Backpropagation

Backpropagation

- Originally introduced in the 1970s
- in 1986 a paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams show several neural networks where backpropagation works far faster than earlier approaches;
- it made it possible to use neural nets to solve problems which had previously been insoluble;
- today, the backpropagation algorithm is the workhorse of learning in neural networks.

Backpropagation vs Gradient Descent

We already saw the gradient descent algorithm. How does backpropagation differ?

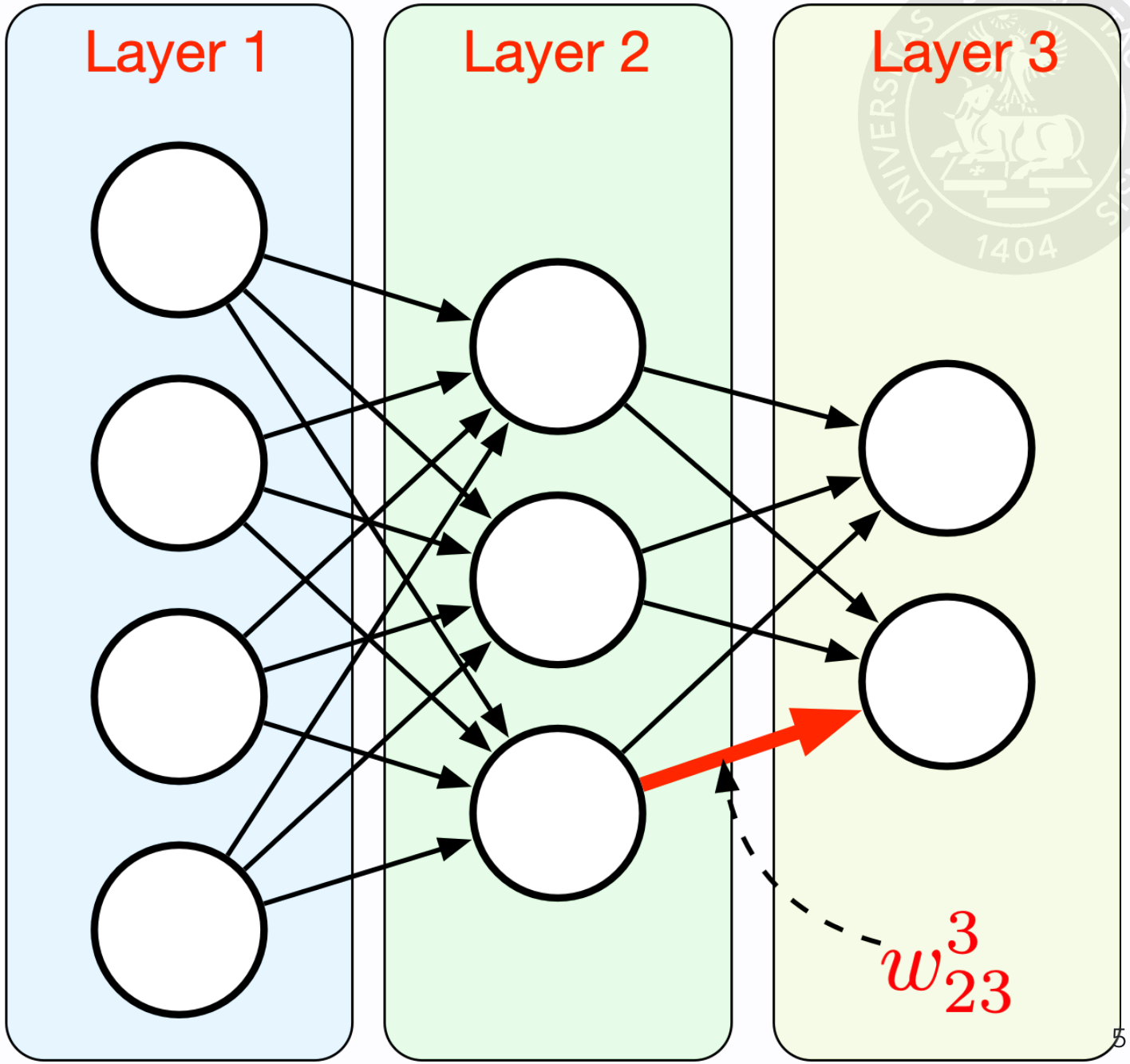
- Gradient descent is a general procedure for optimizing differentiable functions;
- backpropagation is its instantiation in the context of neural networks:
  - it fills in the details about how to compute the gradients;
  - it takes into account the structure of the network;
  - it is computationally efficient by avoiding repeating the calculation of gradients over and over again.

