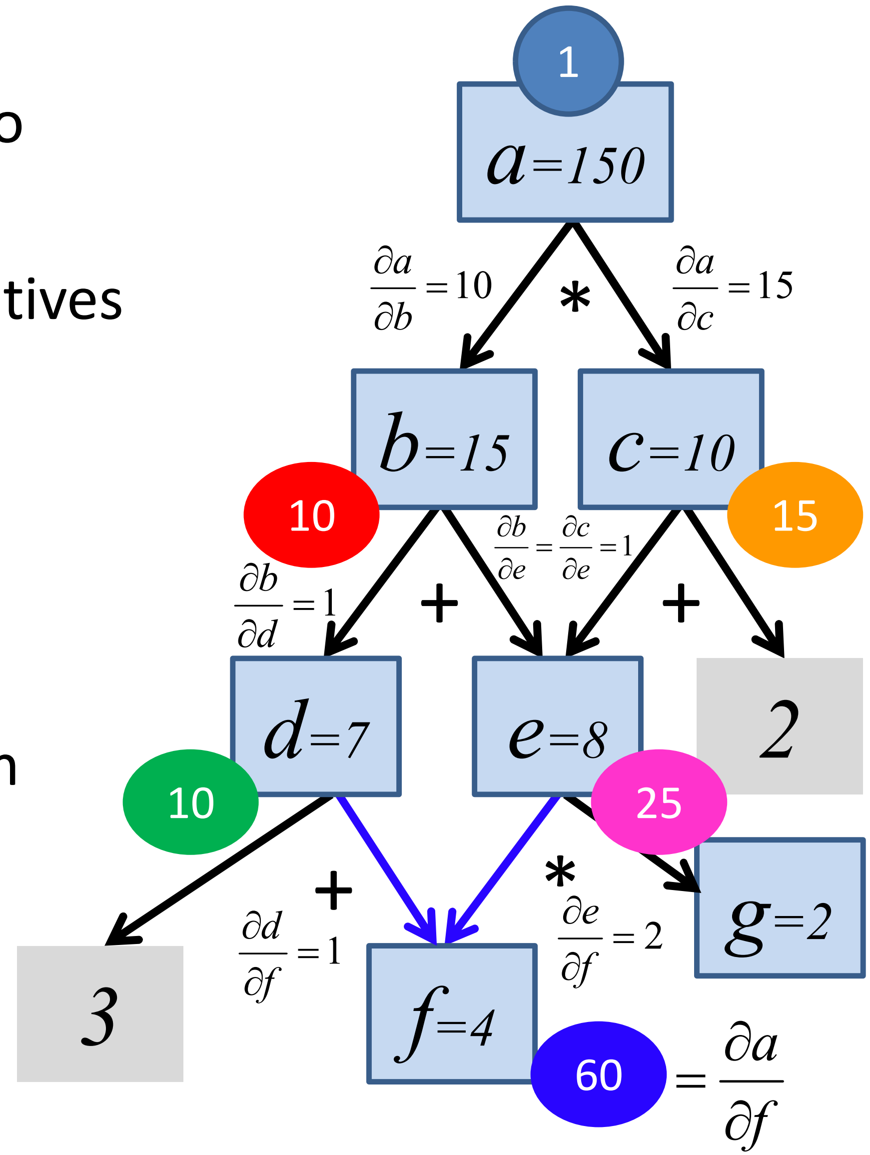
We observe that, to calculate results from the layer above, for each node we can sum over all incoming edges from the layer above and multiply each by the result we have obtained in the node that the edge connects to in the layer above.



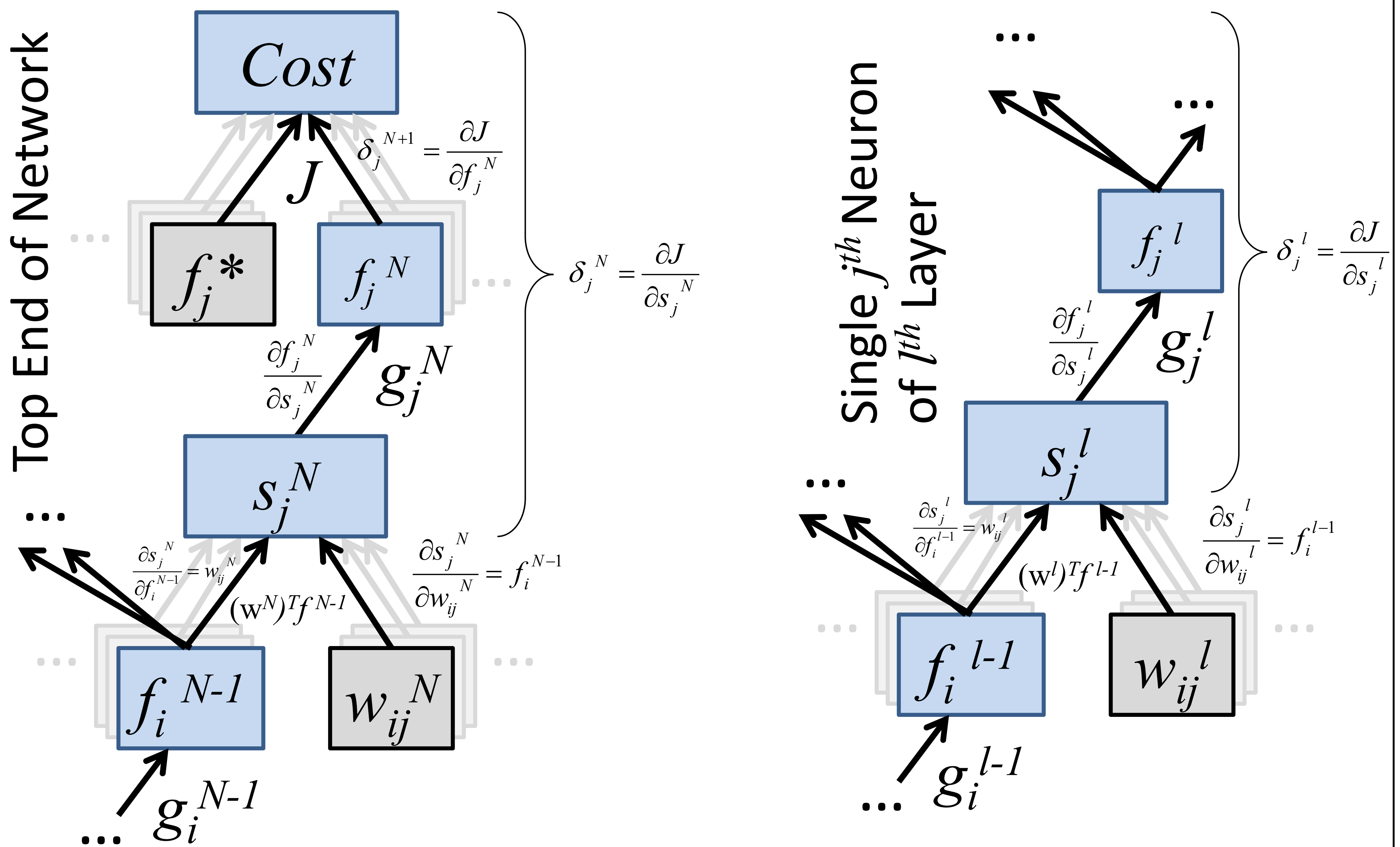
→ once you know all (part-evaluated) derivatives associated to a layer above, summation of them from connected nodes times local derivatives is sufficient to get the next layer of derivatives

