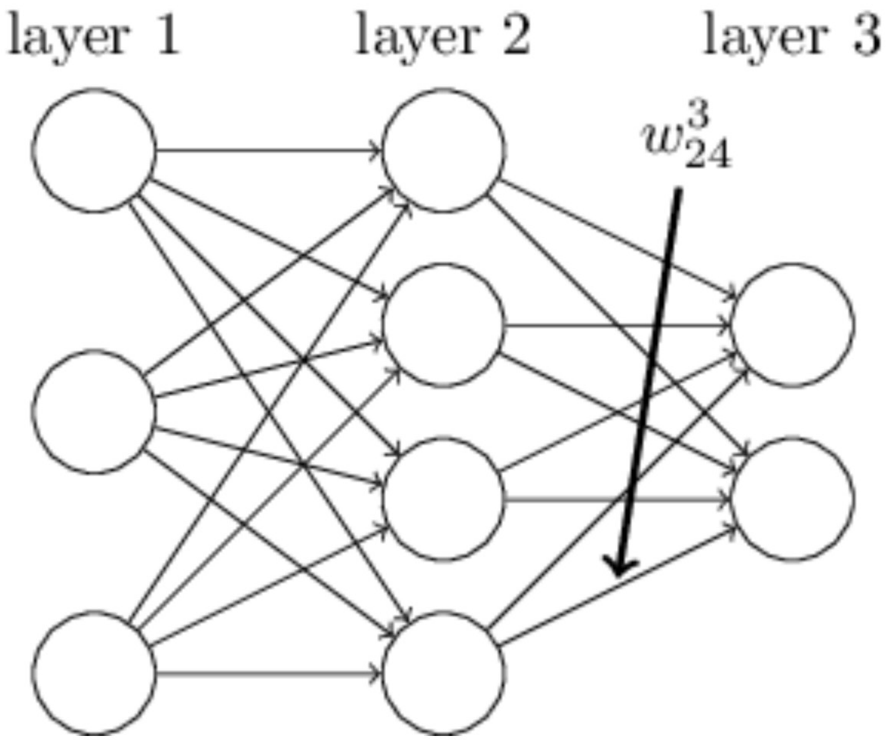
Deep Learning for Particle Physicists

Lewis Tunstall | AEC Graduate Seminar | May 20th 2022

Some loose ends 🧵

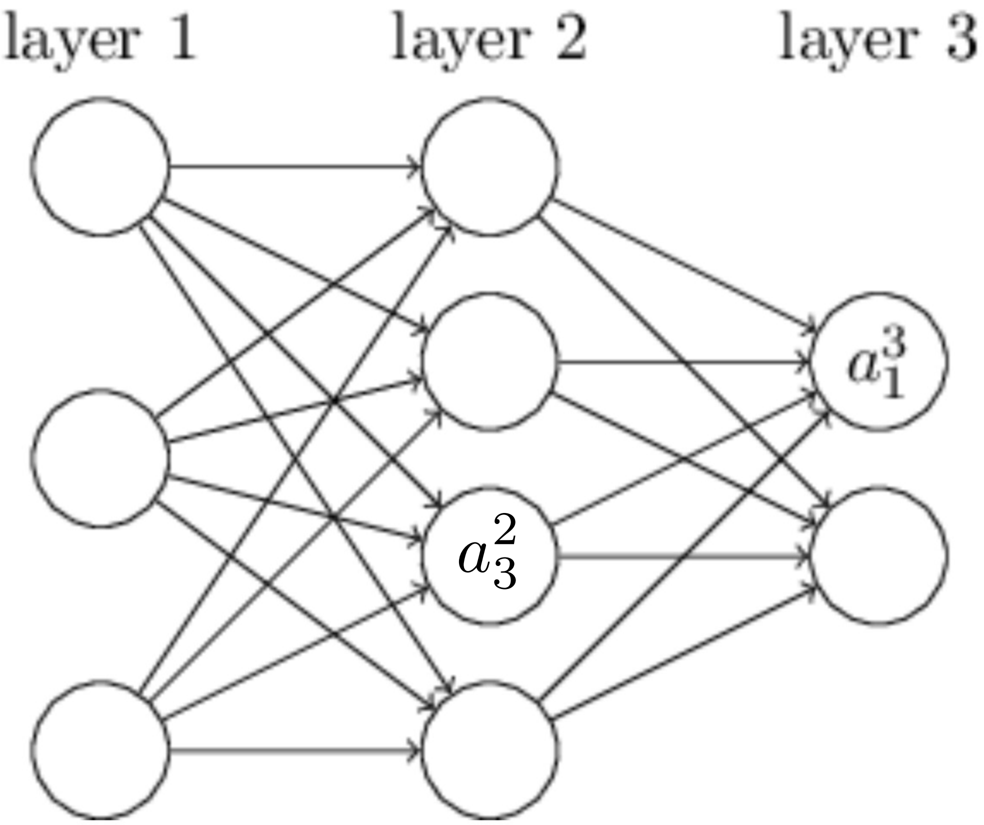
Activation functions



w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

NNs & Deep Learning (Nielsen)