Warm Up: Notation

Matrix Based Notation

We will use w^l_{jk} to denote the weight from the k^{th} neuron in level $l-1$ to the j^{th} neuron in level l.

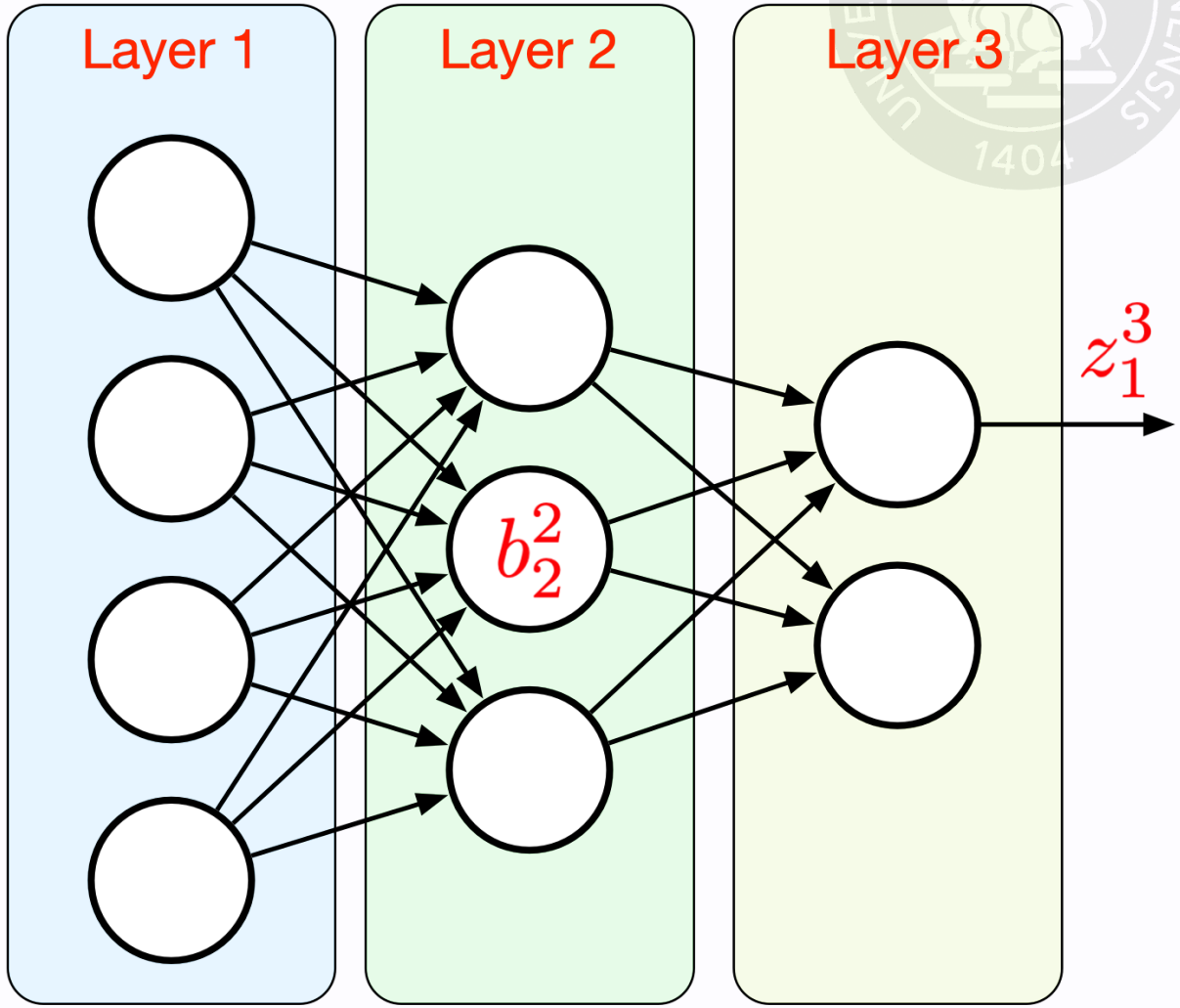
Note the ordering of the indices



Matrix Based Notation

Similarly, we will use:

- b_j^l to denote the bias of the j^{th} neuron in the l^{th} layer;
- z_j^l to denote the activation value of the j^{th} neuron in the l^{th}.



Matrix Based Notation

The activation z_j^l of the j^{th} neuron in the l^{th} layer is related to the activations in the $(l-1)^{\text{th}}$ layer by the equation:

$$z_j^l = \sigma\left(\sum_k w_{jk}^l z_k^{l-1} + b_j^l\right)$$

where the sum is on all neurons k in the $(l-1)^{\text{th}}$ layer.

Matrix Based Notation

Let us now define:

- $\mathbf{W}^l$ as the matrix having element w^l_{jk} at row j and column k;
- $\mathbf{b}^l$ as a column vector having b^l_j as its j^{th} element;
- $\mathbf{z}^l$ as a column vector having z^l_j as its j^{th} element;
- $f(\mathbf{v})$ will denote the "vectorized" version of f if $\mathbf{v}$ is a vector. E.g., if $f(x) = x^2$, we would have:

$$f\left(\begin{bmatrix}2\\3\end{bmatrix}\right) = \begin{bmatrix}f(2)\\f(3)\end{bmatrix}$$

Matrix Based Notation

With these notations we can compute the whole set of activations on level l as:

$$\mathbf{z}^l = \sigma(\mathbf{W}^l \mathbf{z}^{l-1} + \mathbf{b}^l)$$

Advantages:

- it is simpler to write down, to reason about it, and to remember
- it is simpler to implement and the result is faster to compute (by taking advantage of numerical libraries to compute the matrix operations).