Summary of Reverse Auto-Differentiation

- **Two-pass Strategy**
  - forward pass to give values to nodes and output
  - backward pass to establish deltas δ, i.e. **<u>all</u>** partial derivatives
- **Requirements**
  - feed-forward network
  - local per-edge derivatives must be known
- **Solution Tactic**
  - instead of explicit summation over all paths, layer-by-layer evaluation via summation over all incoming local derivatives times their associated deltas



Deep Neural Networks as Special Computational Graphs



The Backpropagation Algorithm

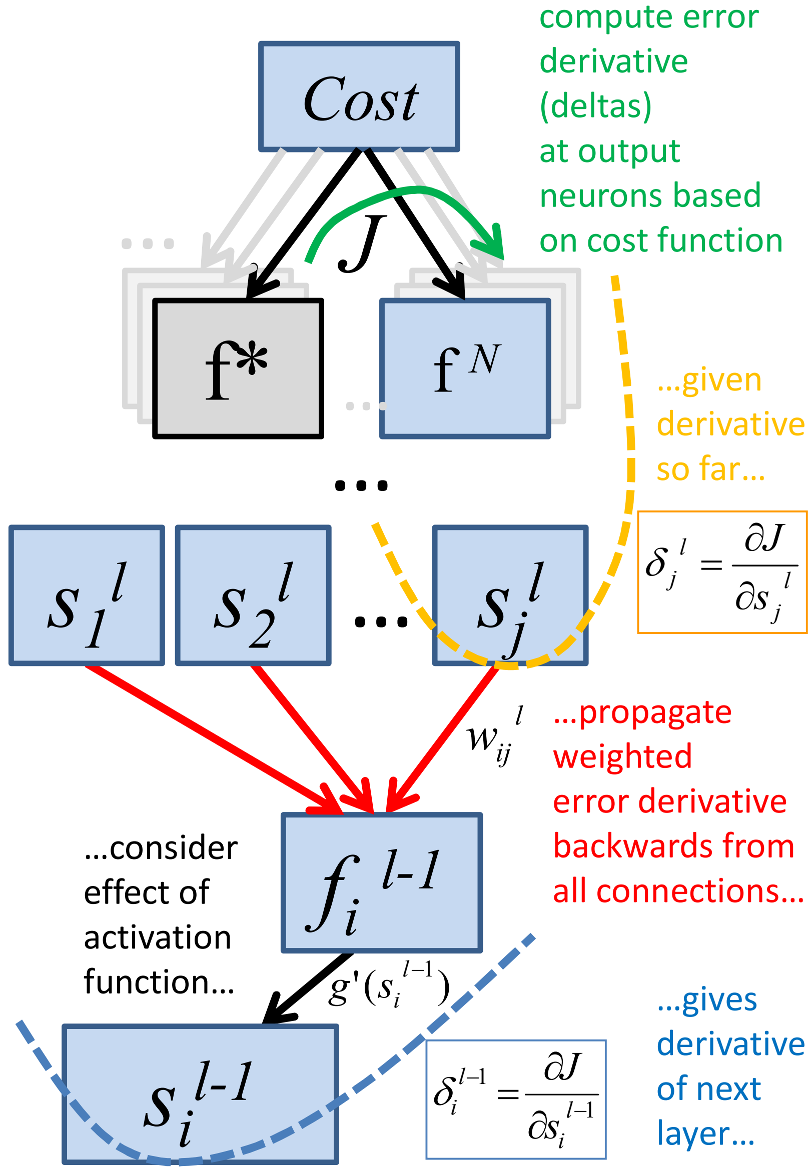
Idea of Backpropagation for Network Training

- **General Concept:**
  combine *reverse auto-differentiation* (for finding relationship of cost function to each weight) with *gradient descent* (for stepwise adjustment of weights)

