Phase 1 — Neural Network Math

Goal

Understand the mathematical foundations of neural networks so they can be implemented *from scratch in C++*, without machine learning libraries.

- Time: approximately 4–6 days
- Difficulty: Medium

Notation (Important)

We explicitly distinguish between the true label and the model prediction:

- y denotes the **true label** provided by the dataset
- $\hat{y}$ denotes the **model prediction**
- a denotes the **activation output** of a neuron

Throughout this document, the final-layer activation satisfies $a = \hat{y}$.

What Changed From Phase 0

Phase 0 — Single Neuron

We begin with a single neuron defined by

$$z = wx + b.$$

The neuron produces a prediction

$$\hat{y} = a = \phi(z),$$

where ϕ is an activation function. This is a dot product plus a bias term.

Phase 1 — Layers of Neurons

For a layer of neurons, the computation generalizes to

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}, \qquad \mathbf{a} = \phi(\mathbf{z}).$$

- One neuron corresponds to a dot product
- Multiple neurons correspond to matrix–vector multiplication

This marks the transition from scalar arithmetic to linear algebra.

Why Matrices and Hidden Layers Exist

Why Matrices

A matrix does not introduce a new operation. It simply allows many dot products to be computed at once.

If multiple neurons receive the same input $\mathbf{x}$, each neuron computes

$$z_i = w_i^T \mathbf{x} + b_i.$$

Stacking these computations yields

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}.$$

Each row of W stores the weights of one neuron. Matrices therefore provide parallel evaluation of scalar equations.

Why Hidden Layers

A single neuron computes a function of the form

$$f(\mathbf{x}) = \phi(w^T \mathbf{x} + b),$$

which corresponds to a single separating hyperplane in input space. Such a function cannot represent problems like XOR.

A hidden layer computes

$$\mathbf{a}_1 = \phi(W_1 \mathbf{x} + \mathbf{b}_1),$$

which maps the input into a new coordinate space. In this new space, the output layer can separate data that was not linearly separable in the original space.

The expressive power of neural networks comes from composing linear maps with nonlinear activation functions.

Role of the Prediction $\hat{y}$

The final-layer activation

$$\hat{y} = a_L$$

is the model's prediction for a single input. Its sole purpose is to be compared with the true label y using a loss function.

During training:

- $\hat{y}$ is computed during the forward pass
- $\hat{y}$ is used to compute the loss $L(\hat{y}, y)$
- $\hat{y}$ initiates backpropagation via $\partial L / \partial \hat{y}$

After gradients are computed, $\hat{y}$ is discarded. Learning is stored only in the updated weights.

Practice Problem — XOR

The XOR dataset is given by

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

A single neuron cannot represent XOR. Solving this problem requires a hidden layer and nonlinear activation functions.

Step 1 — Vectors and Matrices

A vector is an ordered list of numbers. A dot product is defined as the sum of elementwise products. Matrix–vector multiplication computes many dot products simultaneously.

The fundamental shape rule is

$$(m \times k)(k \times n) = (m \times n).$$

Each row of a weight matrix corresponds to one neuron.

Step 2 — Forward Pass

A neural network layer computes

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}, \qquad \mathbf{a} = \phi(\mathbf{z}).$$

In this phase we focus on the ReLU activation function, defined as

$$\phi(z) = \max(0, z).$$

ReLU introduces nonlinearity. Without it, multiple layers collapse into a single linear transformation.

Step 3 — Loss (Measuring Error)

The loss function measures how incorrect a prediction is. For classification, we compare the model prediction $\hat{y}$ to the true label y.

For a single output neuron with target $y = 1$, the cross-entropy loss is

$$L(\hat{y}, y) = -\ln(\hat{y}).$$

Step 4 — Backpropagation (Full Derivation with ReLU)

We define the composite functions

$$z(w) = wx + b, \qquad \hat{y}(z) = a(z) = \max(0, z), \qquad L(\hat{y}, y) = -\ln(\hat{y}).$$

The chain rule gives

$$\frac{dL}{dw} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}.$$

Derivatives

$$\frac{\partial L}{\partial \hat{y}} = -\frac{1}{\hat{y}}, \qquad \frac{\partial z}{\partial w} = x.$$

The ReLU derivative is

$$\frac{\partial \hat{y}}{\partial z} = \begin{cases} 1, & z > 0, \\ 0, & z \leq 0. \end{cases}$$

Final Gradient

$$\frac{dL}{dw} = -\frac{1}{\hat{y}} \cdot \mathbb{1}_{z>0} \cdot x.$$

Step 5 — Learning (Gradient Descent)

Weights are updated using gradient descent:

$$w_{\text{new}} = w - \eta \frac{dL}{dw}.$$

Step 6 — Backpropagation (Two-Layer Network, Matrix Rules)

This section removes ambiguity about how to update entire matrices W_1 and W_2.

