Scientific Machine Learning 03
Backpropagation: Math, Intuition, and the Full Algorithm

Donghyun Ko

January 4, 2026

Backpropagation is the engine that makes neural networks learn *efficiently*. In this post we (i) set up clean layerwise notation, (ii) state two mild assumptions on the cost, (iii) derive the four fundamental backprop equations (BP1–BP4) with short proofs, and (iv) assemble the full algorithm—including how it couples with SGD in practice.

Contents

1 Why this post? What you'll learn

We extend gradient descent from last time to *compute* the gradients needed for learning:

- A compact notation for weights, biases, activations, and weighted inputs.
- Two assumptions on the cost that make backprop work cleanly.
- The four equations behind backprop (BP1–BP4) with step-by-step derivations.
- The backprop algorithm (forward pass, output error, backward sweep) and how it plugs into SGD.

Learning goals — Understand *what* each symbol means, *why* the chain rule yields BP1–BP4, and *how* those equations translate into a fast training loop in code.

2 Motivation & Notations

Recall the SGD update from last time (for a weight w_k and bias b_ℓ):

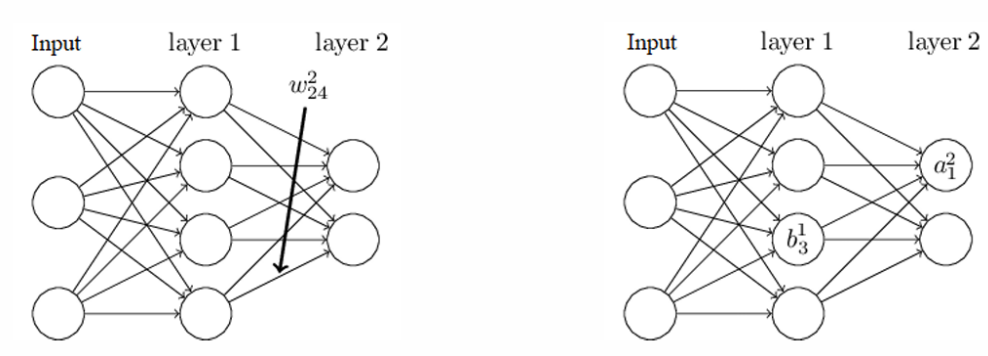
$$\begin{aligned} w_k \;\to\; w_k' \;&=\; w_k - \eta\,\frac{\partial C}{\partial w_k} \;=\; w_k - \eta\,\frac{1}{m}\sum_{i=1}^{m}\frac{\partial C_{x_i}}{\partial w_k}, \\ b_\ell \;\to\; b_\ell' \;&=\; b_\ell - \eta\,\frac{\partial C}{\partial b_\ell} \;=\; b_\ell - \eta\,\frac{1}{m}\sum_{i=1}^{m}\frac{\partial C_{x_i}}{\partial b_\ell} \end{aligned} \tag{1}$$

Backpropagation gives us a fast way to compute those partial derivatives for *all* parameters using just one forward pass and one backward pass per example (or per mini-batch average). This is exponentially cheaper than naive finite differences when the parameters are many.

Extra notes. Backpropagation algorithm was originally introduced in the 1970s, but its pivotal role was cemented by Rumelhart, Hinton, and Williams (1986). It powers modern ANN training by providing efficient formulas for the gradients $\frac{\partial C}{\partial w_k}$ and $\frac{\partial C}{\partial b_\ell}$, making clear how changing the parameters (i.e., weights and biases) changes the overall behaviour(output and cost) of the ANN.

Notation (layerwise and vectorized). We use the following layer-ℓ objects and indices:

- w_{jk}^{ℓ}: **weight** from neuron k in layer $(\ell-1)$ *to* neuron j in layer ℓ.
- b_j^{ℓ}: **bias** of neuron j in layer ℓ.
- a_j^{ℓ}: **activation** (output) of neuron j in layer ℓ.
- $\mathbf{W}^{\ell}$: **weight matrix** into layer ℓ whose (j,k) entry is w_{jk}^{ℓ}.
- $\mathbf{b}^{\ell}$: **bias vector** for layer ℓ with entries b_j^{ℓ}.
- $\mathbf{a}^{\ell}$: **activation vector** for layer ℓ with entries a_j^{ℓ}.