Matrix Based Notation

To simplify notation later on, we define the weighting input to the neurons at level l as a_l:

$$\mathbf{a}^l \equiv \mathbf{W}^l \mathbf{z}^{l-1} + \mathbf{b}^l$$

this notation:

- allow us to write the outputs at level l simply as: $\mathbf{z}^l = \sigma(\mathbf{a}^l)$
- implies that component j of vector $\mathbf{a}^l$ is $a_j^l = \sum_k w_{jk}^l z_k^{l-1} + b_j^l$.

Assumptions to be Made on the Cost Function

In order for the backpropagation algorithm to work, we need to make two assumptions about the cost function. Specifically, we will assume that:

1. it can be written as an average $C = \frac{1}{n} \sum_{\mathbf{x}} C_{\mathbf{x}}$ over cost functions $C_{\mathbf{x}}$ for individual training examples, $\mathbf{x}$.
2. it is a function of the outputs of the neural network.

Assumptions to be Made on the Cost Function

As an example, it is easy to see that both assumptions are satisfied by the quadratic cost function we introduced earlier:

$$C = \frac{1}{2n} \sum_{\mathbf{x}} \|\mathbf{y}(\mathbf{x}) - \mathbf{z}^L(\mathbf{x})\|^2$$

Hadamard Product

In the following we will assume familiarity with all the standard operations over matrices (sum, subtraction, product, etc.).

One operation which is a little less commonly used is the Hadamard product.

Definition

Given two vectors $\mathbf{s}$ and $\mathbf{t}$, the Hadamard product of the two (written $\mathbf{s} \odot \mathbf{t}$) is just the elementwise product of the two vectors, i.e., the components of $\mathbf{s} \odot \mathbf{t}$ are just $(\mathbf{s} \odot \mathbf{t})_j = s_j t_j$.