Activation functions



activation function

l^{th} layer

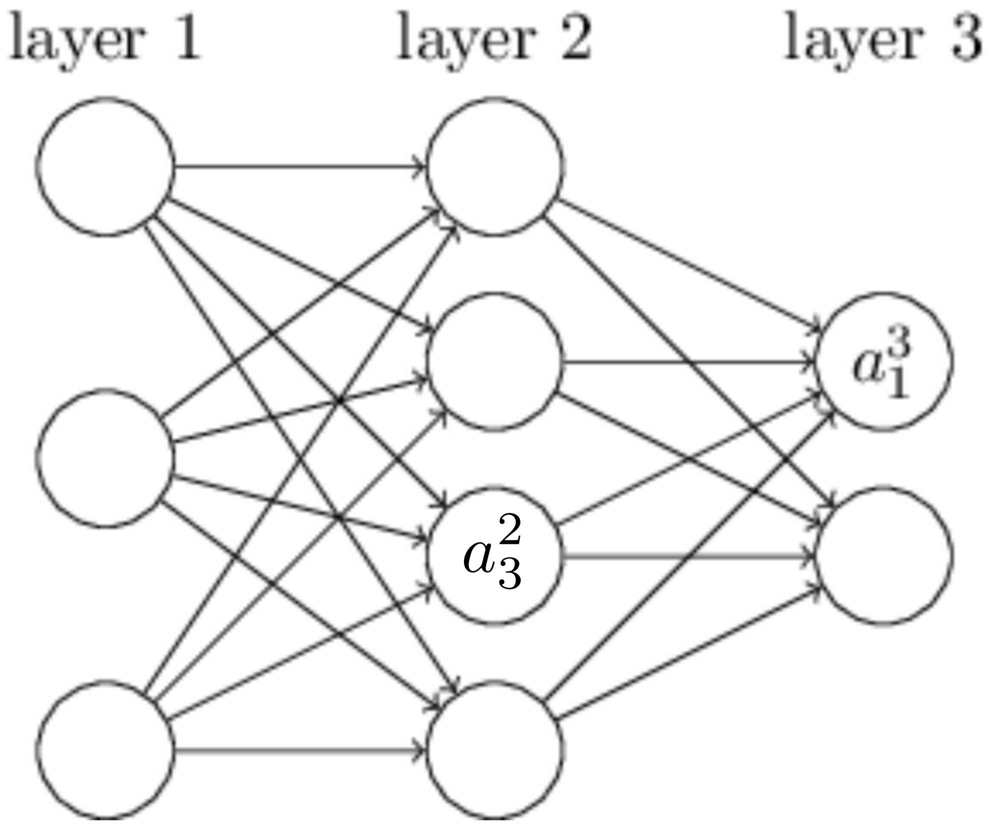
$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

j^{th} neuron

z^l_j weighted input

NNs & Deep Learning (Nielsen)