Applying a non-linearity function $\sigma(\cdot)$ to a vector is *entrywise*:

$$\sigma(\mathbf{v})_j = \sigma(v_j) \quad \text{(e.g., if } f(x)=x^2,\quad f\begin{bmatrix}2\\3\end{bmatrix} = \begin{bmatrix}4\\9\end{bmatrix}\text{)}$$

For each activation component,

$$z_j^\ell \;=\; \sum_k w_{jk}^\ell \, a_k^{\ell-1} \;+\; b_j^\ell, \qquad a_j^\ell \;=\; \sigma\Big(z_j^\ell\Big)$$

where the sum is over all neurons k in the $(l-1)^{th}$ layer.
With this convention, we can rewrite the activation formula in a compact vectorized form s.t:

$$\mathbf{z}^\ell \;=\; \mathbf{W}^\ell \, \mathbf{a}^{\ell-1} + \mathbf{b}^\ell, \qquad \mathbf{a}^\ell \;=\; \sigma(\mathbf{z}^\ell),$$

where $\mathbf{z}^\ell$ is a **weighted input** (pre-activation) vector in the layer ℓ with entries z_j^ℓ.

Why this indexing for $\mathbf{W}^\ell$? Row j of $\mathbf{W}^\ell$ collects *all* incoming weights to neuron j in layer ℓ; column k collects *all* weights that *come from* neuron k in layer $(\ell-1)$. Thus, the (j,k) entry is w_{jk}^ℓ and the product $\mathbf{W}^\ell\mathbf{a}^{\ell-1}$ reads naturally as

$$\mathbf{W}^\ell\mathbf{a}^{\ell-1} = \begin{bmatrix} \sum_k w_{1k}^\ell a_k^{\ell-1} \\ \sum_k w_{2k}^\ell a_k^{\ell-1} \\ \vdots \\ \sum_k w_{Jk}^\ell a_k^{\ell-1} \end{bmatrix}$$

Hadamard (elementwise) product. Backpropagation uses the element-wise product (also known as Schur product), denoted $\odot$. For conformable vectors or matrices,

$$(\mathbf{A} \odot \mathbf{B})_{ij} = A_{ij}\, B_{ij}.$$

It is *not* the usual matrix product. For example,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & a_{13}b_{13} \\ a_{21}b_{21} & a_{22}b_{22} & a_{23}b_{23} \\ a_{31}b_{31} & a_{32}b_{32} & a_{33}b_{33} \end{bmatrix}$$

In vector form, we will often write $(\mathbf{u} \odot \mathbf{v})_j = u_j v_j$.

3 Two mathematical assumptions on the cost function

Recall the quadratic cost function:

$$C(\mathbf{W}, \mathbf{b}) = \frac{1}{2n} \sum_x \Big\| \, y(x) - a(x) \, \Big\|^2 = \frac{1}{2n} \sum_x \Big\| \, y(x) - a^L(x) \, \Big\|^2$$

- 'n' is the total number of training examples
- The sum is over all training examples x
- $y(x)$ is the desired output (the true label for input x)

- L denotes the number of layers in the network; the L-th layer is the output layer
- $a^L(x)$ is the network activation/output at the output layer when x is input

A1 (dataset cost is an average). The first assumption we need is that the cost function can be written as an average over cost functions C_x for individual training examples, x:

$$C \;=\; \frac{1}{n}\sum_x C_x$$

This is the case for the quadratic cost where the per-example cost is

$$C_x \;=\; \tfrac{1}{2}\left\|y(x) - a^L(x)\right\|^2$$

> **Why the $\frac{1}{2}$?** It makes derivatives clean: for both scalars and vectors, $\nabla_a C_x = a - y$. Without $\frac{1}{2}$, we would carry an extra factor of 2. A positive constant factor can be absorbed into the learning rate η.

The reason we need this assumption is because what backpropagation actually allows us to do is to compute the partial derivatives $\dfrac{\partial C_x}{\partial w}$ and $\dfrac{\partial C_x}{\partial b}$ for a single training example. We then recover $\dfrac{\partial C}{\partial w}$ and $\dfrac{\partial C}{\partial b}$ by averaging over all training examples:

$$\frac{\partial C}{\partial w} \;=\; \frac{1}{n}\sum_x \frac{\partial C_x}{\partial w}, \qquad \frac{\partial C}{\partial b} \;=\; \frac{1}{n}\sum_x \frac{\partial C_x}{\partial b}$$

A2 (per-example cost depends on outputs). For a fixed input x, the label $y(x)$ is fixed, so the cost for that example is a function of the network outputs at the output layer, $a^L(x)$:

$$C_x \;=\; \frac{1}{2}\left\|\, y(x) - a^L(x)\,\right\|^2 \;=\; \frac{1}{2}\sum_i \Big(y_i(x) - a_i^L(x)\Big)^2$$