- **Intuition behind 'Error Derivative Propagation':**
  compute discrepancy between network output and target; then propagate this discrepancy backwards through the network adjusted by the influence of the paths travelled in order to determine the influence of each and every weight (ends of paths) towards the discrepancy

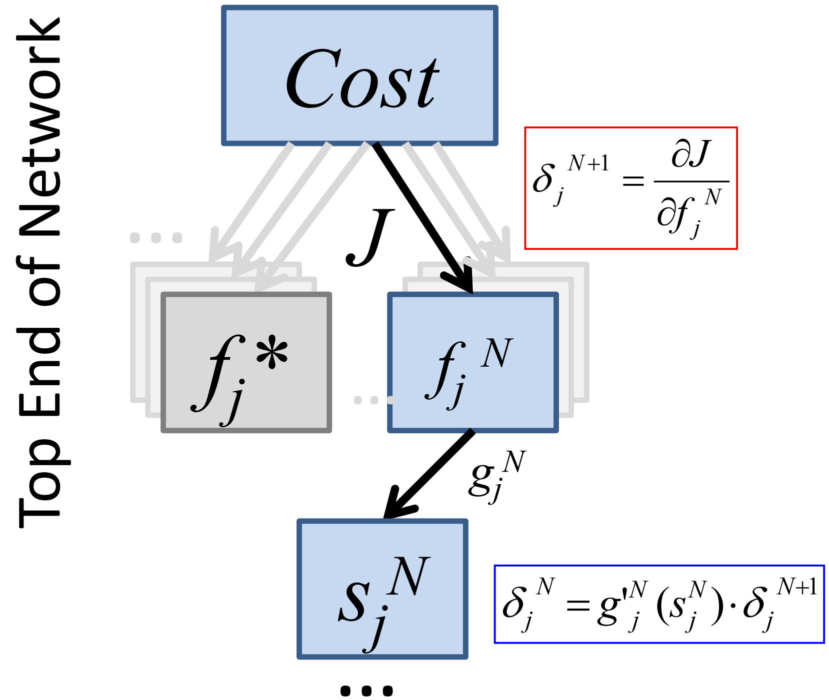
Overall Strategy behind Backpropagation

- First, calculate neuron activities for all layers and cost in a forward pass given the input.
- At the top of the network, convert the discrepancy between outputs and targets according to the cost function into error derivatives linked to the final layer output (the topmost deltas).
- Then, layer by layer, calculate error derivatives for neurons in hidden layers by combining all connected error derivatives in the layer above and considering the effect of activation functions – thereby, propagating error derivatives backwards.
- Use neuron activities to get error derivatives w.r.t. the weights.



The Backpropagation Concept: Step 1

- First, calculate all s_j^l and f_j^l in a single forward pass.
- At the top of the network, convert the discrepancy between outputs f^N and targets f^* into error derivatives δ_j^{N+1} linked to all final layer neurons j according to J, and compute δ_j^N.
- Next, layer by layer, calculate all error derivatives δ_i^{l-1} in each hidden layer from all error derivatives δ_j^l in the layer above.
- Use these error derivatives δ_j^l w.r.t. activities f_i^{l-1} to get error derivatives w.r.t. the weights.



$$e.g. \quad J = \frac{1}{2}\sum_j (f_j^N - f_j^*)^2$$

$$\delta_j^{N+1} = \frac{\partial J}{\partial f_j^N} = f_j^N - f_j^*$$

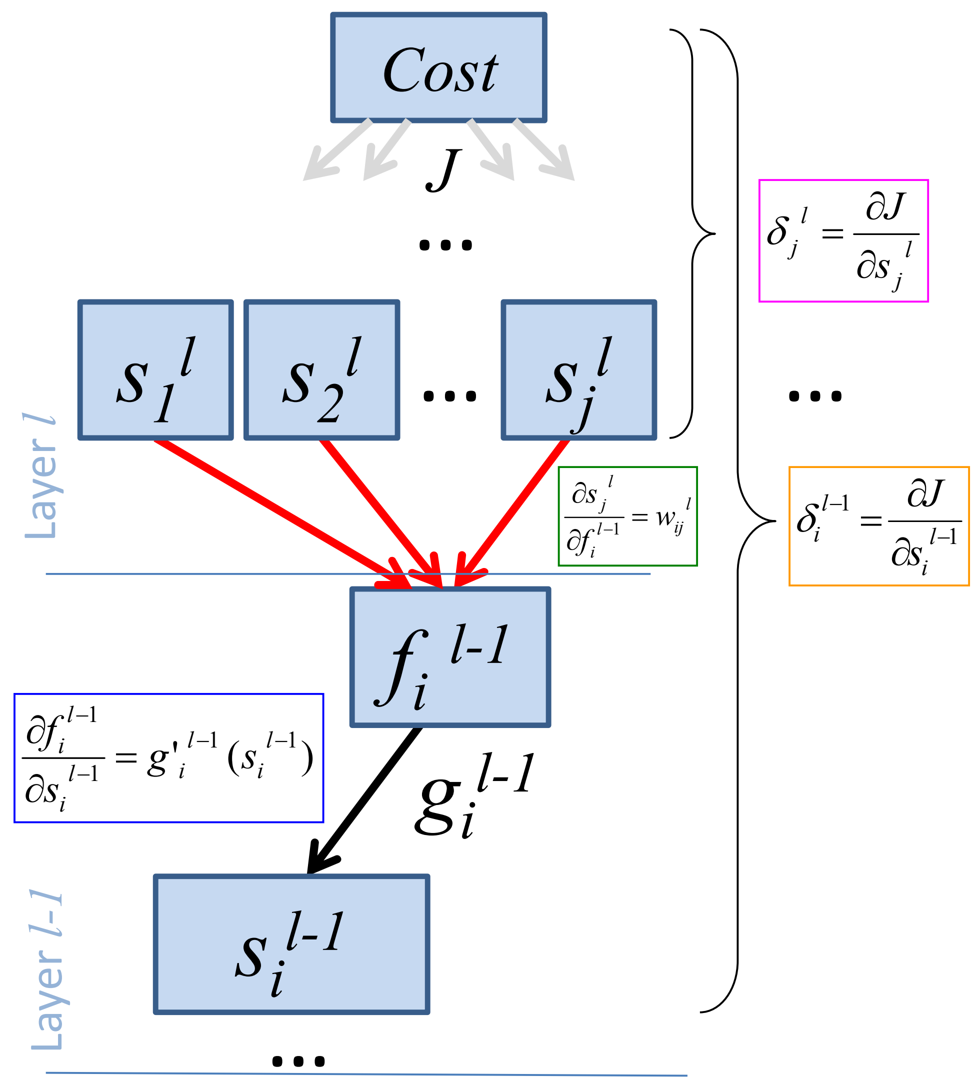
Backpropagation of Error Derivatives between Layers