As an example:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \times 3 \\ 2 \times 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

Backpropagation: the Four Fundamental Equations

In developing our understanding of the backpropagation algorithm, we aim at studying how to efficiently calculate the quantities $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$.

It happens to be easier to do so by first calculating another quantity:

$$\delta^l_j = \frac{\partial C}{\partial a^l_j}.$$

We will refer to δ^l_j as the error at level l and neuron j.

Backpropagation will give us a procedure to compute δ^l_j for every layer l and neuron j, and then will relate δ_j to the quantities of real interest.

Meaning of $\boldsymbol{\delta}^{l}$

The error at level l is meant to measure how much the cost function varies when the inputs of the neurons at that level are slightly perturbed.

The term "error" derives from the idea of introducing a little "error" in the inputs of the layer and observing how it propagates to the cost function.

Backpropagation: the Four Fundamental Equations

The backpropagation algorithm is based on "four fundamental equations".

We will introduce them first without a proof concentrating on their meaning, then we will show how to derive them from first principles.

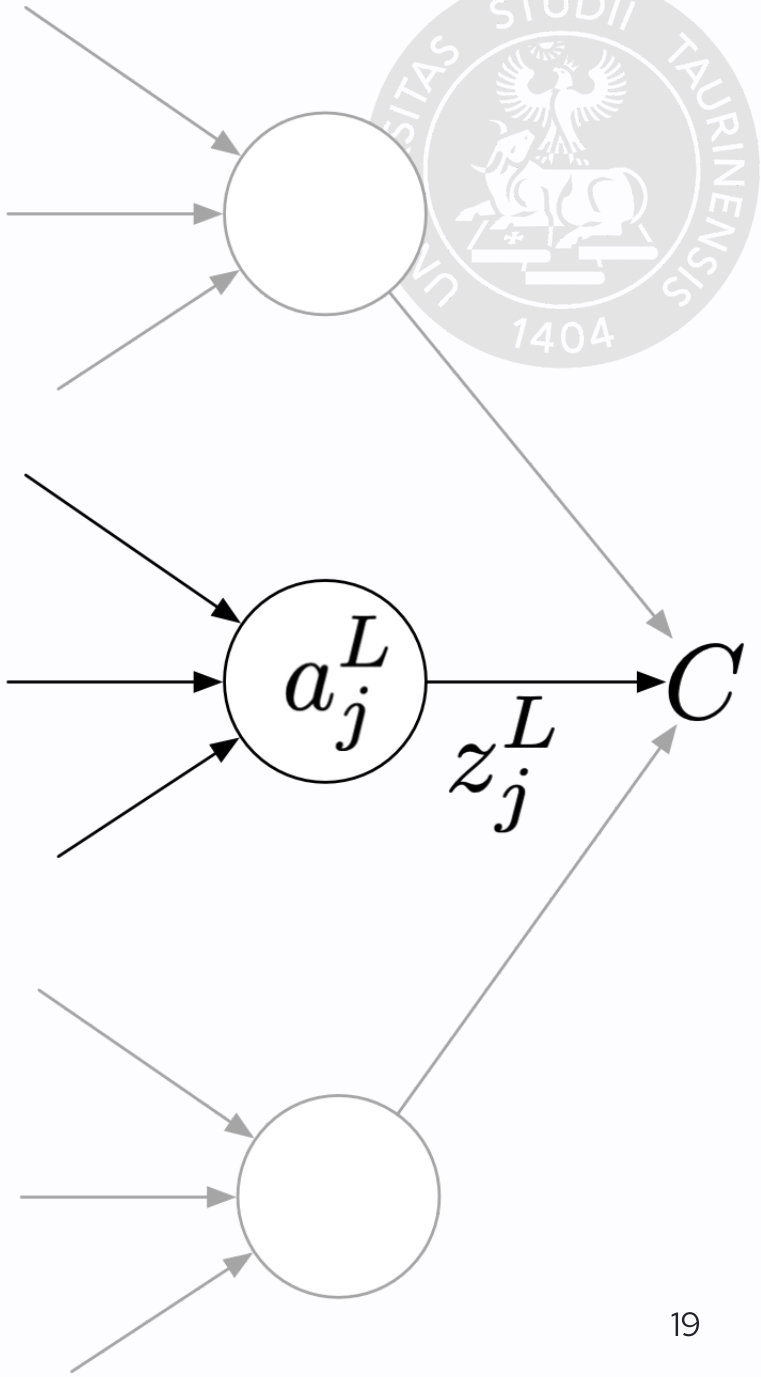
An Equation for the Error in the Output Layer

