A Guided Derivation of Backpropagation Algorithm

An Intuitive Introduction and Step-by-Step Mathematical Derivation
From Algebra to code

Author: Youssef AMAYED

Engineering student at Sup'Com (Higher School of Communications, Tunis)
youssef.amayed@supcom.tn

Contents

Abstract

This paper presents a rigorous step-by-step derivation of the backpropagation algorithm, starting from the basic principles of calculus and linear algebra. While backpropagation is often described either too abstractly or too mechanically, this paper seeks a balance: deeply intuitive, yet mathematically detailed. Readers are guided through gradients, Jacobians, and tensor contraction, culminating in a clean and general backpropagation formula. Although motivated by neural networks, the derivation highlights that backpropagation is a general method for computing derivatives of nested functions. No prior experience with machine learning is assumed only a willingness to follow the math.

Note: This paper also includes a visual and intuitive derivation of backpropagation, guided step by step, and accessible to motivated beginners *Note:* This paper also includes a visual and intuitive derivation of backpropagation, guided step by step, and accessible to motivated beginners.

Introduction

Backpropagation is the mathematical backbone of modern learning algorithms especially in neural networks.But too often, it's either presented as a high-level intuition without substance, or buried in index-heavy summations with no big picture. This paper aims to do something different: to build backpropagation from the ground up, slowly and visually, starting from the very idea of computing derivatives of composite functions.

I'm not an expert in machine learning. I'm a beginner, an engineering student, and someone who fell in love with math and its connection to intelligence.When I started reading about backpropagation, I couldn't find the paper I needed: one that was mathematically honest yet still beginner-friendly. So I decided to write it.

This paper isn't a deep dive into neural networks. In fact, we only touch them lightly, just enough to explain why backpropagation matters. Our focus is on the algorithm itself as a general and beautiful mathematical tool.

If you're new, that's okay. I've done my best to make every step visual, clean, and explainable. You just need to follow through and maybe take some notes along the way .

Note: A full implementation of the backpropagation algorithm, including the final sigmoid-based example in this paper, is available in my public GitHub repository:

`github.com/Ka3baAnanas/mnist-digit-recognizer`

The code is built from scratch using NumPy and follows the mathematical derivation presented here line by line. If you're curious to see how the theory translates into practice or want to train a simple digit recognizer on the MNIST dataset this repository is a perfect place to explore.

Motivation and Personal Perspective

Who is the author?

The author of this paper is not a professor or a machine learning expert. I'm an engineering student and a musician someone who entered the field of machine learning through a genuine curiosity

My path into machine learning was sparked by an idea: to apply it to music. That's what pulled me in. And what kept me here was the beauty of the mathematics especially when it's made intuitive and visual.

The curiosity strongly nurtured during my years in the Tunisian-French preparatory system for engineering schools (*classes préparatoires*), where mathematics is taught with intensity and rigor. That background gave me a solid foundation in calculus, linear algebra,formal reasoning and a passion for understanding things from first principles.

What this paper is ,and what it's not

We will not dive into deep learning or neural network architectures. We will briefly introduce enough context so that readers unfamiliar with neural nets are able to follow. But our focus is squarely on the mathematics of backpropagation algorithm and how it works.

We want to show that this algorithm can be derived from first principles and in a way that makes its recursive structure visible, understandable, and elegant.

Why this paper might help you

If you're a beginner especially one who learns visually, or likes to build understanding from the ground up this paper is for you.

And it's designed to be a guide, not a lecture.

Let's begin.

I The Role of Backpropagation:

The backpropagation algorithm aims to compute the derivative of a function composed of many nested functions:

$$\frac{\partial}{\partial W_l} C(f_L(f_{L-1}(f_{L-2}(\dots f_l(W_l)\dots))))$$

This derivative is the core of the **backpropagation algorithm**, and importantly, **it is not specific to neural networks or machine learning**. That is a widespread misconception. Backpropagation is a general recursive method for computing derivatives of composite functions.

Example use case beyond neural networks: In physics-informed neural networks (PINNs), where physical constraints (like differential equations) are embedded in the loss function, backpropagation is used to compute gradients of complex nested functions derived from PDEs.

To compute the derivative of a composite function with respect to W_l, we use a chain rule expression adapted to our nested functions:

$$\frac{\partial C}{\partial W_l} = \frac{\partial C}{\partial f_L} \cdot \frac{\partial f_L}{\partial f_{L-1}} \cdot \frac{\partial f_{L-1}}{\partial f_{L-2}} \cdots \frac{\partial f_{l+1}}{\partial f_l} \cdot \frac{\partial f_l}{\partial W_l}$$

This expresses how the gradient with respect to W_l propagates through each nested layer of the function composition.

We will later transition from this scalar notation to matrix calculus and vector-Jacobian products. For simplicity and intuition, we keep the scalar version here.

II Backpropagation in Neural Networks

II.1 Neural Networks

A Single Neuron

A neuron takes a vector of inputs (activations from the previous layer), performs a weighted sum, adds a bias, and applies a non-linear activation function. For input vector a^{l-1}, weights W^l, and bias b^l, the output activation is:

$$z^l = W^l a^{l-1} + b^l \quad \text{then} \quad a^l = \sigma(z^l)$$

Here, σ is typically a sigmoid function:

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

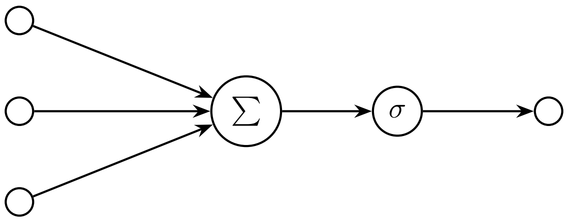


Figure 1: A single neuron applying a weighted sum and activation.

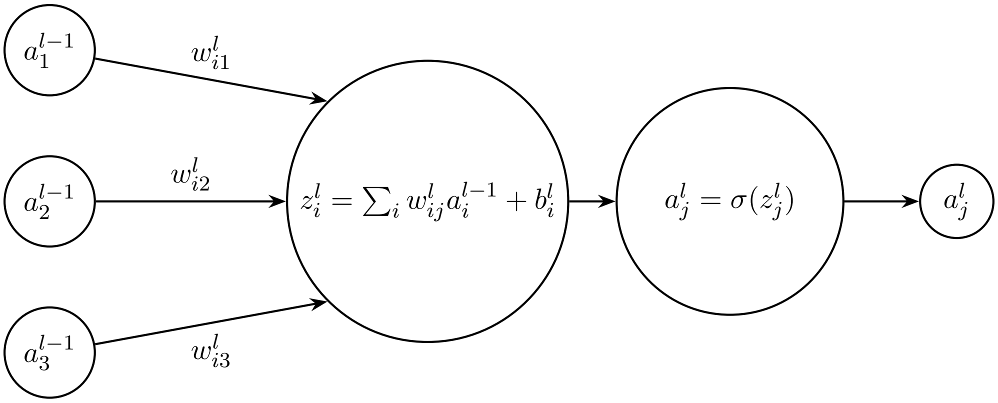


Figure 2: Clean view of a single neuron: input activations are weighted, summed with bias, and passed through an activation function.