Activation functions

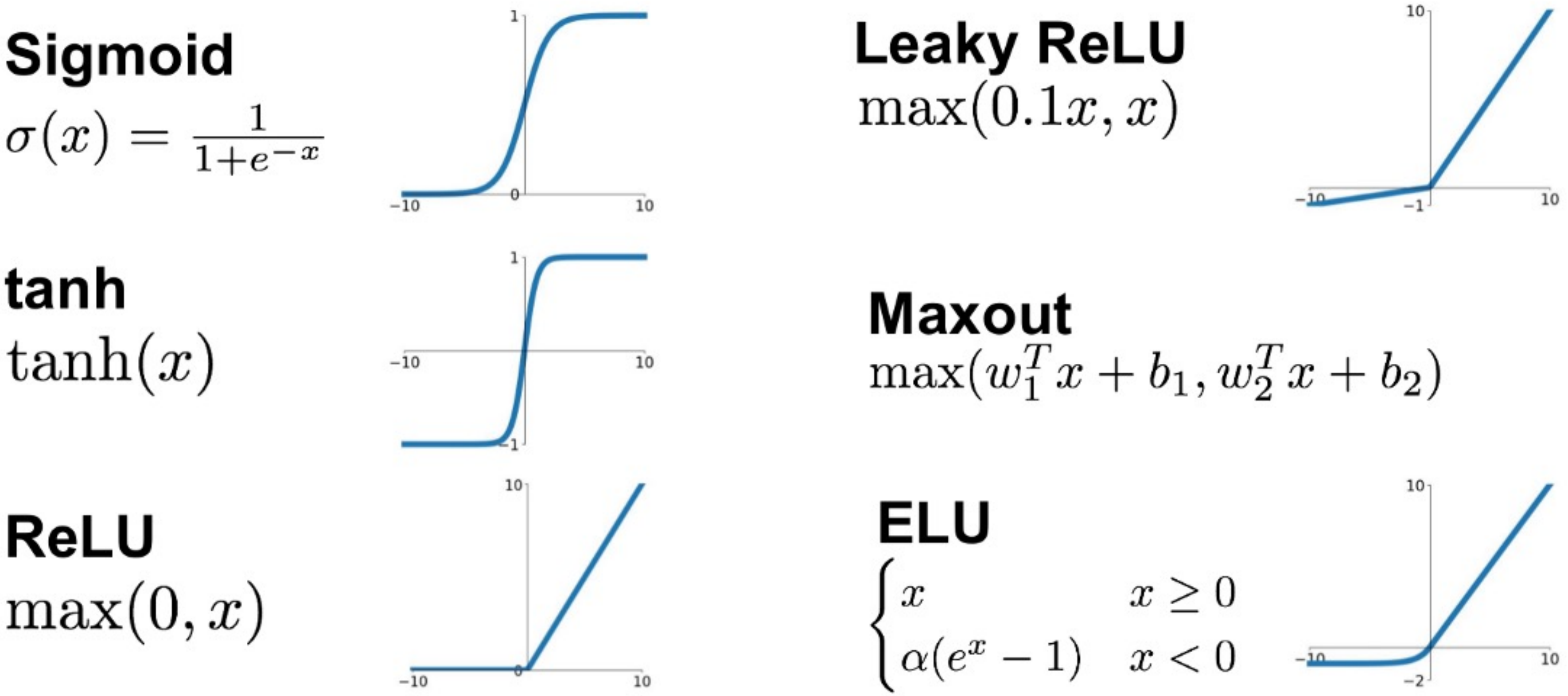


$$a^l = \sigma(w^l a^{l-1} + b^l)$$

matrix multiply = fast!

NNs & Deep Learning (Nielsen)

Activation functions

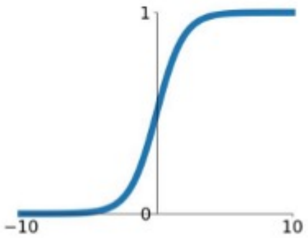


[Introduction to Different Activation Functions for Deep Learning]

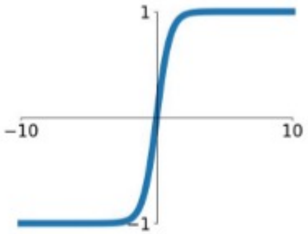
expensive
& saturates

Activation functions

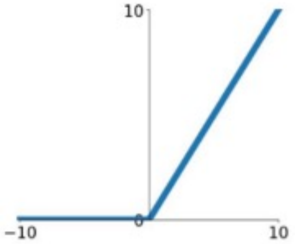
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



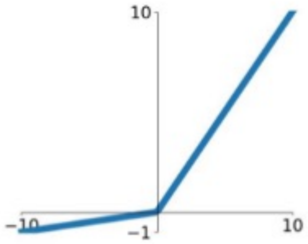
tanh
$\tanh(x)$



ReLU
$\max(0, x)$

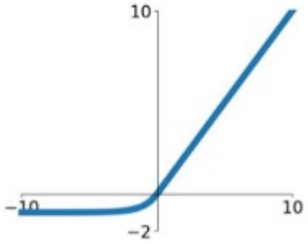


Leaky ReLU
$\max(0.1x, x)$



Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



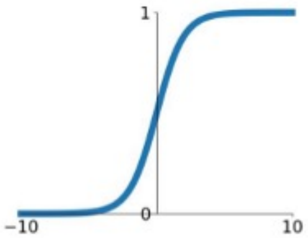
[Introduction to Different Activation Functions for Deep Learning]