Although the quadratic cost uses the desired output y_i, once the training sample x is fixed, $y(x)$ is fixed and *does not depend* on $(\mathbf{W}, \mathbf{b})$. Only the network outputs $a^L(x)$ change with the parameters. Hence, it is natural to view the per-example cost as a function of the outputs alone when we think about derivatives:

$$C_x \;=\; \tfrac{1}{2}\left\|y(x) - a^L(x)\right\|^2 \;\equiv\; \tilde{C}\Big(a^L(x);\, y(x)\Big), \quad \text{with } y(x) \text{ treated as constant}$$

Consequently, all parameter derivatives flow through $a^L(x)$ via the chain rule, e.g.

$$\frac{\partial C_x}{\partial w} = \Big(\nabla_{a^L} C_x\Big)^\top \frac{\partial a^L(x)}{\partial w}, \qquad \frac{\partial C_x}{\partial b} = \Big(\nabla_{a^L} C_x\Big)^\top \frac{\partial a^L(x)}{\partial b}$$

> **Why these assumptions?** Backprop efficiently computes $\partial C_x/\partial w$ and $\partial C_x/\partial b$ for a *single* example by chaining derivatives from the output layer backward. Assumption A1 then lets us recover dataset gradients by averaging:
>
> $$\nabla C \;=\; \frac{1}{n}\sum_x \nabla C_x$$
>
> Assumption A2 ensures the chain rule applies cleanly from C_x back through the layers.

4 The four equations of backpropagation

Define the error δ_j^ℓ Backpropagation introduces an intermediate neuron-wise *error* in the process, which we call the error in the j^{th} neuron in the l^{th} layer:

$$\delta_j^\ell \;=\; \frac{\partial C}{\partial z_j^\ell}, \qquad \delta^\ell = \frac{\partial C}{\partial z^\ell} \text{ (vector)}$$

It looks slightly unnatural at first, but such a definition makes the derivation of the backpropagation algorithm less algebraically complicated because z^ℓ sits just *before* the activation. Gradients w.r.t. W^ℓ and b^ℓ will factor nicely through δ^ℓ.

> **Intuition** — δ_j^ℓ measures how much the cost would change if we nudged the pre-activation z_j^ℓ. This is exactly the signal we need to attribute the error to the weights and biases that feed into the neuron (ℓ, j).

We now derive (BP1)–(BP4). Throughout, $\odot$ denotes the Hadamard product.

BP1: Error at the output layer

Component form. For an output layer L, the error at neuron j is

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

Matrix–vector form. Stacking all j gives

$$\boxed{\delta^L \;=\; \nabla_{a^L} C \;\odot\; \sigma'(z^L)}$$

Proof. At the output layer, we have the following.

$$a^L = \sigma(z^L) \quad \text{(entry-wise)}, \quad \text{and} \quad C = C\Big(a^L\Big)$$

So 'C' depends on z^L *only through* a^L. Now, imagine fixing an output neuron index j. Then, by the chain rule (i.e., think of it as we have one extra layer between z and C, called 'a'),

$$\delta_j^L \;\overset{\text{def}}{=}\; \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L}\,\frac{\partial a_k^L}{\partial z_j^L}$$

But, $a_k^L = \sigma(z_k^L)$ depends on z_j^L *only when* $k = j$ since they are element-wised. Hence,

$$\frac{\partial a_k^L}{\partial z_j^L} = \begin{cases} \sigma'(z_j^L), & k = j, \\ 0, & k \neq j, \end{cases} \quad\Longrightarrow\quad \delta_j^L \;=\; \frac{\partial C}{\partial a_j^L}\,\sigma'(z_j^L)$$

Stacking over j gives the vector form

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

BP1 from Jacobian perspective, and example

Setup. The output layer applies the nonlinearity entry-wise: $a^L = \sigma(z^L)$. Let the Jacobian of a^L w.r.t. z^L be

$$J \overset{\text{def}}{=} \frac{\partial a^L}{\partial z^L} = \operatorname{diag}(\sigma'(z^L))$$

Chain rule (vector form). The output-layer error is

$$\delta^L \overset{\text{def}}{=} \frac{\partial C}{\partial z^L} = \left(\frac{\partial a^L}{\partial z^L}\right)^\top \frac{\partial C}{\partial a^L} = \operatorname{diag}(\sigma'(z^L))\, \nabla_{a^L} C = \nabla_{a^L} C \odot \sigma'(z^L)$$