The Neural Network

A neural network is a stack of such layers, where each layer applies a linear transformation followed by a non-linear activation function. For any layer l, the transformation is:

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

where:

- a^{l-1} is the input vector (activations from the previous layer),
- W^l is the weight matrix connecting layer $l-1$ to layer l,
- b^l is the bias vector for layer l,
- σ is an activation function (e.g., sigmoid, ReLU),
- a^l is the output activation of the current layer.

Understanding the components:

- **Weights** W^l: A matrix of parameters where each entry w_{ij}^l controls how much the j-th neuron in layer $l-1$ influences the i-th neuron in layer l.

  $$W^l = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots & w_{1n}^l \\ w_{21}^l & w_{22}^l & \cdots & w_{2n}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1}^l & w_{m2}^l & \cdots & w_{mn}^l \end{bmatrix}$$

  Here, n is the number of neurons in layer $l-1$ and m is the number of neurons in layer l.

- **Biases** b^l: A vector of offsets added to each neuron's weighted input, allowing flexibility:

  $$b^l = \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_m^l \end{bmatrix}$$

- **Activation function** σ: A non-linear function applied element-wise to the linear combination $W^l a^{l-1} + b^l$, enabling the network to model complex, non-linear relationships:

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

Note: The notation w_{ij}^l might seem non-intuitive at first, but we use it this way because it aligns perfectly with matrix multiplication conventions: the weight at row i, column j connects the j-th neuron in the previous layer to the i-th neuron in the current layer. This makes the calculation of $W^l a^{l-1}$ straightforward and consistent with linear algebra rules.

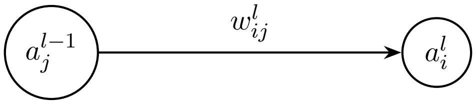


Figure 3: Connection weight w_{ij}^l from input neuron j in layer $l-1$ to output neuron i in layer l.

Example: MNIST Digit Classifier

A typical architecture for digit recognition using the MNIST dataset is as follows:

Each neuron in the output layer corresponds to a digit label. The network outputs an activation vector:

$$a^L = [a_0^L, a_1^L, \ldots, a_9^L]$$

The predicted digit is the index k for which a_k^L is the largest:

$$\text{Prediction} = \arg\max_k a_k^L$$

The neuron with the highest activation is taken as the predicted digit.

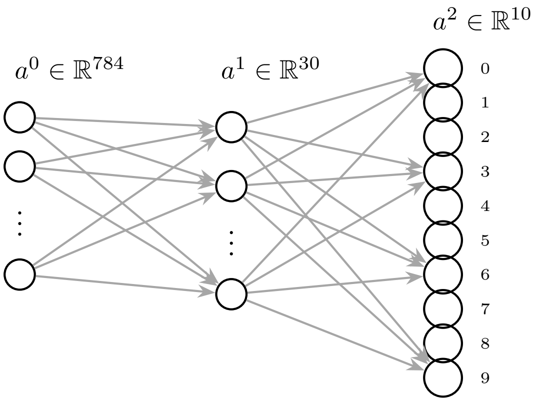

$a^2 \in \mathbb{R}^{10}$
$a^0 \in \mathbb{R}^{784}$
$a^1 \in \mathbb{R}^{30}$
0
1
2
3
4
5
6
7
8
9

Figure 4: MNIST Neural Network: 784 input neurons (abstracted), 30 hidden neurons (abstracted), 10 fully drawn output neurons labeled 0–9.

In a neural net for digit recognition (MNIST), we have: The input layer of the network contains neurons encoding the values of the input pixels

- Input: 784 neurons (28x28 pixels)

- Hidden layer: e.g. 30 neurons
- Output layer: 10 neurons, each representing a digit from 0 to 9

The prediction is read by checking which output neuron has the highest activation.

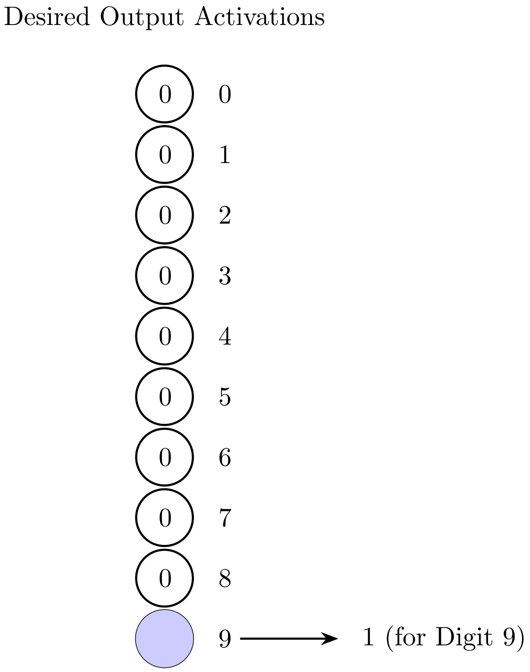


Figure 5: Desired output activations with target output for digit “9”: one-hot encoded target has a 1 at position 9, 0 elsewhere.

II.2 The Cost Function and Its Role

The cost function $C(a^L, y)$ measures how far the output a^L is from the desired output y. For example, in digit recognition, if the input is a ”9”, the correct output vector y would have a 1 in the 10th position and 0 elsewhere.

We can use the **quadratic cost function**:

$$C = \frac{1}{2}\|a^L - y\|^2$$

Why Minimize the Cost? The cost function tells us how far our network's output is from the correct result. So, **learning** is just minimizing C by adjusting a^L (eventually adjusting the weights and biases as we will see).

To minimize it, we use **gradient descent**, a general algorithm to reduce a function by moving opposite the gradient.

Recursive Dependence of the Cost At first glance, we might want to minimize C with respect to the output activation a^L, since the cost depends directly on it:

$$C = C(a^L)$$

But we quickly realize that a^L itself depends on a^{L-1}, which depends on a^{L-2}, and so on, down to a^l, which depends on W^l. This recursive dependency means that minimizing C with respect to a^L implies we must understand how every earlier activation and weight contributes to it.

Hence, minimizing C with respect to weights like W^l becomes equivalent to:

$$\frac{\partial}{\partial W_l} C(f_L(f_{L-1}(f_{L-2}(\dots f_l(W_l)\dots))))$$

Which brings us back to the **chain rule** ;the reason we use the **backpropagation algorithm**.