Equation **BP1** specifies how to compute $\boldsymbol{\delta}^L$, the error at the output layer.

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}\sigma'(a_j^L)$$

Interpretation

- $\delta_j^L = \frac{\partial C}{\partial z_j^L} \sigma'(a_j^L)$
- $z_j^l = \sigma(a_j^L)$



An Equation for the Error in the Output Layer

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}\sigma'(a_j^L)$$

Notice that everything in `BP1` is easily computable. a_j^L is easily computed by just letting the inputs to flow through the network up to neuron j at level L and it is then very easy to compute $\sigma'(a_j^L)$.

The exact form of $\frac{\partial C}{\partial z_j^L}$ will depend on the form of the cost function. For instance, if the cost is the quadratic cost we already introduced, then

$$C = \frac{1}{2}\sum_j (y_j - z_j^L)^2 \Rightarrow \frac{\partial C}{\partial z_j^L} = (z_j^L - y_j).$$

An Equation for the Error in the Output Layer

Given the componentwise expression $\delta_j^L = \frac{\partial C}{\partial z_j^L}\sigma'(a_j^L)$, it should not be hard to see that the whole $\boldsymbol{\delta}^L$ could be written as:

$$\boldsymbol{\delta}^L = \nabla_{\mathbf{z}} C \odot \sigma'(\mathbf{a}^L).$$

Where $\nabla_{\mathbf{z}} C$ is the vector whose components are the partial derivatives $\partial C / \partial z_j^L$.

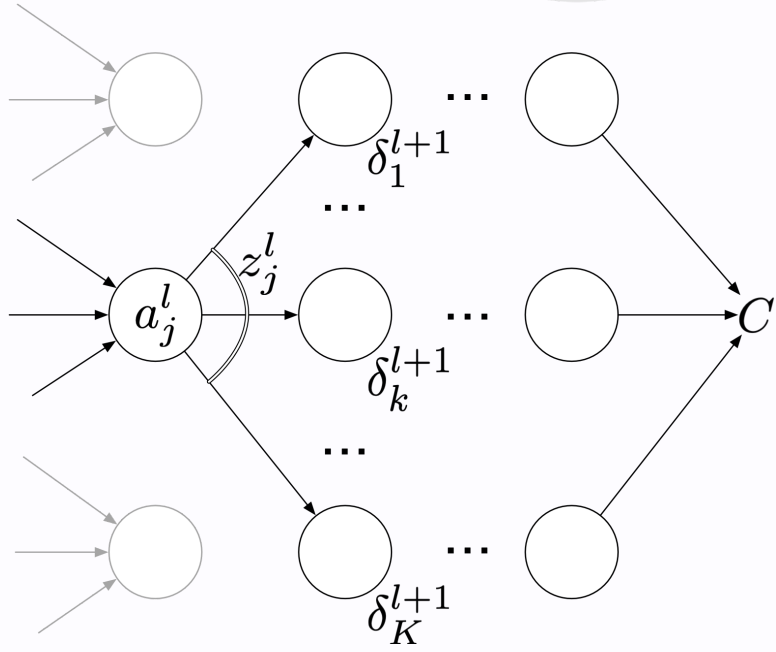
An equation for $\boldsymbol{\delta}^l$ in terms of $\boldsymbol{\delta}^{l+1}$