$$\delta_i^{l-1} = \frac{\partial J}{\partial s_i^{l-1}}$$

$$= \sum_{j=1}^{d(l)} \underbrace{\frac{\partial J}{\partial s_j^l}}_{\delta_j^l} \underbrace{\frac{\partial s_j^l}{\partial f_i^{l-1}}}_{w_{ij}^l} \underbrace{\frac{\partial f_i^{l-1}}{\partial s_i^{l-1}}}_{g'^{l-1}_i(s_i^{l-1})}$$

$$= \sum_{j=1}^{d(l)} \delta_j^l w_{ij}^l g'^{l-1}_i(s_i^{l-1})$$

$$= g'^{l-1}_i(s_i^{l-1}) \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l$$

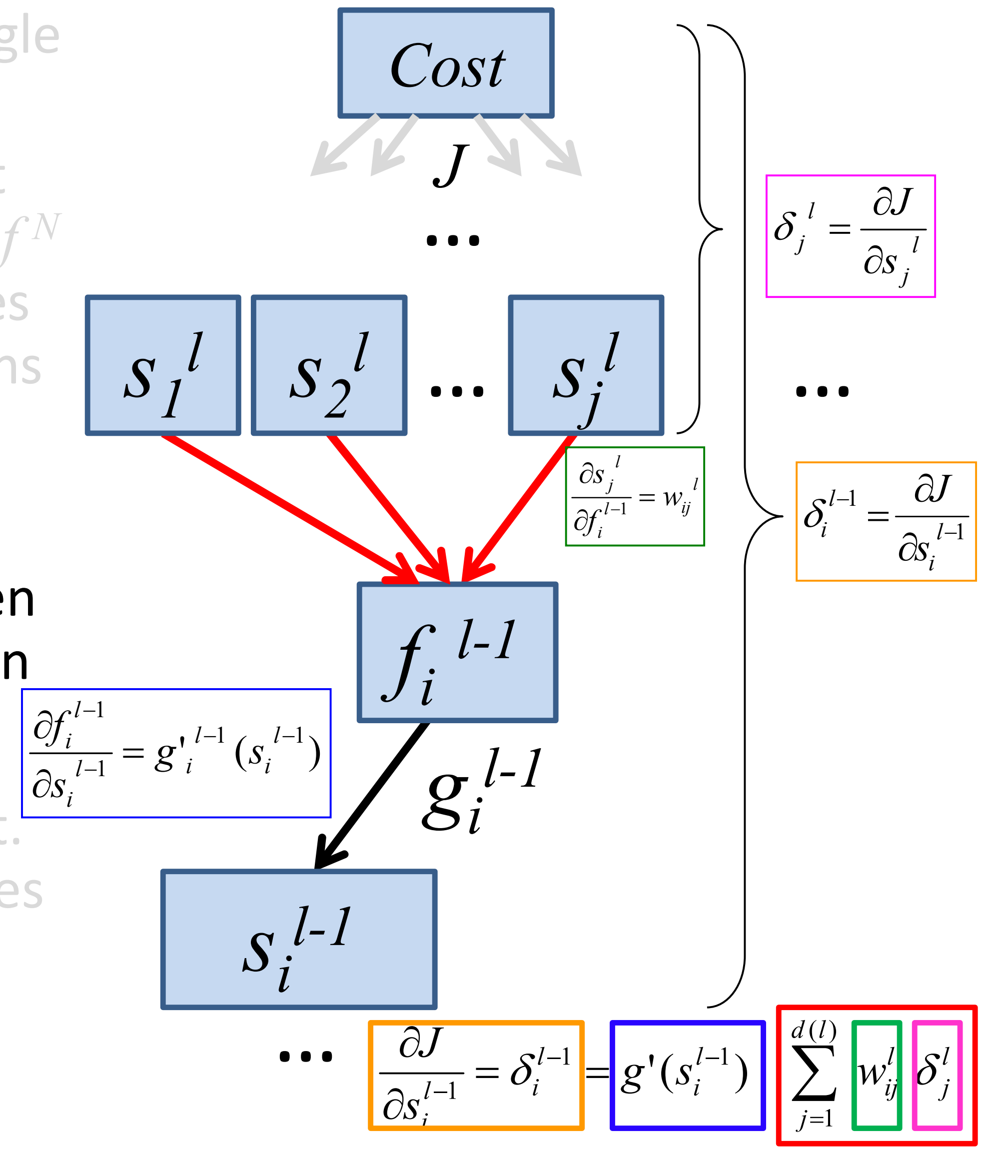


The Backpropagation Concept: Step 2

- First, calculate all s_j^l and f_j^l in a single forward pass.
- At the top of the network, convert the discrepancy between outputs f^N and targets f^* into error derivatives δ_j^{N+1} linked to all final layer neurons j according to J, and compute δ_j^N.