Forward Pass (Two Layers)

Let the input be $\mathbf{x} \in \mathbb{R}^n$. Choose a hidden width h.

Hidden layer:

$$\mathbf{z}_1 = W_1\mathbf{x} + \mathbf{b}_1, \qquad \mathbf{a}_1 = \phi(\mathbf{z}_1),$$

where $W_1 \in \mathbb{R}^{h\times n}$ and $\mathbf{b}_1 \in \mathbb{R}^h$.

Output layer (single output neuron):

$$z_2 = W_2\mathbf{a}_1 + b_2, \qquad \hat{y} = a_2 = \phi_{\text{out}}(z_2),$$

where $W_2 \in \mathbb{R}^{1\times h}$ and $b_2 \in \mathbb{R}$.

Loss:

$$L = L(\hat{y}, y).$$

Define Error Signals

Define:

$$\delta_2 = \frac{\partial L}{\partial z_2}, \qquad \boldsymbol{\delta}_1 = \frac{\partial L}{\partial \mathbf{z}_1}.$$

Output Layer Gradients

Because $z_2 = W_2\mathbf{a}_1 + b_2$:

$$\frac{\partial L}{\partial W_2} = \delta_2\, \mathbf{a}_1^T, \qquad \frac{\partial L}{\partial b_2} = \delta_2.$$

Entry-wise (what your loops compute):

$$\left(\frac{\partial L}{\partial W_2}\right)_{1j} = \delta_2 (a_1)_j.$$

Backpropagate Into the Hidden Layer

First move the error back through W_2:

$$\frac{\partial L}{\partial \mathbf{a}_1} = W_2^T \delta_2.$$

Then apply the activation derivative (ReLU gate) elementwise:

$$\boldsymbol{\delta}_1 = \left(W_2^T \delta_2\right) \odot \phi'(\mathbf{z}_1),$$

where $\odot$ is elementwise multiplication and for ReLU

$$\phi'(z) = \begin{cases} 1, & z > 0, \\ 0, & z \leq 0. \end{cases}$$

Entry-wise:

$$(\delta_1)_i = (W_2)_{1i}\, \delta_2 \cdot \mathbb{1}_{(z_1)_i > 0}.$$

Hidden Layer Gradients

Because $\mathbf{z}_1 = W_1\mathbf{x} + \mathbf{b}_1$:

$$\frac{\partial L}{\partial W_1} = \boldsymbol{\delta}_1\, \mathbf{x}^T, \qquad \frac{\partial L}{\partial \mathbf{b}_1} = \boldsymbol{\delta}_1.$$

Entry-wise (this answers your exact question):

$$\left(\frac{\partial L}{\partial W_1}\right)_{ij} = (\delta_1)_i\, x_j.$$

So **every** weight $W_1[i][j]$ gets its own gradient $(\delta_1)_i x_j$.

Where Does δ_2 Come From?

δ_2 depends on your chosen output activation and loss.

If ϕ_{out} is identity (regression):

$$\delta_2 = \frac{\partial L}{\partial \hat{y}}.$$

If ϕ_{out} is sigmoid and L is binary cross-entropy, a common simplification is:

$$\delta_2 = \hat{y} - y.$$

Order of Operations (No Interpretation)

For one training example $(\mathbf{x}, y)$:

1. Forward: compute $\mathbf{z}_1, \mathbf{a}_1, z_2, \hat{y}$.
2. Compute loss $L(\hat{y}, y)$.
3. Compute δ_2.
4. Compute $\partial L/\partial W_2$ and $\partial L/\partial b_2$.
5. Compute $\boldsymbol{\delta}_1 = (W_2^T \delta_2) \odot \phi'(\mathbf{z}_1)$.
6. Compute $\partial L/\partial W_1$ and $\partial L/\partial \mathbf{b}_1$.

Step 7 — Learning (Gradient Descent, Matrix Updates)

Update every parameter with:

$$\text{param}_{\text{new}} = \text{param} - \eta \frac{\partial L}{\partial \text{param}}.$$

So:

$$W_2^{\text{new}} = W_2 - \eta \frac{\partial L}{\partial W_2}, \qquad b_2^{\text{new}} = b_2 - \eta \frac{\partial L}{\partial b_2},$$

$$W_1^{\text{new}} = W_1 - \eta \frac{\partial L}{\partial W_1}, \qquad \mathbf{b}_1^{\text{new}} = \mathbf{b}_1 - \eta \frac{\partial L}{\partial \mathbf{b}_1}.$$

Entry-wise, for W_1 this means:

$$(W_1^{\text{new}})_{ij} = (W_1)_{ij} - \eta(\delta_1)_i x_j.$$

You do this for **all** i, j. There is no single constant used for the entire matrix.

Next Phase

Phase 2 — CPU Neural Network in C++

You will implement:

- Matrix operations
- Forward propagation
- Backpropagation
- Gradient descent