II.3 Gradient Descent Overview

Once we compute the gradient of the cost function with respect to each parameter, the natural next step is to update those parameters to reduce the cost. This is where the **gradient descent algorithm** comes in.

An Intuitive Example

Let's imagine that the cost function $C(\mathbf{u})$ depends on a vector of parameters $\mathbf{u}$, which could be weights and biases in a network. Let's say we are currently at some point $\mathbf{u}_1$ in parameter space.

We compute the gradient $\nabla C(\mathbf{u}_1)$, which points in the direction of steepest *increase* of the cost. To reduce the cost, we move in the *opposite* direction:

$$\mathbf{u}_1' = \mathbf{u}_1 - \eta \nabla C(\mathbf{u}_1)$$

Here:

- $\mathbf{u}_1$ is the current point in parameter space,
- $\nabla C(\mathbf{u}_1)$ is the gradient vector at that point,
- η is the learning rate,
- $\mathbf{u}_1'$ is the updated parameter vector after one gradient descent step.

Because the gradient points uphill and we move downhill, this update leads to:

$$C(\mathbf{u}_1') < C(\mathbf{u}_1)$$

This is the essence of gradient descent: it gives us a simple way to find a direction that guarantees (at least locally) that the cost function decreases.

"Follow the slope downward and each step takes you to a better solution."

Even though the function C might be complex, possibly high-dimensional and non-linear, the gradient gives us the best local linear approximation for descending. Over many steps, this process slowly leads us toward a (local) minimum of the cost function.

The Idea

Gradient descent is an optimization technique that iteratively moves the parameters (weights and biases) in the direction that most rapidly decreases the cost function. Mathematically, it updates each parameter θ using:

$$\theta \leftarrow \theta - \eta \frac{\partial C}{\partial \theta}$$

Where:

- θ is a parameter (like W^l or b^l),
- $\eta > 0$ is the **learning rate**, controlling the step size,
- $\frac{\partial C}{\partial \theta}$ is the gradient of the cost with respect to that parameter.

In a neural network each layer l has its own weights W^l and biases b^l. The update rule for these parameters becomes:

$$W^l \leftarrow W^l - \eta \frac{\partial C}{\partial W^l}, \quad b^l \leftarrow b^l - \eta \frac{\partial C}{\partial b^l}$$

These gradients are computed using the backpropagation algorithm.

Visualizing Gradient Descent

Imagine the cost function as a surface over parameter space. At any point (i.e., for a specific set of weights), the gradient vector points in the direction of steepest increase. Gradient descent moves us in the opposite direction, downhill on this surface.

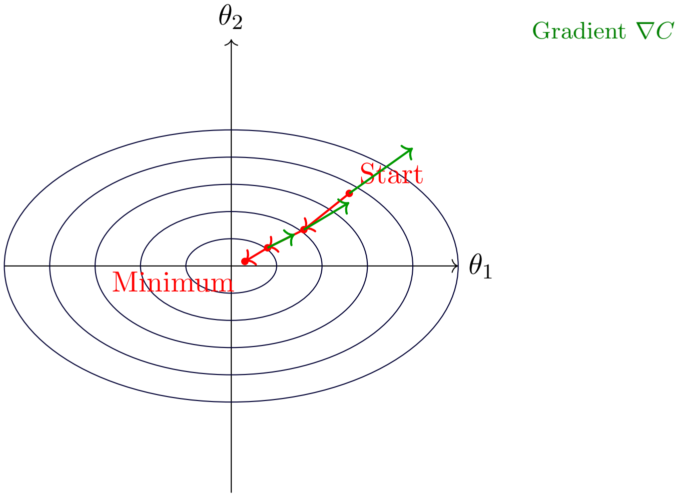


Figure 6: Gradient descent on a 2D cost surface. Red arrows show the descent path toward the minimum. Green arrows show the local gradients at each point (pointing uphill).

Choosing the Learning Rate

The learning rate η is critical:

- If η is too small: training is slow, and we may get stuck in local minima.
- If η is too large: updates may overshoot and cause divergence.

Often, modern training uses **adaptive learning rates** or more advanced optimizers (like Adam or RMSProp), but the core idea remains based on gradient descent.

Variants

- **Batch Gradient Descent:** Computes gradient over the entire dataset.
- **Stochastic Gradient Descent (SGD):** Updates using only one training example at a time.
- **Mini-Batch Gradient Descent:** A compromise — uses a small batch (e.g., 32 samples) per update.

III An Intuitive Backpropagation prespective: Step-by-Step Flow of Information

We aim to compute:

$$\frac{\partial C(a^L(a^{L-1}(\dots a^l(w^l))))}{\partial w^l}$$

That is, we need to find the derivative of the cost function C with respect to the weights represented by w^l in the network.

The Intuitive Idea

Let's break down the process step-by-step.

The first idea is to apply the **chain rule**, which is the natural way we compute derivatives of nested functions. At first glance, this might seem like:

$$\frac{\partial C}{\partial w^l} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial a^{L-1}} \cdots \frac{\partial a^2}{\partial a^l} \cdot \frac{\partial a^l}{\partial w^l}$$

It's the corret way of thinking and almost the correct answer. But here's the catch: this isn't quite right because these are not scalars. **Activations** like a^ℓ are **vectors**, and w^ℓ is a **matrix**. So, we need to carefully account for how these quantities interact at each layer.

Start Simple

For clarity and for the purpose of this part, suppose there's no difference. That is, the chain rule works as usual:

$$\frac{\partial C}{\partial w^l} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial a^{L-1}} \cdots \frac{\partial a^1}{\partial w^l}$$

This is **straightforward**, we'll revisit the chain rule later with more structured tools (like Jacobians and matrix operations), but let's keep the intuition a priority here.

Note: We cannot stay in the scalar world forever. If we tried to write everything in terms of scalar components (e.g., component i of vector a^ℓ, and component (i, j) of matrix w^l), we'd end up with complicated summation expressions like:

$$\sum_{m,n,\dots} \frac{\partial C}{\partial a^L_m} \cdot \frac{\partial a^L_m}{\partial a^{L-1}_n} \cdots \frac{\partial a^{l+1}_k}{\partial a^l_i} \cdot \frac{\partial a^l_i}{\partial w^l_{ij}}$$

This becomes hard to read and reason about. It hides the beauty and structure of the backpropagation algorithm. So, we will use vectors and matrices. It's cleaner, more insightful, and actually easier to manipulate once we introduce a few key rules.

Moving backward Through the Network,the start

Now let's think of the **real computation** as we move through the network.