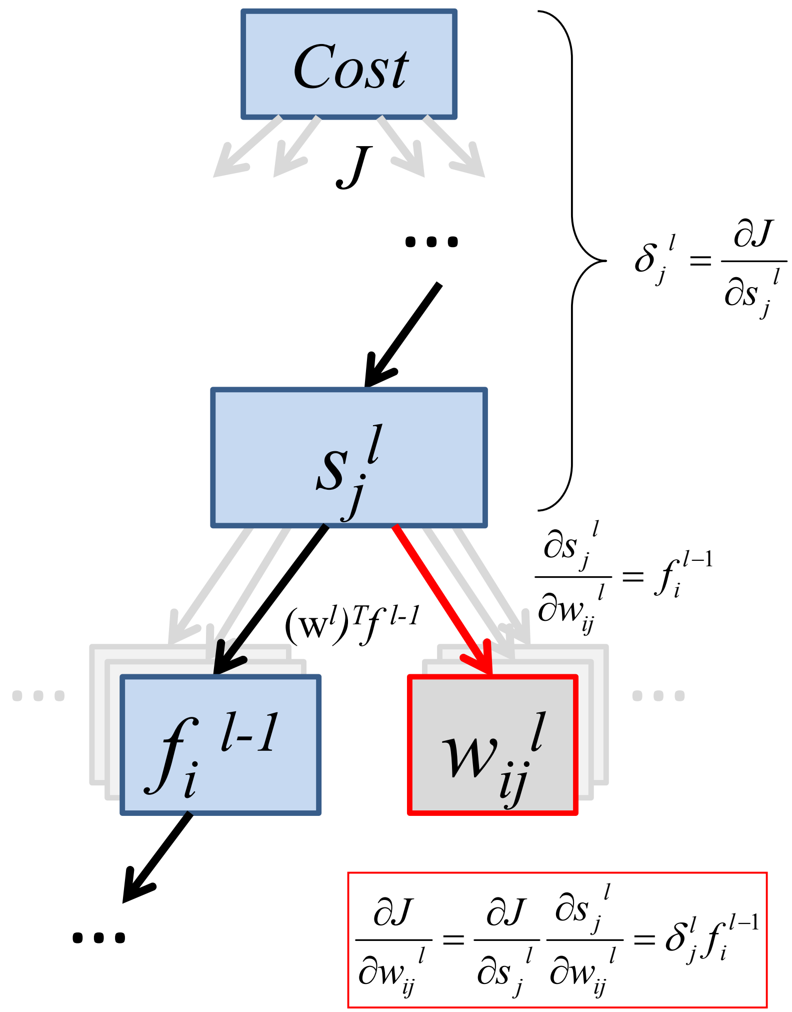
- Next, layer by layer, calculate all error derivatives δ_i^{l-1} in each hidden layer from all error derivatives δ_j^l in the layer above.

- Use these error derivatives δ_j^l w.r.t. activities f_i^{l-1} to get error derivatives w.r.t. the weights.



The Backpropagation Concept: Step 3

- First, calculate all s_j^l and f_j^l in a single forward pass.
- At the top of the network, convert the discrepancy between outputs f^N and targets f^* into error derivatives δ_j^{N+1} linked to all final layer neurons j according to J, and compute δ_j^N.
- Next, layer by layer, calculate all error derivatives δ_i^{l-1} in each hidden layer from all error derivatives δ_j^l in the layer above.
- Use these error derivatives δ_j^l w.r.t. activities f_i^{l-1} to get error derivatives w.r.t. the weights.



Backpropagation Algorithm (Sketch)

initialise all weights randomly

for $t=0, 1, 2, \ldots$ **do**

- **pick** next training sample
- **FORWARD PASS:** compute all layer outputs
- **compute** derivative of cost function w.r.t. final layer
- **BACKWARD PASS:** compute all deltas
- **update** all weights based on deltas and activities
- **check** if stopping criteria are met to break loop

return final weights

Backpropagation Algorithm (Details)

initialise all weights w_{ij}^{l} randomly

for $t=0, 1, 2, \ldots$ **do**

pick next training sample $([f_1^0, f_2^0, \ldots], [f_1^*, f_2^*, \ldots])$

FORWARD PASS: compute all $s_j^l = \sum_{i=1}^{d(l-1)} w_{ij}^l f_i^{l-1}$ and $f_j^l = g_j^l(s_j^l)$

compute top deltas $\delta_j^N = g'^N_j(s_j^N) \cdot \partial J / \partial f_j^N$

BACKWARD PASS: compute all $\delta_i^{l-1} = g'^{l-1}_i(s_i^{l-1}) \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l$

update weights $w_{ij}^l \leftarrow w_{ij}^l - \eta\, f_i^{l-1} \delta_j^l$

check if stopping criteria are met to break loop

return final weights w_{ij}^l

Right, can we train deep networks now? – Not quite...

- Backpropagation has been known since 1970s
- What stood in the way of training deep nets effectively?
  - There is a fundamental issue of gradient instability when training truly deep architectures: originally known as the vanishing gradient problem since 1990s.
  - We need fast, differentiable and meaningful robust neuron layouts that address this issue (e.g. ReLU, LSTM).
  - Descent-based optimisation techniques need to work accurately and *fast in practice* despite large training data sets (GPU parallelisation and improved optimisers help a lot here).
  - Number of parameters explode in deep networks; we may need to share them or reuse the entire net (e.g. CNNs/RNNs).
  - Regularisation techniques are critical to achieve good generalisation beyond the training data available!
- It took until the late 2000s to address these arising issues adequately and make deep learning work well in practice...