Concrete example (MSE). For $C_x = \frac{1}{2}\|y - a^L\|^2$, we have

$$\nabla_{a^L} C_x = a^L - y \quad \Rightarrow \quad \boxed{\delta^L = (a^L - y) \odot \sigma'(z^L)}$$

If σ is sigmoid and a neuron is saturated, then $\sigma'(z^L) \approx 0$, so δ^L becomes tiny *even when the prediction error* $(a^L - y)$ *is not small*—this explains slow learning at saturated outputs. $\nabla_{a^L} C$ measures "how the cost changes under a *small change* in the outputs." $\sigma'(z^L)$ gates that signal by the local slope of the nonlinearity. Element-wise multiplying them gives "how the cost changes under a *small change* in the pre-activations z^L," i.e., δ^L.

BP2: Error at a hidden layer

Component form. For a hidden layer ℓ ($\ell = L-1, \ldots, 1$), the error at neuron j is

$$\boxed{\delta_j^\ell = \left(\sum_k w_{kj}^{\ell+1}\, \delta_k^{\ell+1}\right) \sigma'(z_j^\ell)}$$

Matrix–vector form. Stacking all j gives

$$\boxed{\boldsymbol{\delta}^\ell = \left(\mathbf{W}^{\ell+1}\right)^\top \boldsymbol{\delta}^{\ell+1} \odot \sigma'(\mathbf{z}^\ell)}$$

where $\odot$ denotes the Hadamard (element-wise) product and σ' is applied entry-wise.

Proof. Fix a hidden layer ℓ and a neuron index j. We are interested in how the cost function changes when the pre-activation in the neuron j changes:

$$\delta_j^\ell \overset{\text{def}}{=} \frac{\partial C}{\partial z_j^\ell}$$

It helps to remember that any influence of z_j^ℓ on C must pass through the next layer $(\ell+1)$:

$$z^\ell \xrightarrow{a^\ell = \sigma(z^\ell)} a^\ell \xrightarrow{z^{\ell+1} = W^{\ell+1} a^\ell + b^{\ell+1}} z^{\ell+1} \to \cdots \to C$$

With this dependency in mind, the chain rule tells us:

$$\delta_j^\ell = \sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_k \delta_k^{\ell+1} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} \tag{1}$$

Between the two layers, we have the familiar forward map such as:

$$z_k^{\ell+1} = \sum_r w_{kr}^{\ell+1} a_r^\ell + b_k^{\ell+1}, \qquad a_r^\ell = \sigma(z_r^\ell) \tag{2}$$

So, $\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell}$ in (1) is obtained by letting the effect pass first through a^ℓ:

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_r \frac{\partial z_k^{\ell+1}}{\partial a_r^\ell} \frac{\partial a_r^\ell}{\partial z_j^\ell} \tag{3}$$

Here, the two factors are straightforward. From (2), the coefficient of a_r^ℓ in $z_k^{\ell+1}$ is $w_{kr}^{\ell+1}$, hence

$$\frac{\partial z_k^{\ell+1}}{\partial a_r^\ell} = w_{kr}^{\ell+1}$$

And, because $a_r^\ell = \sigma(z_r^\ell)$ is applied entry-wise,

$$\frac{\partial a_r^\ell}{\partial z_j^\ell} = \begin{cases} \sigma'(z_j^\ell), & r = j, \\ 0, & r \neq j, \end{cases} \qquad \text{equivalently} \qquad \frac{\partial a_r^\ell}{\partial z_j^\ell} = \sigma'(z_j^\ell) \tag{4}$$

Putting (4) into (3) reveals exactly one surviving term, namely the case $r = j$:

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_r w_{kr}^{\ell+1} \left[\sigma'(z_j^\ell)\right] = w_{kj}^{\ell+1} \sigma'(z_j^\ell) \tag{5}$$

Feeding (5) back into (1), we arrive at

$$\delta_j^\ell = \sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_k \delta_k^{\ell+1} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_k \delta_k^{\ell+1} w_{kj}^{\ell+1} \sigma'(z_j^\ell) = \left(\sum_k w_{kj}^{\ell+1} \delta_k^{\ell+1}\right) \sigma'(z_j^\ell), \tag{6}$$

which is the component form of:

$$\boxed{\delta_j^\ell = \left(\sum_k w_{kj}^{\ell+1} \delta_k^{\ell+1}\right) \sigma'(z_j^\ell)}$$