Equation BP2 specifies how to compute the error at level l in terms of the error at level $l+1$:

$$\boldsymbol{\delta}^l = ((\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot \sigma'(\mathbf{a}^l).$$

Interpretation

- $\boldsymbol{\delta}^l = ((\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot \sigma'(\mathbf{a}^l)$.
- a term measuring how fast the output at this level varies in response to a variation of its input: $\sigma'(\mathbf{a}^l)$;
- multiplied by a term that measures how that changes propagate through the network via the connecting weights: $(\mathbf{W}^{l+1})^T$;
- multiplied by how fast that level will change in response to changes in its input.



An equation for $\boldsymbol{\delta}^l$ in terms of $\boldsymbol{\delta}^{l+1}$

Again everything in the expression

$$\boldsymbol{\delta}^l = ((\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot \sigma'(\mathbf{a}^l).$$

is easily computable since the only non-trivial term, i.e., $\boldsymbol{\delta}^{l+1}$ can be computed by applying equation `BP1` to the last level of the network and then equation `BP2` until we reach the level we are interested in.

An equation for the rate of change of the cost w.r.t. the biases in the network

BP3 : this is the first of the two equations we are *really* interested in. In fact, these are the ones that are directly related to applying gradient descent to a neural network.

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

That is, th error δ^l_j is e*xactly equal* to the rate of change $\partial C / \partial b^l_j$.

An equation for the rate of change of the cost w.r.t. the weights in the network

BP4 specifies how to compute the rate of change of the cost w.r.t. any of the weights in the network:

$$\frac{\partial C}{\partial w^l_{jk}} = z^{l-1}_k \delta^l_j$$

It can also be written in a more compact form as:

$$\frac{\partial C}{\partial w} = z_{\text{in}} \delta_{\text{out}}$$

where it is intended that the weight w w.r.t. we are taking the derivative determines the levels and the indices of z_{in} and δ_{out}.

Consequences of equations `BP1` - `BP4`

Let us now draw some consequences about how learning evolves given the above equation. Starting from `BP4` :

$$\frac{\partial C}{\partial w} = z_{\text{in}} \delta_{\text{out}}$$

we note that it implies that whenever a_{in} is small, the gradient will also tend to be small and the learning of that weight will be slow. In other words, weights originating from *low activation* units will evolve slowly.

Consequences of equations **BP1 - BP4**

Consider again equation **BP1** :

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}\sigma'(a_j^L)$$

and in particular the term $\sigma'(a_j^L)$. The graph of the sigmoid function implies that its derivative is almost zero when its argument is either very large or very small.

I.e., learning will proceed slowly for an output neuron if it is either *low activation* (≈ 0) or high activation (≈ 1).

When the derivative of the activation function is almost zero it is common to say that the neuron **saturated**.

Consequences of equations `BP1` – `BP4`

Similar considerations can be made about the implications of `BP2` over the rate of learning of any weight in the network: if the neuron saturates the learning will be slow.

Importantly, these observations do not rely on the activation function being the sigmoid (since the derivations of `BP1` - `BP4` do not depend on that), they *do apply* to any activation function.

Designing new activation functions

The generality of the observations about `BP1` - `BP4` means that we can use these insights to design new activation functions which have particular desired learning properties.

As an example, suppose we were to choose a (non-sigmoid) activation function σ so that σ' is always positive, and never gets close to zero.

That would prevent the slow-down of learning that occurs when ordinary sigmoid neurons saturate.

Summary: the equations of backpropagation

- **(BP1)**: $\boldsymbol{\delta}^L = \nabla_{\mathbf{z}} C \odot \sigma'(\mathbf{a}^L)$
- **(BP2)**: $\boldsymbol{\delta}^l = ((\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot \sigma'(\mathbf{a}^l)$
- **(BP3)**: $\frac{\partial C}{\partial b^l_j} = \delta^l_j$
- **(BP4)**: $\frac{\partial C}{\partial w^l_{jk}} = z^{l-1}_k \delta^l_j$