Activation Functions

Requirement for Differentiable, Non-Linear Activation Functions

initialise all weights w_{ij}^{l} randomly

for $t=0, 1, 2, \ldots$ **do**

pick next training sample $([f_1^0, f_2^0, \ldots], [f_1^*, f_2^*, \ldots])$

FORWARD PASS: compute all $s_j^l = \sum_{i=1}^{d(l-1)} w_{ij}^l f_i^{l-1}$ and $f_j^l = g_j^l(s_j^l)$

compute top deltas $\delta_j^N = g'^N_j(s_j^N) \cdot \partial J / \partial f_j^N$

BACKWARD PASS: compute all $\delta_i^{l-1} = g'^{l-1}_i(s_i^{l-1}) \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l$

update weights $w_{ij}^l \leftarrow w_{ij}^l - \eta \, f_i^{l-1} \delta_j^l$

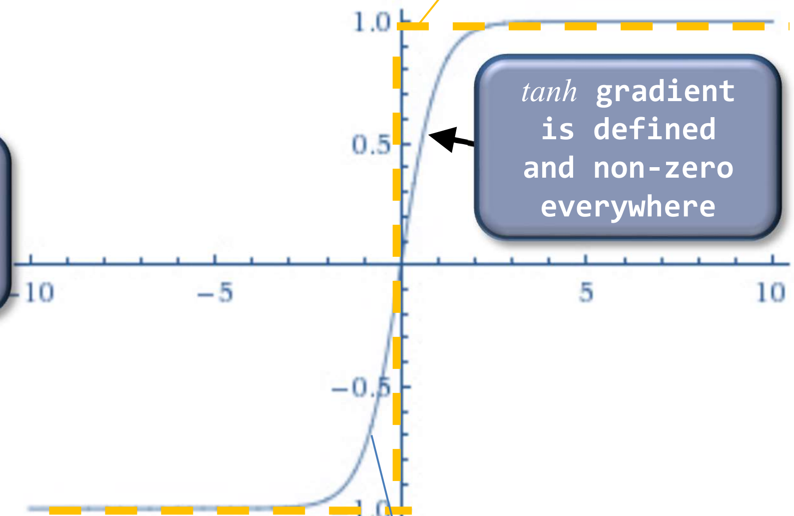
check if stopping criteria are met to break loop

return final weights w_{ij}^l

we require a differentiable non-linearity

$$g_{step}(s) = \begin{cases} 1 & if \quad s \geq 0 \\ -1 & otherwise \end{cases}$$

$$g'_{step}(s) = \begin{cases} 0 & if \quad s \neq 0 \\ ? & otherwise \end{cases}$$



tanh gradient is defined and non-zero everywhere

- **First Idea:** replace step function with *tanh* to provide a fully differentiable, similarly structured alternative
- However, what happens to the gradient at the tail ends of *tanh*?

appealingly simple derivative using squared output

$$g_{\tanh}(s) = \frac{2}{1+e^{-2s}} - 1$$

$$g'_{\tanh}(s) = 1 - g_{\tanh}^{\,2}$$

The 'Vanishing Gradient' Problem

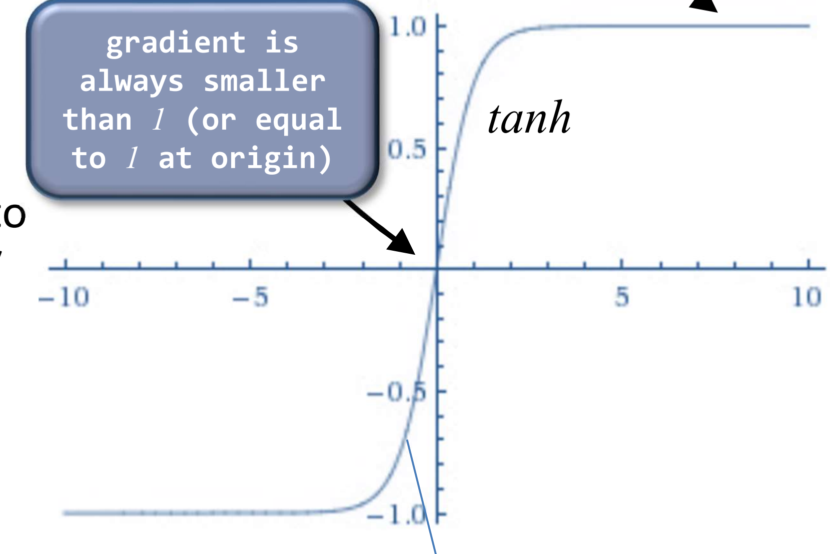
Example: the further we propagate the error derivative backwards (e.g. *l* being small), the more often we multiply δ with a very small number *tanh'<1*, potentially making learning extremely slow or suppress it completely in early layers.

$$\delta_i^{l-1} = g'^{l-1}_i \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l = g'^{l-1}_i \sum_{j=1}^{d(l)} w_{ij}^l \left(g'^{l}_j \sum_{k=1}^{d(l+1)} w_{jk}^{l+1} (\ldots) \right)$$