Activation functions

expensive
& saturates

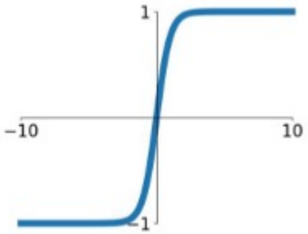
Sigmoid

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



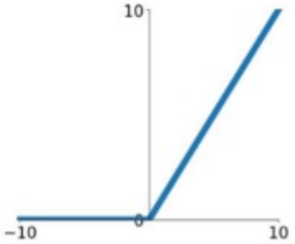
tanh

$\tanh(x)$



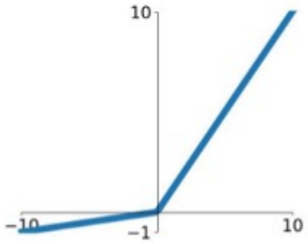
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

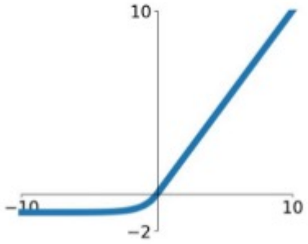


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



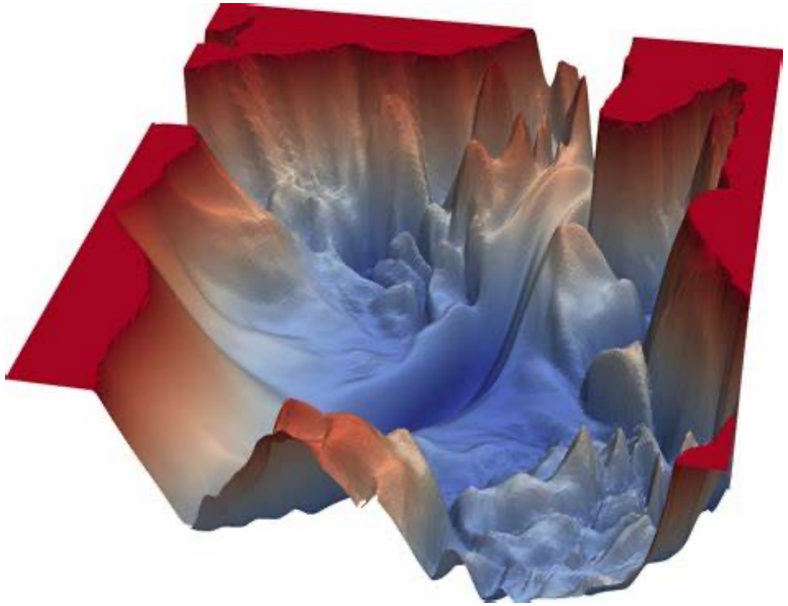
fast &
no saturation

[Introduction to Different Activation Functions for Deep Learning]

Backpropagation

(or what is really happening with `loss.backward()`?)

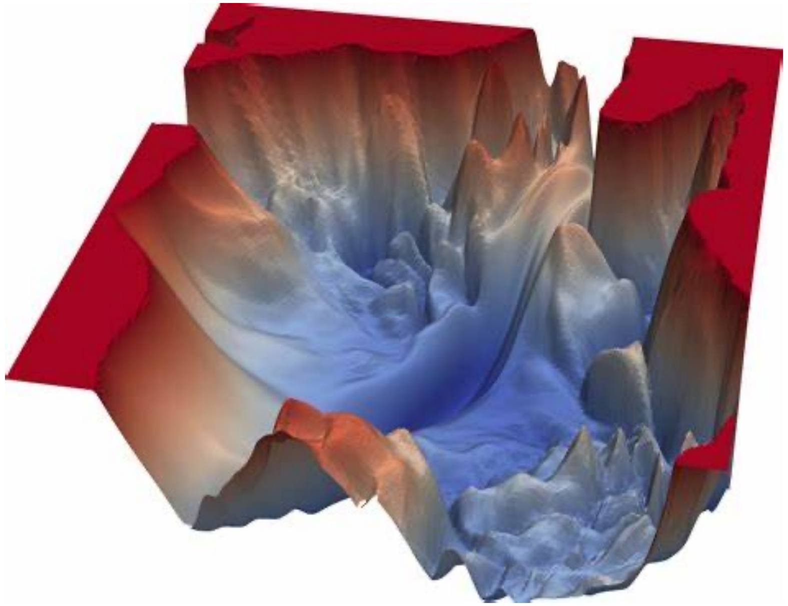
Update rule for SGD



$$w_k \to w'_k = w_k - \eta \frac{\partial L}{\partial w_k}$$

$$b_l \to b'_l = b_l - \eta \frac{\partial L}{\partial b_l}$$

Update rule for SGD



$$w_k \to w_k' = w_k - \eta \frac{\partial L}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial L}{\partial b_l}$$

compute gradients?

Computing gradients

$$\frac{\partial L}{\partial w_j} \approx \frac{L(w + \epsilon e_j) - L(w)}{\epsilon}$$

Try calculus?