At the **output layer** L, we first calculate how the **cost** C depends on the output activations a^L. For example, with a quadratic cost function:

$$C = \frac{1}{2}\|a^L - y\|^2 \quad \Rightarrow \quad \frac{\partial C}{\partial a^L} = (a^L - y)$$

This term tells us how the **error at the output layer** affects the overall cost. At this point, we have **the first piece of information** we need but we are not able to calculate the gradient with respect to the weights.To do that, we will need to propagate the error **backward through the network**.

Continuing Backward: Each Layer's Contribution

We now need to calculate how the **error at** a^L propagates backward to earlier layers. Let's see how the activations a^L depend on a^{L-1}, and so on.

At layer L, we have:

$$a^L = \sigma(z^L), \quad z^L = w^L a^{L-1} + b^L$$

To move backward, we calculate how each activation a^ℓ depends on the previous one. For instance:

$$\frac{\partial a^L}{\partial a^{L-1}} = \sigma'(z^L) \cdot w^L$$

This tells us how **activations at layer** L change with respect to those at $L-1$. We then move to the next layer, calculating each $\frac{\partial a^{L-1}}{\partial a^{L-2}}$, and so on, until we reach layer l.

Step-by-Step Backpropagation

At each stage, we find that the error at one layer depends on:

- The **weights** between layers
- The **activation function's derivative**

Thus, as we move through the layers, we calculate the required derivatives and piece them together. So far, we've been gathering the necessary information **from the last layer backwards**

Finally, Calculate the Gradient at Layer l

Once we have all the required terms, we finally reach layer l. At this point, we can compute the derivative of the cost function with respect to the weight w^l.

We need to calculate:

$$\frac{\partial C}{\partial w^l} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial a^{L-1}} \cdot \cdots \cdot \frac{\partial a^l}{\partial w^l}$$

The term $\frac{\partial a^l}{\partial w^l}$ is easy to compute:

$$\frac{\partial a^l}{\partial w^l} = \sigma'(z^l) \cdot a^{l-1}$$

Thus, the gradient for w^l becomes:

$$\frac{\partial C}{\partial w^l} = \delta^l \cdot a^{l-1}$$

where δ^l is the **error term at layer** l, which we now compute as:

$$\delta^l = \frac{\partial C}{\partial a^L} \cdot \prod_{k=l+1}^{L} \frac{\partial a^k}{\partial a^{k-1}}$$

But here's the **beauty** of backpropagation: we don't need to compute δ^l directly. It **emerges naturally** as a recursive function of the subsequent layer's errors δ^{l+1}.

Thus, each δ^l depends on δ^{l+1}, and the process works **backward**, from the output layer to the input, **propagating the error** in each layer.

Conclusion: The Power of Backpropagation

We've now walked through the entire backpropagation process:

- **First**, we gather the necessary information from the **output layer**.
- **Then**, we propagate the error **backward** layer by layer, calculating the required terms and **gradients** at each step.
- **Finally**, we update the weights by using the computed gradient at each layer.

This is how backpropagation works:**moving forward to gather information, then moving backward to adjust the weights**, using the errors propagated through the network.

IV The formal and mathematical derivation of the backpropagation algorithm

IV.1 Vector and Matrix Derivatives: Preparing the Groundwork

We saw that in neural networks, we are dealing with expressions like:

$$\frac{\partial C\left(a^L(a^{L-1}(\ldots a^l(w^l)\ldots))\right)}{\partial w^l}$$

And here is the issue:

- C is a scalar function,
- $a^L, a^{L-1}, \ldots$ are vectors,
- w^l is a matrix.

So... what does it even mean to differentiate a scalar with respect to a matrix when everything in between involves vectors?

Before we manipulate such expressions, we need to be precise about what kind of functions we're dealing with, and how their derivatives are defined.

Types of Functions and Variables
We can have:

- Scalar-valued functions: $f : \mathbb{R}^n \to \mathbb{R}$
- Vector-valued functions: $f : \mathbb{R}^n \to \mathbb{R}^m$

And the inputs themselves (the variables) can also be:

- Scalars,
- Vectors,
- Matrices.

So depending on what the input and output are, the derivative will have different shapes and interpretations.

Let's build up step-by-step.

1. Scalar Function of a Scalar Variable: This is basic:

$$f(x) = x^2 \quad \Rightarrow \quad \frac{df}{dx} = 2x$$

2. Scalar Function of a Vector Variable: Let $C : \mathbb{R}^n \to \mathbb{R}$, with $v = (x_1, x_2, \ldots, x_n)$. Then:

$$\frac{\partial C}{\partial v} = \nabla C = \begin{pmatrix} \frac{\partial C}{\partial x_1} \\ \frac{\partial C}{\partial x_2} \\ \vdots \\ \frac{\partial C}{\partial x_n} \end{pmatrix}$$

This is the gradient a column vector.
Example: Let $a^L \in \mathbb{R}^m$, and suppose:

$$C = C(a^L), \quad \text{then} \quad \frac{\partial C}{\partial a^L} = \begin{pmatrix} \frac{\partial C}{\partial a^L_1} \\ \frac{\partial C}{\partial a^L_2} \\ \vdots \\ \frac{\partial C}{\partial a^L_m} \end{pmatrix}$$

3. Vector Function of a Vector Variable: Now let $f : \mathbb{R}^n \to \mathbb{R}^m$ and $x = (x_1, x_2, \ldots, x_n)$. That means:

$$f(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_m(x) \end{pmatrix}$$

Each f_i is a scalar function of the same input vector $x \in \mathbb{R}^n$.
Then the derivative is the **Jacobian matrix**:

$$\frac{\partial f}{\partial x} = J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

This generalizes the gradient it tells how each component of the output vector changes according to every input component.

Now: What about Matrix Variables?
Let us now consider the kind of function we see in neural nets:

$$f : \mathbb{R}^{m \times n} \to \mathbb{R}$$

So f is a scalar function of a matrix.

The derivative is then defined as:

$$\frac{\partial f}{\partial W} = \begin{pmatrix} \frac{\partial f}{\partial w_{11}} & \cdots & \frac{\partial f}{\partial w_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial w_{m1}} & \cdots & \frac{\partial f}{\partial w_{mn}} \end{pmatrix} \in \mathbb{R}^{m\times n}$$

That is: we differentiate the scalar function with respect to each component of the matrix — and the result is again a matrix.

Next step: what if the function is a vector? What if:

$$f : \mathbb{R}^{m\times n} \to \mathbb{R}^k$$

...and that will be our next challenge.

Alright,so what if the function itself is a vector?

Let us say that now we have:

$$f : \mathbb{R}^{m\times n} \to \mathbb{R}^k$$

That is: the function takes in a matrix, and gives a vector as output:

$$f(W) = \begin{pmatrix} f_1(W) \\ f_2(W) \\ \vdots \\ f_k(W) \end{pmatrix}$$

What is $\frac{\partial f}{\partial W}$?