$g'\ appears\ (N-l+1)\ times\ as\ factor$

gradient becomes very small when the input is saturating the neuron (either very high positive or negative input)

gradient is always smaller than *1* (or equal to *1* at origin)



$$g_{\tanh}(s) = \frac{2}{1+e^{-2s}} - 1$$

$$g'_{\tanh}(s) = 1 - g_{\tanh}^2$$

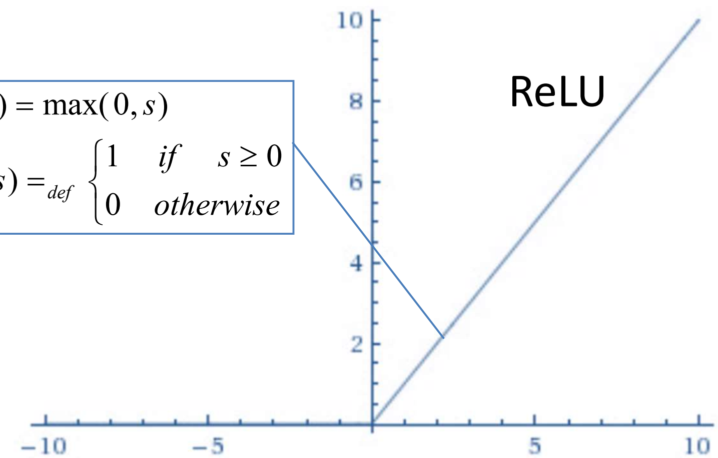
- **Problem:** *tanh* becomes close to 0 when 'saturated' – this causes early layers in particular to learn much slower if at all (since the gradient may vanish exponentially)
- first explained by Sepp Hochreiter in 1990s
- **Some helpful measures may include:**
  - hierarchical pre-training of shallow networks
  - extensively slow+long training on all data helps
  - propagate alternatives to gradient (e.g. sign of gradient)
  - forward signal via residual neural networks (ResNet)
  - other, specially robust neuron layouts

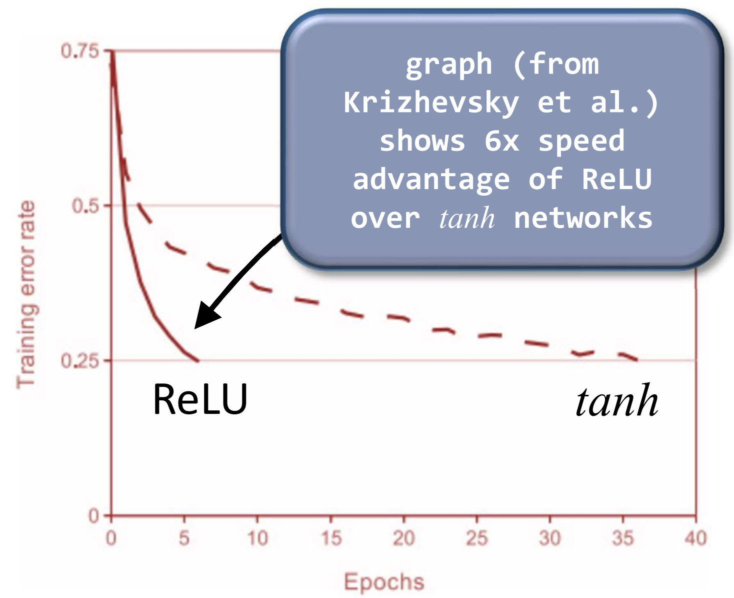
Rectifying Linear Unit (ReLU)

- **Second Idea:** ReLU combines high speed of evaluation with a non-saturating non-linearity
- combined effect may yield practically 5-10 times faster convergence of a network
- however, it introduces a new problem a.k.a. **'Dying Neurons'**: a large gradient flowing through a ReLU unit may force the neuron to never activate again (with the incoming signal always averaging under zero)
- thus, a network may end up carrying a lot of dead units that will not contribute to learning anymore

$$g_{ReLU}(s) = \max(0, s)$$

$$g'_{ReLU}(s) =_{def} \begin{cases} 1 & if \quad s \geq 0 \\ 0 & otherwise \end{cases}$$