Computing gradients

$$\frac{\partial L}{\partial w_j} \approx \frac{L(w + \epsilon e_j) - L(w)}{\epsilon}$$

✅ Conceptually simple

✅ Simple to implement

❌ Slooooow!

Try calculus?

Computing gradients

one forward pass
per weight

$$\frac{\partial L}{\partial w_j} \approx \frac{L(w + \epsilon e_j) - L(w)}{\epsilon}$$

✅ Conceptually simple

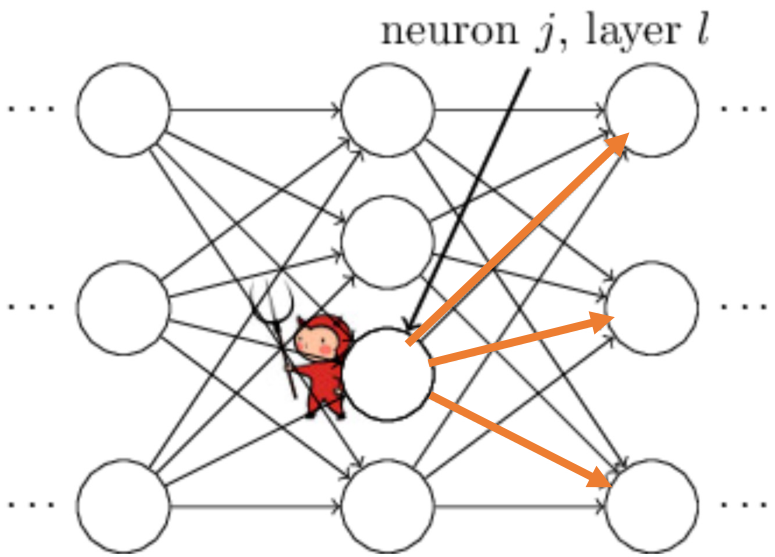
✅ Simple to implement

❌ Slooooow!

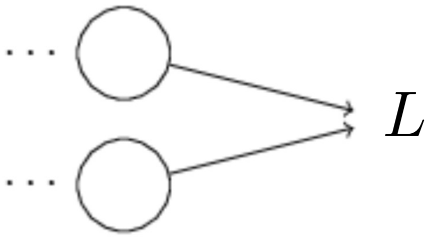
Try calculus?

Errors and backpropagation

perturb Δz_j^l and propagate $\Rightarrow$ $\frac{\partial L}{\partial z_j^l}\Delta z_j^l$ loss change



NNs & Deep Learning (Nielsen)

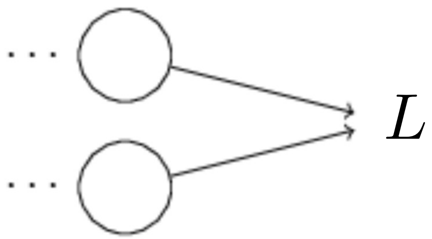


$$\delta_j^l \equiv \frac{\partial L}{\partial z_j^l}$$

neuron error

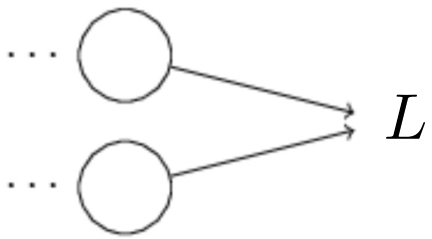
Basic idea: compute *neuron error* & relate to gradients via backprop

Step 1: Error in the output layer



$$
\begin{aligned}
\delta_j^L &= \frac{\partial L}{\partial z_j^L} \\
&= \sum_k \frac{\partial L}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} && \text{chain rule} \\
&= \frac{\partial L}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} && \text{connected neurons} \\
&= \frac{\partial L}{\partial a_j^L} \sigma'(z_j^L) && a_j^L = \sigma(z_j^L)
\end{aligned}
$$

Step 1: Error in the output layer



Hadamard product (element-wise)

$$\delta^L = \nabla_a L \odot \sigma'(z^L)$$

vectorised = fast!

Step 2: propagate error backwards

weight matrix transpose moves
error *backward* through network

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

error for (l+1)th layer

Basic idea: compute δ^l to compute δ^{l-1} then compute δ^{l-2} etc

Step 3: compute gradients

$$\delta^L = \nabla_a L \odot \sigma'(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial L}{\partial b_j^l} = \delta_j^l$$

compute gradients via error vectors,
starting from final layer

$$\frac{\partial L}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

Backpropagation algorithm

1. **Input x:** Set the corresponding activation a^1 for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.