If we now gather the equations for all j simultaneously, the column-wise sums in (6) assemble into a matrix–vector product with the transpose weights, while σ' acts element-wise:

$$\boxed{\boldsymbol{\delta}^\ell = \left(\mathbf{W}^{\ell+1}\right)^\top \boldsymbol{\delta}^{\ell+1} \odot \sigma'(\mathbf{z}^\ell)} \qquad (\ell = L-1, \ldots, 1)$$

The appearance of the transpose reflects that each δ_j^ℓ collects contributions along the *column* j of $\mathbf{W}^{\ell+1}$ and stacking those column sums for all j is exactly multiplication by $(\mathbf{W}^{\ell+1})^\top$.

Intuition. Take the next layer's error $\delta^{\ell+1}$ and push it *back* through the transpose weights to measure how much each hidden neuron influences downstream cost; then *gate* that signal by the local slope $\sigma'(z^\ell)$ at layer ℓ.

BP3: Gradient with respect to biases

An equation for the rate of change of the cost with respect to any bias in the network is:

$$\boxed{\frac{\partial C}{\partial b_j^\ell} \;=\; \delta_j{}^\ell, \text{ or } \frac{\partial C}{\partial \mathbf{b}^\ell} \;=\; \boldsymbol{\delta}^\ell}$$

Proof. Let the layer sizes be $n_{\ell-1} \to n_\ell$. Then,

$$\mathbf{W}^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \quad \mathbf{a}^{\ell-1} \in \mathbb{R}^{n_{\ell-1}}, \quad \mathbf{b}^\ell,\, \mathbf{z}^\ell,\, \boldsymbol{\delta}^\ell \in \mathbb{R}^{n_\ell}$$

For neuron k in layer ℓ,

$$z_k^\ell \;=\; \sum_{r=1}^{n_{\ell-1}} w_{kr}^\ell \, a_r^{\ell-1} \;+\; b_k^\ell$$

If we change b_j^ℓ slightly, *only* z_j^ℓ changes by the same amount; all other z_k^ℓ for $k \neq j$ do not change. Thus,

$$\frac{\partial z_k^\ell}{\partial b_j^\ell} \;=\; \begin{cases} 1, & k = j \\ 0, & k \neq j \end{cases} \tag{1a}$$

Applying the chain rule component-wise,

$$\frac{\partial C}{\partial b_j^\ell} = \sum_{k=1}^{n_\ell} \frac{\partial C}{\partial z_k^\ell} \frac{\partial z_k^\ell}{\partial b_j^\ell} = \underbrace{\frac{\partial C}{\partial z_j^\ell}}_{=\,\delta_j^\ell} \cdot 1 \;+\; \sum_{k \neq j} \frac{\partial C}{\partial z_k^\ell} \cdot 0 = \delta_j^\ell \tag{2a}$$

Stacking $j = 1, \ldots, n_\ell$ gives us the following:

$$\boxed{\frac{\partial C}{\partial \mathbf{b}^\ell} = \boldsymbol{\delta}^\ell}$$

Matrix–vector form: The pre-activation vector is

$$\mathbf{z}^\ell \;=\; \mathbf{W}^\ell \, \mathbf{a}^{\ell-1} \;+\; \mathbf{b}^\ell$$

A bias change shifts $\mathbf{z}^\ell$ component-wise by the same amount, so the Jacobian is the identity:

$$\frac{\partial \mathbf{z}^\ell}{\partial \mathbf{b}^\ell} \;=\; \mathbf{I}_{n_\ell} \tag{1b}$$

Using the multivariate chain rule (compose $C(\mathbf{z}^\ell, \ldots)$ with $\mathbf{z}^\ell(\mathbf{b}^\ell)$),

$$\boxed{\frac{\partial C}{\partial \mathbf{b}^\ell} \;=\; \left(\frac{\partial \mathbf{z}^\ell}{\partial \mathbf{b}^\ell}\right)^\top \frac{\partial C}{\partial \mathbf{z}^\ell} \;=\; \mathbf{I}_{n_\ell}^\top \, \boldsymbol{\delta}^\ell \;=\; \boldsymbol{\delta}^\ell} \tag{2b}$$

Summary. Component-wise: $\frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell$. Vector form: $\dfrac{\partial C}{\partial \mathbf{b}^\ell} = \boldsymbol{\delta}^\ell$. The identity Jacobian reflects that b_j^ℓ shifts only z_j^ℓ (not other units).