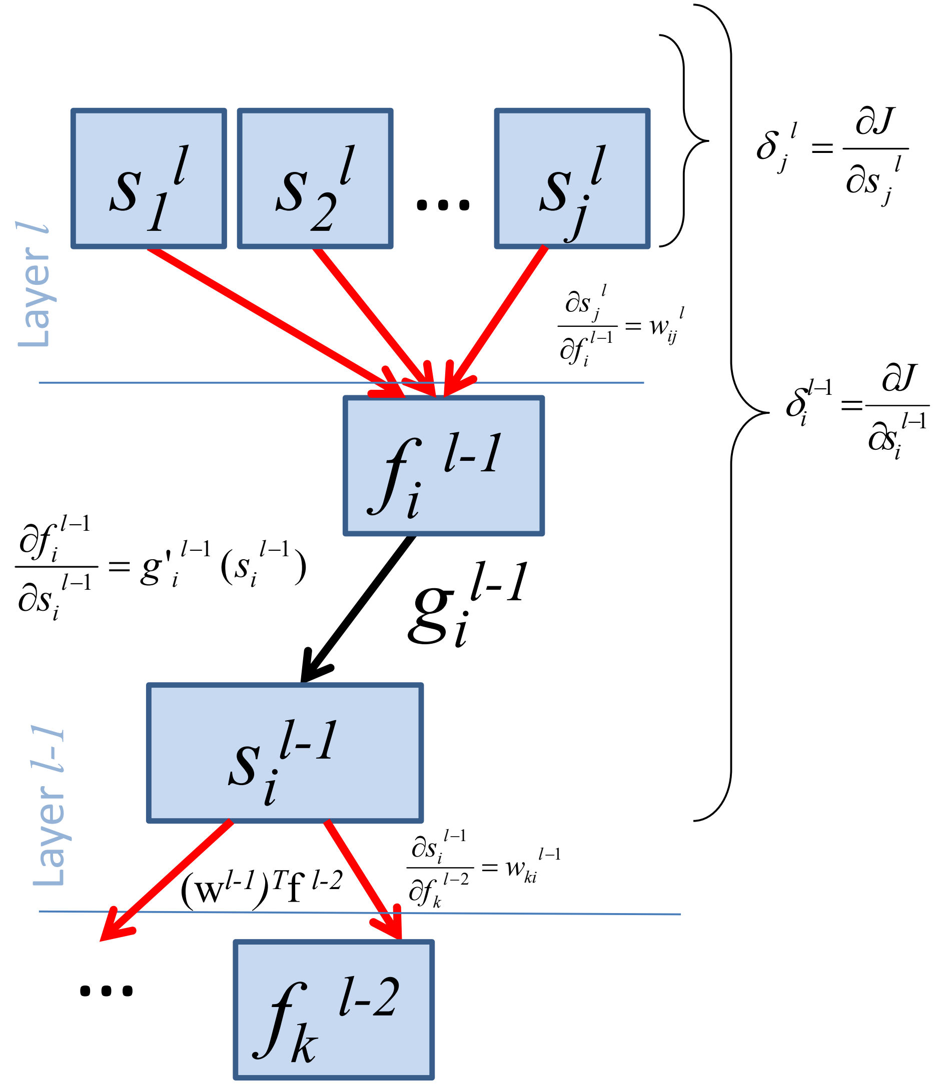
The ‘Dying Neuron’ Problem

- **Problem:** under circumstances where very large gradients are being passed through a ReLU unit, incoming weights may be changed (for instance towards strong negative values) so that the unit will not receive a signal above zero (ever) again and remains without output or learning contribution for the rest of training. Qualitatively, the following sequence may occur:

$$\underbrace{g'^{\,l-1}_i(s_i^{l-1})}_{\text{assumed open}}\sum_{j=1}^{d(l)}\underbrace{w_{ij}^l}_{\text{say pos}}\;\underbrace{\delta_j^l}_{\text{BIGpos}} \Rightarrow \underbrace{\delta_i^{l-1}}_{\text{BIGpos}} \Rightarrow$$

$$\underbrace{\eta\, f_k^{l-2}\delta_i^{l-1}}_{\text{BIGpos}} \Rightarrow \underbrace{w_{ki}^{l-1}}_{\text{BIGneg}} \Rightarrow \underbrace{s_i^{l-1}}_{\text{neg}}$$

$$\Rightarrow \underbrace{f_i^{l-1} = g_i^{l-1}(s_i^{l-1}) = 0}_{\text{zero}}$$



Guidance in the Zoo of Activation Functions

- avoid using sigmoids and expect *tanh* to work worse than ReLU
- as a current standard use ReLU as activation function: you may need to control the learning rate (set fairly low) and monitor the fraction of dead units
- leaky versions or generalisations of ReLU may help combat 'Dying Neurons'
- many more activation functions have been proposed, here is some of them:

$f(x)$	$f'(x)$
$f(x) = x$	$f'(x) = 1$
$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
$f(x) = \frac{x}{1+\lvert x \rvert}$	$f'(x) = \frac{1}{(1+\lvert x \rvert)^2}$
$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

sigmoid

ReLU

leaky ReLU

f	f'
$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} f(\alpha, x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ with $\lambda = 1.0507$ and $\alpha = 1.67326$	$f'(\alpha, x) = \lambda \begin{cases} f(\alpha, x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
$f_{t_l,a_l,t_r,a_r}(x) = \begin{cases} t_l + a_l(x - t_l) & \text{for } x \leq t_l \\ x & \text{for } t_l < x < t_r \\ t_r + a_r(x - t_r) & \text{for } x \geq t_r \end{cases}$ t_l, a_l, t_r, a_r are parameters.	$f'_{t_l,a_l,t_r,a_r}(x) = \begin{cases} a_l & \text{for } x \leq t_l \\ 1 & \text{for } t_l < x < t_r \\ a_r & \text{for } x \geq t_r \end{cases}$
$f(x) = \max(0, x) + \sum_{s=1}^{S} a_i^s \max(0, -x + b_i^s)$	$f'(x) = H(x) - \sum_{s=1}^{S} a_i^s H(-x + b_i^s)$
$f(x) = \ln(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$
$f(x) = \frac{\sqrt{x^2+1} - 1}{2} + x$	$f'(x) = \frac{x}{2\sqrt{x^2+1}} + 1$
$f(\alpha, x) = \begin{cases} -\frac{\ln(1-\alpha(x+\alpha))}{\alpha} & \text{for } \alpha < 0 \\ x & \text{for } \alpha = 0 \\ \frac{e^{\alpha x} - 1}{\alpha} + \alpha & \text{for } \alpha > 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \frac{1}{1-\alpha(\alpha+x)} & \text{for } \alpha < 0 \\ e^{\alpha x} & \text{for } \alpha \geq 0 \end{cases}$
$f(x) = \sin(x)$	$f'(x) = \cos(x)$
$f(x) = \begin{cases} 1 & \text{for } x = 0 \\ \frac{\sin(x)}{x} & \text{for } x \neq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x = 0 \\ \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} & \text{for } x \neq 0 \end{cases}$

overview source: wikipedia.com

Next Time: Optimisation Techniques

- Stochastic Gradient Descent
- Momentum and Nesterov Acceleration
- Newton's Method (2nd Order)
- Saddle Point Arguments
- Adaptive Gradient Descent