

Inside the Brain

Build a neural network by hand — nothing but NumPy

Day 04 · Week 1

Engine day · no module — you forge the core

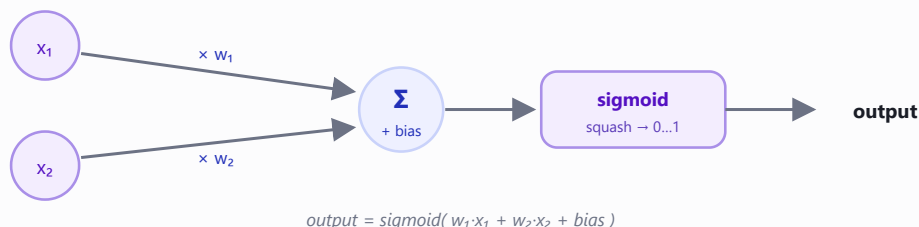
1 Mission briefing

The Chief Scientist drops by with a grin: "Anyone can call a library. Today you build a neural network with your bare hands — just NumPy — and train it to think." No PyTorch, no shortcuts. *Previously:* you unlocked the Pattern Finder and watched machines find structure with no labels. Today you build the thing everyone *means* when they say "AI" — and after this, the black box is never black again.

- ☐ Understand a single **neuron** and why one is never enough
- ☐ Discover a layer is really **one matrix multiplication**
- ☐ Write the **forward pass**, a **loss**, and the **nudge** update
- ☐ Watch the decision boundary bend around the data as it learns
- ☐ See how **hidden size** and **learning rate** change everything
- ☐ Peek at the **loss landscape** the network rolls down

2 Field guide the concepts

A neuron is a bouncer at a door. Each input (a reason to open) has a **weight** (how much it matters). The bouncer multiplies each input by its weight, sums them, adds a personal **bias**, and runs the total through an **activation** — the **sigmoid** squash that turns any number into a 0-to-1 confidence. One neuron draws one straight cut, so it can't split two tangled moons; stack a **hidden layer** of them and the boundary can bend.



One neuron. A whole **layer** of them is a single matrix multiply: $(300, 2) @ (2, 16) \rightarrow (300, 16)$.

Learning = walking downhill. A **loss** scores how wrong the guesses are (one number). The **gradient** says which way each weight should move to shrink it; today you estimate it by *nudging* each weight and re-checking the loss, then step every weight a little downhill (the **learning rate** sets the step size). Remember the big secret: **matrix multiplication is central to the network** — high-performance computing makes those fast.

A precise reading

The gradient points toward increasing loss; subtract it to descend. Finite differences approximate derivatives. Backpropagation uses one forward pass and one backward pass, not two forward passes. Matrix products are central, alongside biases, nonlinear activations and other operations. A sigmoid output between 0 and 1 is not automatically a calibrated probability.

3 Lab missions check each off in the notebook

1 Implement the sigmoid ☐

`1 / (1 + np.exp(-z))`, working on a whole array at once.

Expect: `sigmoid(0) = 0.5` and a smooth S-curve from 0 to 1.

2 Matrix multiply, by hand vs NumPy ☐

Write the triple-loop `matmul_by_hand`, then race it against `@`.

Expect: the exact same answer — and NumPy winning by hundreds of times.

3 Write the forward pass ☐

Chain two layers: `H = sigmoid(X@W1+b1)`, then the output layer.

Expect: a (300, 1) array of mush near 0.5 — it hasn't learned yet.

4 Implement the loss ☐

Mean squared error: the average of `(pred - truth)2`.

Expect: an untrained loss near 0.25.

5 Take the step (this is learning) ☐

Nudge every weight downhill, in place: `p -= lr * g`.

Expect: the loss falls epoch by epoch as the net figures out the moons.

6 Score it ☐

Threshold predictions at 0.5 and take the fraction correct.

Expect: a well-trained net lands around 88% on the moons (the ceiling is ~97% — tomorrow's power tools close the gap).

7 Retrain with 2 and 64 hidden neurons ☐

Train both and plot their boundaries side by side.

Expect: 2 is too stiff to bend; 64 is far wigglier.

4 Cheat sheet a neural net in pure NumPy

<code>sigmoid(z) = 1 / (1 + np.exp(-z))</code>	Squash any number to a 0–1 confidence
<code>X @ W1 + b1</code>	A whole layer of neurons in one matrix multiply
<code>(300,2) @ (2,16) → (300,16)</code>	Shapes: data @ weights → hidden values
<code>sigmoid(sigmoid(X@W1+b1) @ W2 + b2)</code>	The full two-layer forward pass
<code>np.mean((y_pred - y_true) ** 2)</code>	MSE loss — one number of wrongness
<code>p -= lr * g</code>	The update: step each weight downhill (learning!)
<code>loss(w+eps) vs loss(w-eps)</code>	The nudge: estimate a weight's gradient
<code>np.random.default_rng(SEED)</code>	Reproducible small random starting weights

5 What you forged today

The engine — built by hand

No file to save today. You built the **core** that tomorrow's Studio module runs on: a neural network from nothing but NumPy that teaches itself to separate two interlocking moons.

65

weights & biases
(2·16 + 16 + 16 + 1)

~88%

accuracy on the
two-moons task

130 → 1 + 1

130 forward passes today:
1 forward + 1 backward tomorrow

Tomorrow the Lab bolts on power tools — PyTorch autograd and a GPU — and this same engine learns to read handwriting.

6 Mini-glossary

neuron

Weighted sum of inputs + bias, run through an activation.

bias

A neuron's offset — its baseline mood, added before the squash.

hidden layer

Neurons between input and output; more = a wigglier boundary.

gradient

The slope of the loss — which way to move each weight to improve.

weight

How much one input counts — a knob the network tunes.

activation

The squashing step (sigmoid) that bends straight lines into curves.

loss

One number saying how wrong the predictions are (here, MSE).

learning rate

How big a step to take downhill each update.

7 Show & tell

1. How many times did NumPy beat your hand-written **matmul**? Why does that gap decide the back half of this course?
2. Show the boundary at **2 vs 64** hidden neurons — which one memorized, which generalized?
3. The nudge method spent ~**130 forward passes** per step. What does tomorrow's **autograd** promise instead?