BP4: Gradient with respect to weights

An equation for the rate of change of the cost with respect to any weight in the network is

$$\boxed{\frac{\partial C}{\partial w^{\ell}_{jk}} \;=\; a^{\ell-1}_{k}\,\delta^{\ell}_{j}}$$

Equivalently, in matrix form (the outer product of "error at layer ℓ" and "inputs from layer $\ell-1$"):

$$\boxed{\frac{\partial C}{\partial \mathbf{W}^{\ell}} \;=\; \boldsymbol{\delta}^{\ell}\,(\mathbf{a}^{\ell-1})^{\top} \quad \in \mathbb{R}^{\,n_{\ell}\times n_{\ell-1}}}$$

This equation can be rewritten in a less-heavy notation as:

$$\boxed{\frac{\partial C}{\partial \mathbf{W}^{\ell}} = \; a^{\ell-1}_{\text{in}}\;\delta^{\ell}_{\text{out}},}$$

where it's understood that a^{l-1}_{in} is the activation of the neuron input to the weight from layer (ℓ-1), and δ^{ℓ}_{out} is the error of the neuron output from the weight in layer ℓ. With our convention of "rows = destination neurons, columns = source neurons," this is exactly the same as the outer product $\boldsymbol{\delta}^{\ell}(\mathbf{a}^{\ell-1})^{\top}$.

Proof. Let the layer sizes be $n_{\ell-1} \to n_{\ell}$. Then,

$$\mathbf{W}^{\ell} \in \mathbb{R}^{\,n_{\ell}\times n_{\ell-1}}, \quad \mathbf{a}^{\ell-1} \in \mathbb{R}^{\,n_{\ell-1}}, \quad \mathbf{b}^{\ell},\, \mathbf{z}^{\ell},\, \boldsymbol{\delta}^{\ell} \in \mathbb{R}^{\,n_{\ell}}$$

Consider a single weight w^{ℓ}_{jk} connecting the source unit k in layer $\ell-1$ to the destination unit j in layer ℓ. In the forward map,

$$z^{\ell}_{i} = \sum_{r=1}^{n_{\ell-1}} w^{\ell}_{ir}\, a^{\ell-1}_{r} + b^{\ell}_{i}, \qquad \delta^{\ell}_{i} = \frac{\partial C}{\partial z^{\ell}_{i}},$$

the weight w^{ℓ}_{jk} appears in z^{ℓ}_{j} only when $i = j$ and not in z^{ℓ}_{i} for $i \neq j$. Thus, when differentiating the cost with respect to w^{ℓ}_{jk} via the chain rule,

$$\frac{\partial C}{\partial w^{\ell}_{jk}} = \sum_{i=1}^{n_{\ell}} \frac{\partial C}{\partial z^{\ell}_{i}}\,\frac{\partial z^{\ell}_{i}}{\partial w^{\ell}_{jk}},$$

all terms except the $i = j$ term vanish, because z^{ℓ}_{i} does not depend on w^{ℓ}_{jk} for $i \neq j$. Focusing on z^{ℓ}_{j}, its dependence on w^{ℓ}_{jk} is linear and isolated through the product $w^{\ell}_{jk}a^{\ell-1}_{k}$, so

$$\frac{\partial z^{\ell}_{j}}{\partial w^{\ell}_{jk}} = a^{\ell-1}_{k}$$

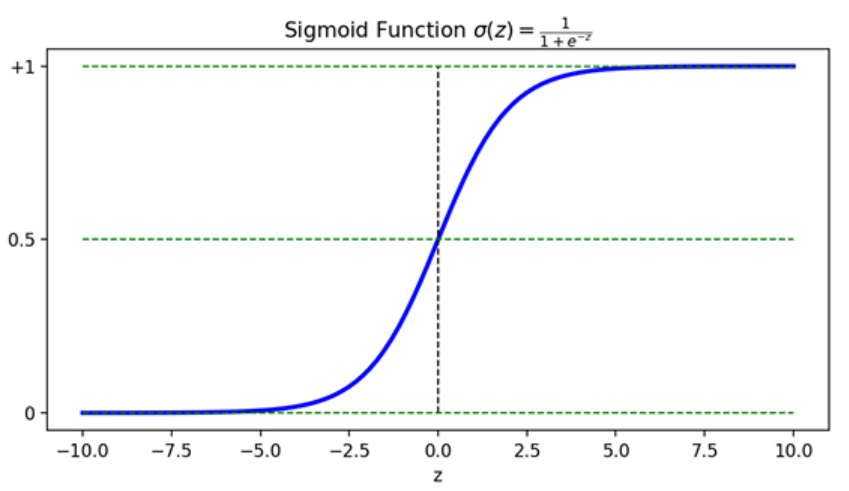
Substituting this and $\frac{\partial C}{\partial z^{\ell}_{j}} = \delta^{\ell}_{j}$ into the chain rule immediately gives the component result

$$\frac{\partial C}{\partial w^{\ell}_{jk}} = \delta^{\ell}_{j}\, a^{\ell-1}_{k}$$