This is where the word *tensor* comes in and it's not as scary as it sounds.

Let's break it down.

Each component function $f_i(W)$ is a scalar function of the matrix W. So we can take its derivative just like before:

$$\frac{\partial f_i}{\partial W} \in \mathbb{R}^{m\times n}$$

Now since there are k of them we just **stack these matrices** the one after the other. That gives us a cube-like object with shape:

$$\frac{\partial f}{\partial W} \in \mathbb{R}^{k\times m\times n}$$

This is a **rank-3 tensor**.

*Note:*A matrix is a rank-2 tensor,avector is a rank-1 tensor

You can think of it as a 3D stack of k matrices, one for each output component.

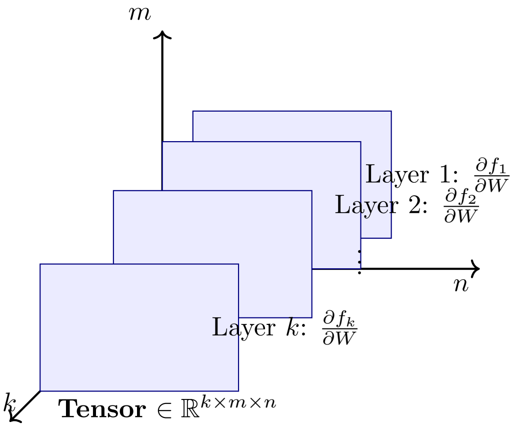

m
Layer 1: $\frac{\partial f_1}{\partial W}$
Layer 2: $\frac{\partial f_2}{\partial W}$
$\vdots$
n
Layer k: $\frac{\partial f_k}{\partial W}$
k
Tensor $\in \mathbb{R}^{k\times m\times n}$

Figure 7: Visualizing $\frac{\partial f}{\partial W}$ as a 3D stack of matrices. Each "layer" corresponds to one output component f_i.

Each individual matrix is:

$$\frac{\partial f_i}{\partial W} = \begin{pmatrix} \frac{\partial f_i}{\partial w_{11}} & \cdots & \frac{\partial f_i}{\partial w_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_i}{\partial w_{m1}} & \cdots & \frac{\partial f_i}{\partial w_{mn}} \end{pmatrix}$$

So again:

- The **output** of $\frac{\partial f}{\partial W}$ is a cube-shaped object with shape $k \times m \times n$
- It holds all the information: how each output component f_i depends on each entry of the matrix W

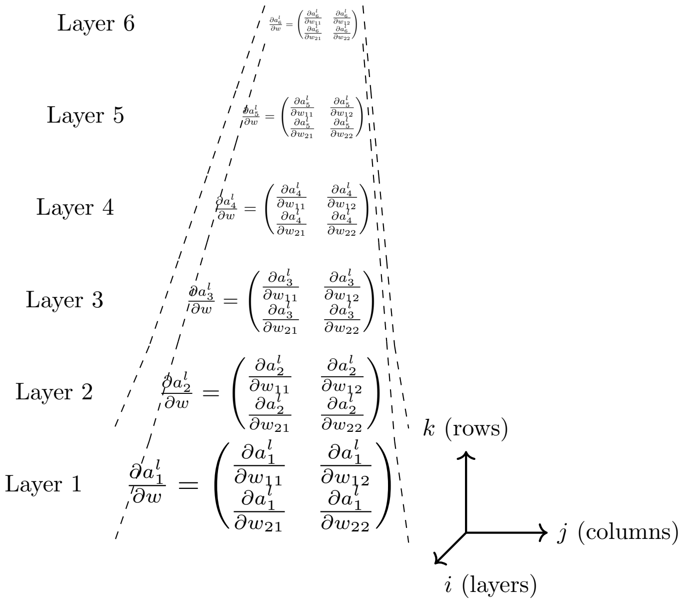
Summary:

- Derivative of a scalar wrt vector ⇒ vector (the gradient)
- Derivative of a scalar wrt matrix ⇒ matrix
- Derivative of a vector wrt matrix ⇒ tensor (a stack of matrices)

And that's all a tensor is in this context,just a 3D array that contains all the derivatives you care about.

Nothing scary. Just the natural next step.

Note: In our case, the function f corresponds to a^l , and the components of a^l are the individual activations a^l_i, which correspond to the different f_i .


Layer 6
Layer 5
Layer 4
Layer 3
Layer 2
Layer 1
$\frac{\partial a^l_1}{\partial w} = \begin{pmatrix} \frac{\partial a^l_1}{\partial w_{11}} & \frac{\partial a^l_1}{\partial w_{12}} \\ \frac{\partial a^l_1}{\partial w_{21}} & \frac{\partial a^l_1}{\partial w_{22}} \end{pmatrix}$
k (rows)
j (columns)
i (layers)

Each matrix is a **slice of the tensor**. The whole cube forms a tensor T with components:

$$T_{ijk} = \frac{\partial a^l_i}{\partial w_{jk}}$$

Where: - i: which activation neuron (layer index), - j: row index of weight, - k: column index of weight.

IV.2 The Chain Rule for Vector-Valued Functions: Step by Step

Alright now that we've seen how to differentiate with respect to vectors and matrices, let's bring things together.

We want to understand how to compute something like:

$$\frac{\partial C(y(x))}{\partial x}$$

Where:

- C is a scalar function,
- $y : \mathbb{R}^n \to \mathbb{R}^m$ is a vector-valued function,
- and $x \in \mathbb{R}^n$.

At first glance, this might look intimidating. But let's slow down and recall what we already know.

We're taking the derivative of a scalar with respect to a vector. That's just... the gradient!

So we write:

$$\frac{\partial C(y(x))}{\partial x} = \nabla_x C = \begin{pmatrix} \frac{\partial C}{\partial x_1} \\ \frac{\partial C}{\partial x_2} \\ \vdots \\ \frac{\partial C}{\partial x_n} \end{pmatrix}$$

But we can also express this using the intermediate variable y. Since:

$$C = C(y_1(x), y_2(x), \ldots, y_m(x)),$$

we're really dealing with a composition of functions.

So apply the multivariable chain rule:

$$\frac{\partial C}{\partial x_j} = \sum_{i=1}^{m} \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_j}$$

Stack these n rows together and we get:

$$\nabla_x C = \begin{pmatrix} \sum_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_1} \\ \sum_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_2} \\ \vdots \\ \sum_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_n} \end{pmatrix}$$

Or, more compactly, as a matrix product:

$$\nabla_x C = \left(\frac{\partial y}{\partial x}\right)^{\top} \cdot \nabla_y C$$

That's the chain rule for vector-valued functions.