Stacking these entries for all (j, k) simply assembles the outer product of the destination-layer error and the source-layer activations:

$$\boxed{\frac{\partial C}{\partial \mathbf{W}^\ell} = \boldsymbol{\delta}^\ell \left(\mathbf{a}^{\ell-1}\right)^\top} \quad \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$$

Intuitively, each weight's gradient is "how wrong the destination neuron is" times "how active the source neuron was"—if either factor is small, that weight learns slowly. In other words, if *input activation* is tiny ($a_k^{\ell-1} \approx 0$), then $\partial C/\partial w_{jk}^\ell \approx 0$. If *output neuron* (especially at the last layer with a sigmoid) is saturated ($\sigma'(z) \approx 0$ when $a \approx 0$ or 1), then δ^L is small and the final-layer weights/biases update slowly.



Mini-batch version (for updates). For a mini-batch $\{x_i\}_{i=1}^m$,

$$\frac{\partial C_{\text{batch}}}{\partial \mathbf{W}^\ell} = \frac{1}{m} \sum_{i=1}^m \boldsymbol{\delta}^\ell(x_i) \left(\mathbf{a}^{\ell-1}(x_i)\right)^\top, \qquad \frac{\partial C_{\text{batch}}}{\partial \mathbf{b}^\ell} = \frac{1}{m} \sum_{i=1}^m \boldsymbol{\delta}^\ell(x_i)$$

5 The Backpropagation Algorithm

By putting from BP1 to BP4 all together given an input x:

1. **Feedforward:** For $\ell = 1, \ldots, L$, compute $z^\ell = W^\ell a^{\ell-1} + b^\ell$ with $a^\ell = \sigma(z^\ell)$
2. **Output error:** Compute $\delta^L = \nabla_a C \odot \sigma'(z^L)$
3. **Backpropagate the error:** For $\ell = L-1, \ldots, 1$, compute $\delta^\ell = \left(\left(W^{\ell+1}\right)^\top \delta^{\ell+1}\right) \odot \sigma'(z^\ell)$
4. **Output:** Gradients of the cost w.r.t the weights and bias: $\dfrac{\partial C}{\partial w_{jk}^\ell} = a_k^{\ell-1}\, \delta_j^\ell, \quad \dfrac{\partial C}{\partial b_j^\ell} = \delta_j{}^\ell$

What Backpropagation means in a feedforward Neural Network Backpropagation computes a sequence of *error vectors* δ^ℓ going *backward* through the network, starting from the output layer. The backward direction is not arbitrary: By design, the cost C is a function of the network *outputs*. To understand how earlier weights and biases influence C, we repeatedly apply the chain rule from the outputs back to the inputs.
Concretely, the output-layer error is

$$\delta^L = \nabla_{a^L} C \ \odot \ \sigma'(z^L) \quad \text{(BP1)},$$

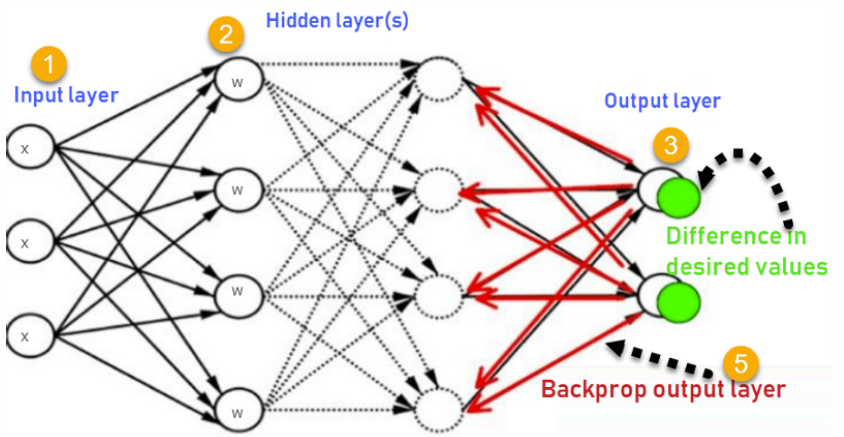
and for hidden layers, we "pull" the error back through the transpose weights and gate it by the local slope:

$$\delta^\ell = \left(\left(W^{\ell+1}\right)^\top \delta^{\ell+1}\right) \odot \sigma'(z^\ell) \qquad (\ell = L-1, \ldots, 1) \quad \text{(BP2)}$$

Once the δ are known, the gradients fall out immediately:

$$\frac{\partial C}{\partial b_j^\ell} = \delta_j{}^\ell \quad \text{(BP3)}, \qquad \frac{\partial C}{\partial w_{jk}^\ell} = a_k^{\ell-1}\, \delta_j^\ell \quad \text{(BP4)}$$

In one *epoch*, each training sample makes exactly one forward pass to produce activations and one backward pass to propagate errors; these two passes give you all partial derivatives needed for learning.



Why backprop is efficient A naive way to get derivatives is to use perturbation-based methods according to the definitions of derivatives for each parameter and re-evaluate the cost,

$$\frac{\partial C}{\partial w_j} \approx \frac{C\left(w + \epsilon e_j\right) - C(w)}{\epsilon},$$

However, this can be very expensive because we need to evaluate the ANN many times, as it contains millions of parameters. With millions of weights, that's prohibitively expensive. Instead, Backpropagation is clever because it delivers *all* partial derivatives $\frac{\partial C}{\partial w_{jk}^\ell}$ and $\frac{\partial C}{\partial b_j^\ell}$ with just *one* forward pass (to compute z^ℓ, a^ℓ) and *one* backward pass (to compute δ^ℓ). Roughly

speaking, the backward pass costs about the same as the forward pass, so the total work is comparable to *two* forwards—not millions. This practical speed-up, fully appreciated in 1986 (Rumelhart–Hinton–Williams), dramatically expanded the scale of problems neural networks could tackle.

Backpropagation with SGD In practice, we combine backpropagation with stochastic gradient descent (SGD). An outer loop walks through epochs; within each epoch, we iterate over *mini-batches* of size 'm'.

Per sample (inside the mini-batch). Given an input x,

$$z^\ell = W^\ell a^{\ell-1} + b^\ell, \qquad a^\ell = \sigma(z^\ell) \quad (\ell = 1, \dots, L)$$

At the output layer,

$$\delta^{x,L} = \nabla_a C_x \ \odot \ \sigma'(z^{x,L}),$$

and for hidden layers,

$$\delta^{x,\ell} = (W^{\ell+1})^\top \delta^{x,\ell+1} \ \odot \ \sigma'(z^{x,\ell}) \qquad (\ell = L-1, \dots, 1)$$

Accumulate and update (once per mini-batch). Average the per-sample gradients and take a step:

$$\Delta W^\ell \ = \ \frac{\eta}{m} \sum_{i=1}^{m} \delta^{x_i,\ell} \left(a^{x_i,\ell-1}\right)^\top, \qquad \Delta b^\ell \ = \ \frac{\eta}{m} \sum_{i=1}^{m} \delta^{x_i,\ell},$$

$$W^\ell \leftarrow W^\ell - \Delta W^\ell, \qquad b^\ell \leftarrow b^\ell - \Delta b^\ell \qquad (\ell = L, \dots, 1)$$

Then, move on to the next mini-batch. After all mini-batches are processed, the epoch ends—every training example has gone *forward once* and *backward once*.

6 Summary

Forward

$$z^\ell = W^\ell a^{\ell-1} + b^\ell, \quad a^\ell = \sigma(z^\ell), \quad \sigma \text{ entrywise}, \quad \odot = \text{Hadamard}$$

Assumptions

$$C = \frac{1}{n} \sum_x C_x, \qquad C_x = \tfrac{1}{2} \|y(x) - a^L(x)\|^2 = \tfrac{1}{2} \sum_i (y_i - a_i^L)^2$$

Backprop equations

$$\delta^L = \nabla_a C \odot \sigma'(z^L), \quad \delta^\ell = \left(W^{\ell+1}\right)^\top \delta^{\ell+1} \odot \sigma'(z^\ell), \quad \frac{\partial C}{\partial b^\ell} = \delta^\ell, \quad \frac{\partial C}{\partial W^\ell} = \delta^\ell (a^{\ell-1})^\top$$

vfill

This is a posting that I summarized with study-purpose and is adapted from lecture notes of NE-795(Scientific Machine Learning), given by professor Xu Wu, NC State University.