- $\nabla_y C \in \mathbb{R}^m$ is the gradient of C with respect to y,
- $\frac{\partial y}{\partial x} \in \mathbb{R}^{m \times n}$ is the Jacobian of y with respect to x,
- So the transpose $\left(\frac{\partial y}{\partial x}\right)^{\top} \in \mathbb{R}^{n \times m}$ fits perfectly to give a result in $\mathbb{R}^n$.

To sum up:

$$\boxed{\frac{\partial C(y(x))}{\partial x} = \left(\frac{\partial y}{\partial x}\right)^{\top} \cdot \frac{\partial C}{\partial y}} \quad \text{(Chain rule for vectors)}$$

So clean and powerful and will be the key to backpropagation.

Wait what if the variable is a matrix?

Okay now let's take the next step.

We want to compute:

$$\frac{\partial C(y(W))}{\partial W} \quad \text{where} \quad W \in \mathbb{R}^{m \times n}, \quad y : \mathbb{R}^{m \times n} \to \mathbb{R}^k$$

At first glance, this might feel like it came out of nowhere. But actually, we've been preparing for this the whole time. So let's take it one step at a time.

Step 1: Differentiate a scalar with respect to a matrix.

We already saw earlier:

$$\frac{\partial C}{\partial W} = \begin{pmatrix} \frac{\partial C}{\partial w_{11}} & \frac{\partial C}{\partial w_{12}} & \cdots & \frac{\partial C}{\partial w_{1n}} \\ \frac{\partial C}{\partial w_{21}} & \frac{\partial C}{\partial w_{22}} & \cdots & \frac{\partial C}{\partial w_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial w_{m1}} & \frac{\partial C}{\partial w_{m2}} & \cdots & \frac{\partial C}{\partial w_{mn}} \end{pmatrix}$$

Simple: just compute the partial derivative of C with respect to each entry of W.

Step 2: Our case the scalar C depends on a vector y, and that vector depends on the matrix W.

So we have a composition:

$$C(y(W)) \quad \text{with} \quad y = \begin{pmatrix} y_1(W) \\ y_2(W) \\ \vdots \\ y_k(W) \end{pmatrix}$$

And we want to compute:

$$\frac{\partial C(y(W))}{\partial W}$$

Let's not panic. We'll use the full matrix definition like before let's just look at the components again:

$$\left[\frac{\partial C(y(W))}{\partial W}\right]_{jk} = \frac{\partial C(y(W))}{\partial w_{jk}}$$

That's what we want.

Step 3: Think about that component.

How does C depend on w_{jk}? Well indirectly, through the vector y, which itself depends on W. So we apply the chain rule but this time, for scalars:

$$\frac{\partial C(y(W))}{\partial w_{jk}} = \sum_{i=1}^{k} \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{jk}}$$

This is the typical scalar chain rule. Nothing new.

And now we can put this back into the full matrix:

$$\frac{\partial C(y(W))}{\partial W} = \begin{pmatrix} \sum\limits_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{11}} & \cdots & \sum\limits_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{1n}} \\ \sum\limits_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{21}} & \cdots & \sum\limits_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{2n}} \\ \vdots & \ddots & \vdots \\ \sum\limits_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{m1}} & \cdots & \sum\limits_i \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{mn}} \end{pmatrix}$$

Now this is satisfying.

Each entry in this matrix is a sum of scalar-by-scalar products. We just applied the scalar chain rule to each component and it's all there in one matrix.

Let's simplify the notation a bit:

We saw earlier that:

- $\frac{\partial y}{\partial W}$ is a tensor because it's a vector with respect to a matrix,
- $\frac{\partial C}{\partial y}$ is a vector.

So what we just wrote is equivalent to:

$$\frac{\partial C(y(W))}{\partial W} = \sum_{i=1}^{k} \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial W}$$

So... we now have this beautiful result:

$$\frac{\partial C(y(W))}{\partial W} = \sum_{i=1}^{k} \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial W}$$

It's clean but it still looks like a summation. And if we're going to scale this up to deep neural networks, we want to write things in a form that is both clean and efficient. That's where **tensor contraction** comes in.

Why introduce tensor contraction?

Because it's the natural extension of the operations we already know and use every day:

- Dot product: contraction over one axis of two vectors.
- Matrix multiplication: contraction over one axis (columns of the first, rows of the second).
- And now... tensor contraction: contraction over one or more axes of a tensor.

The idea is the same: when we "contract" over an axis, we're summing along that dimension reducing its size.

In our case:

$$\frac{\partial C(y(W))}{\partial W}$$

is a matrix, and each

$$\frac{\partial y_i}{\partial W}$$

is a matrix so the sum above is a sum of matrices weighted by the corresponding scalar derivatives.

We can think of:

$$\left\{ \frac{\partial y_i}{\partial W} \right\}_{i=1}^{k}$$

as forming a rank-3 tensor in $\mathbb{R}^{k \times m \times n}$, and then we're contracting over the i-axis using the vector $\frac{\partial C}{\partial y} \in \mathbb{R}^k$.

This is exactly what tensor contraction is: we're taking a tensor and a vector and "collapsing" one axis.

Let's build intuition through an example.

Suppose we're working with images. An RGB image is just a tensor of shape $3 \times H \times W$, where the first axis corresponds to the three color channels (Red, Green, Blue), and the other two are spatial.

Now suppose we want to convert it to grayscale.

How do we do it?

We take a weighted sum of the R, G, and B channels. For example:

$$\text{Gray}(h, w) = 0.3 \cdot R(h, w) + 0.59 \cdot G(h, w) + 0.11 \cdot B(h, w)$$

This is a contraction over the **color axis** the first one. We're reducing a $3 \times H \times W$ tensor into a $H \times W$ matrix by combining slices using fixed weights (0.3, 0.59, 0.11).

That's all tensor contraction is.

And that's exactly what we're doing here.

In our case:

- The tensor is $\frac{\partial y}{\partial W} \in \mathbb{R}^{k \times m \times n}$
- The vector is $\frac{\partial C}{\partial y} \in \mathbb{R}^{k}$
- We are contracting over the i-axis (the output components of y)

So the final expression:

$$\frac{\partial C}{\partial W} = \sum_{i=1}^{k} \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial W} \quad \longrightarrow \quad \frac{\partial C}{\partial W} = \frac{\partial y}{\partial W} \overset{i}{\cdot} \frac{\partial C}{\partial y}$$

is nothing but a **tensor contraction over the i-axis**.

That's how we simplify the chain rule when the variable is a matrix and the intermediate function is vector-valued.

We're now fully equipped to understand backpropagation in its cleanest form.

IV.3 Backpropagation: Computing $\frac{\partial C}{\partial w^l}$ Step by Step

Let's now return to our original goal — the heart of backpropagation:

$$\frac{\partial C\left(a^L(a^{L-1}(\ldots a^l(w^l)\ldots))\right)}{\partial w^l}$$

This is a scalar cost function C applied to the final activation a^L, which depends recursively on all the previous activations down to a^l, which in turn depends on the weights w^l.

Step 1: Recognize the structure

We're taking the derivative of a scalar with respect to a matrix — but this scalar depends on a vector, which depends on a vector, which depends on a vector... and so on, until finally a vector depends on the matrix w^l.

So let's go one step at a time.

Step 2: Apply the chain rule for scalar of vector of matrix

We start from the outermost layer and go backward. At layer l, we apply:

$$\frac{\partial C}{\partial w^l} = \frac{\partial a^l}{\partial w^l} \overset{i}{\cdot} \frac{\partial C}{\partial a^l}$$

This is our first tensor contraction: we contract the tensor $\frac{\partial a^l}{\partial w^l} \in \mathbb{R}^{n_l \times m \times n}$ with the vector $\frac{\partial C}{\partial a^l} \in \mathbb{R}^{n_l}$

Step 3: But where does $\frac{\partial C}{\partial a^l}$ come from?

Well, it flows from the layer above because C ultimately depends on a^L, which depends on a^{L-1}, which depends on a^{L-2}, and so on, all the way down to a^l. That is, we have a full composition:

$$C(a^L(a^{L-1}(\dots(a^{l+1}(a^l)))$$

So when we take the derivative of C with respect to a^l, we apply the chain rule:

$$\frac{\partial C}{\partial a^l} = \left(\frac{\partial a^{l+1}}{\partial a^l}\right)^\top \cdot \frac{\partial C}{\partial a^{l+1}}$$

And again:

$$\frac{\partial C}{\partial a^{l+1}} = \left(\frac{\partial a^{l+2}}{\partial a^{l+1}}\right)^\top \cdot \frac{\partial C}{\partial a^{l+2}}$$

And so on. That's the recursive heart of backpropagation.

All the way until the final layer:

$$\frac{\partial C}{\partial a^L}$$

Step 4: Let's define the backpropagated error signal

We now introduce the recursive quantity:

$$\delta^l = \frac{\partial C}{\partial a^l}$$

Which satisfies:

$$\delta^l = \left(\frac{\partial a^{l+1}}{\partial a^l}\right)^\top \cdot \delta^{l+1} \quad \text{with} \quad \delta^L = \frac{\partial C}{\partial a^L}$$

This is the backbone of backpropagation: the error vector flows backward through the network using this recursion.

Step 5: Final expression

Now the full expression for the gradient becomes:

$$\boxed{\frac{\partial C}{\partial w^l} = \frac{\partial a^l}{\partial w^l} \overset{i}{\cdot} \delta^l}$$

This is the most compact and elegant form of the backpropagation formula — expressed using tensor contraction, and built step-by-step using the chain rule.

We did it.

Alright. We've now laid out all the mathematical tools, climbed carefully through the chain rule jungle, and introduced tensors and contractions where necessary.

Let's put everything together.

What are we trying to compute again?

We're interested in:

$$\frac{\partial C}{\partial w^l} \quad \text{for all layers } l$$

The derivative of the cost with respect to the weights at each layer.

We've seen that:

$$\frac{\partial C}{\partial w^l} = \frac{\partial a^l}{\partial w^l} \overset{i}{\cdot} \delta^l \quad \text{where} \quad \delta^l := \frac{\partial C}{\partial a^l}$$

*Note:*The symbol := means "is defined as" or "is assigned to". It's used when you're introducing a new definition of quantity This is the key idea: once we know the vector δ^l, we can compute the weight gradient using a tensor contraction.

So how do we compute δ^l?

We start at the final layer:

$$\delta^L = \frac{\partial C}{\partial a^L}$$

Then we propagate it backwards using:

$$\delta^l = \left(\frac{\partial a^{l+1}}{\partial a^l}\right)^\top \cdot \delta^{l+1}$$

This is the **recursive heart of backpropagation**. The error signal flows backward through the network.

Summary: The Backpropagation Algorithm

For each layer l, starting from the last one and moving backward:

$$\begin{aligned} \delta^L &:= \frac{\partial C}{\partial a^L} \\ \delta^l &:= \left(\frac{\partial a^{l+1}}{\partial a^l}\right)^\top \cdot \delta^{l+1} \quad \text{for } l = L-1, L-2, \ldots, 1 \\ \frac{\partial C}{\partial w^l} &= \frac{\partial a^l}{\partial w^l} \overset{i}{\cdot} \delta^l \end{aligned}$$

Each step involves:

- One Jacobian matrix transpose,
- One matrix-vector product,
- And one tensor-vector contraction.

From here on:

You can plug in specific expressions for the activation functions (like sigmoid or ReLU), use batch notation, or optimize computations using matrix algebra but this is the fundamental idea behind every gradient computed in neural networks.

We've built it from scratch, and now we own it.

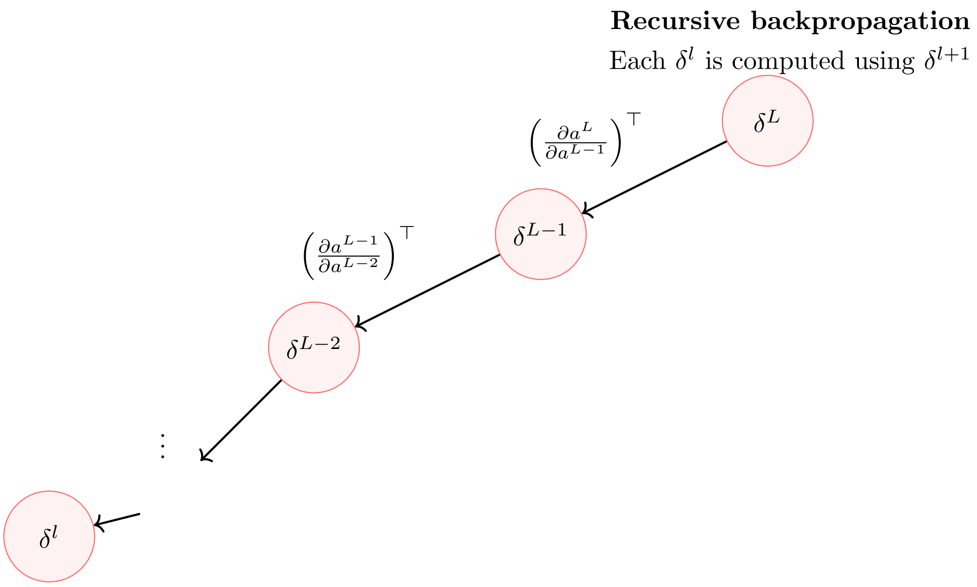

Recursive backpropagation
Each δ^l is computed using δ^{l+1}
δ^L
$\left(\frac{\partial a^L}{\partial a^{L-1}}\right)^\top$
δ^{L-1}
$\left(\frac{\partial a^{L-1}}{\partial a^{L-2}}\right)^\top$
δ^{L-2}
$\vdots$
δ^l

Figure 8: The recursive flow of the backpropagation signal δ^l.

IV.4 Worked Example: Backpropagation with Sigmoid Activation

Let's walk through a simple but powerful example just to see the full machinery in action.

Suppose:

- The final activation function is the sigmoid:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \text{with} \quad \sigma'(z) = \sigma(z)(1-\sigma(z))$$

- The cost function is the squared error:

$$C = \frac{1}{2}\|a^L - y\|^2$$

  where y is the true label and a^L is the output of the final layer.

Let's compute δ^L:

We know:

$$\delta^L = \frac{\partial C}{\partial a^L}$$

But $C = \frac{1}{2}\sum_i (a_i^L - y_i)^2$, so:

$$\frac{\partial C}{\partial a^L} = a^L - y$$

Now let's incorporate the activation derivative:

$$\delta^L = \frac{\partial C}{\partial a^L} \odot \sigma'(z^L) = (a^L - y) \odot a^L \odot (1 - a^L)$$

(We write $\odot$ for elementwise product.)

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \odot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_1 y_1 \\ x_2 y_2 \\ x_3 y_3 \end{pmatrix}$$

Each component multiplies only with its counterpart — no mixing across entries.

Then, recursively:

$$\delta^l = \left(\frac{\partial a^{l+1}}{\partial a^l} \right)^\top \cdot \delta^{l+1}$$

In the case of sigmoid:

$$\frac{\partial a^{l+1}}{\partial a^l} = W^{l+1} \cdot \text{diag}\left(\sigma'(z^{l+1})\right)$$

Why does the derivative take this form?

Because the activation is applied elementwise:

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

So when we differentiate:

$$\frac{\partial a^{l+1}}{\partial a^l} = \frac{\partial \sigma(z^{l+1})}{\partial a^l}$$

We apply the chain rule:

$$\frac{\partial \sigma(z^{l+1})}{\partial a^l} = \frac{\partial z^{l+1}}{\partial a^l} \cdot \frac{\partial \sigma(z^{l+1})}{\partial z^{l+1}}$$

But:

$$\frac{\partial z^{l+1}}{\partial a^l} = W^{l+1}, \quad \frac{\partial \sigma(z^{l+1})}{\partial z^{l+1}} = \text{diag}\left(\sigma'(z^{l+1})\right)$$

So altogether:

$$\frac{\partial a^{l+1}}{\partial a^l} = W^{l+1} \cdot \text{diag}\left(\sigma'(z^{l+1})\right)$$

Each unit only depends on its own z_i^{l+1}, so the Jacobian is diagonal.
So how do we go from that to the elementwise product $\odot$?
Let's say we want to compute:

$$\left(\frac{\partial a^{l+1}}{\partial a^l} \right)^\top \cdot \frac{\partial C}{\partial a^{l+1}}$$

We just said:

$$\frac{\partial a^{l+1}}{\partial a^l} = W^{l+1} \cdot \text{diag}\left(\sigma'(z^{l+1})\right)$$

So the transpose becomes:

$$\left(\frac{\partial a^{l+1}}{\partial a^l} \right)^\top = \left(\text{diag}\left(\sigma'(z^{l+1})\right) \cdot (W^{l+1})^\top \right)$$

Then applying it to $\frac{\partial C}{\partial a^{l+1}}$, we get:

$$\left(\mathrm{diag}\left(\sigma'(z^{l+1})\right) \cdot (W^{l+1})^\top\right) \cdot \frac{\partial C}{\partial a^{l+1}}$$

And here's the trick: multiplying a diagonal matrix by a vector is just an elementwise product! So we write:

$$\sigma'(z^{l+1}) \odot \left((W^{l+1})^\top \cdot \frac{\partial C}{\partial a^{l+1}}\right)$$

This gives the clean recursive formula we love in backpropagation.
So:

$$\delta^l = \left(W^{l+1}\right)^\top \cdot \delta^{l+1} \odot \sigma'(z^l) = \left(W^{l+1}\right)^\top \cdot \delta^{l+1} \odot a^l \odot (1 - a^l)$$

Then compute the gradient of weights:

$$\frac{\partial C}{\partial w^l} = \frac{\partial a^l}{\partial w^l}{}^{i} \cdot \delta^l$$

And in this case:

$$\frac{\partial a_i^l}{\partial w_{jk}^l} = \frac{\partial \sigma(z_i^l)}{\partial w_{jk}^l} = \sigma'(z_i^l) \cdot \frac{\partial z_i^l}{\partial w_{jk}^l} = \sigma'(z_i^l) \cdot a_k^{l-1}$$

Which leads to the familiar expression:

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

Or in full matrix form:

$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^\top$$

And that's it. This is the exact form used in all modern deep learning libraries.

Note: What about the bias b^l?
The bias appears in the pre-activation like this:

$$z^l = W^l a^{l-1} + b^l$$

So when we differentiate with respect to b^l, it's even simpler.
We know:

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

So b^l affects z^l directly and linearly. That means:

$$\frac{\partial z^l}{\partial b^l} = I$$

And by the chain rule:

$$\frac{\partial C}{\partial b^l} = \frac{\partial z^l}{\partial b^l} \cdot \frac{\partial C}{\partial z^l} = I \cdot \delta^l = \delta^l$$

So we get:

$$\boxed{\frac{\partial C}{\partial b^l} = \delta^l}$$

Same process. The gradient with respect to the bias is just the error signal at that layer — no need to multiply by anything else.

For each layer l, compute:

1. **Error vector (delta):**
$$\delta^L = (a^L - y) \odot a^L \odot (1 - a^L)$$
$$\delta^l = \left(W^{l+1}\right)^\top \delta^{l+1} \odot a^l \odot (1 - a^l)$$
2. **Weight gradient:**
$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^\top$$
3. **Bias gradient:**
$$\frac{\partial C}{\partial b^l} = \delta^l$$

These are the exact formulas used in practice compact, efficient, and derived from the full theory.

You can find a working implementation of this example in my accompanying code repository: `github.com/Ka3baAnanas/mnist-digit-recognizer`

We've gone all the way from the full recursive chain rule, through tensors and contractions, to this elegant and practical formula.

References

[1] Michael A. Nielsen. *Neural Networks and Deep Learning.* http://neuralnetworksanddeeplearning.com

[2] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors.* Nature, 323:533–536, 1986.

[3] Grant Sanderson. *3Blue1Brown: Neural Networks Series.* https://www.3blue1brown.com