

The Digit Reader

Superpowers: PyTorch + GPU

Day 05 · Week 1

Unlocks 🔓 Digit Reader

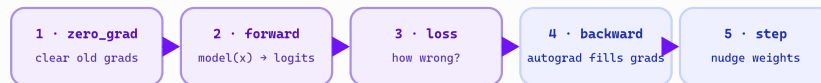
1 Mission briefing

Yesterday you hand-built a network's engine in pure NumPy — and felt the pain: ~130 forward passes just to nudge the weights *once*. Today the Lab issues power tools. **PyTorch** computes those nudges automatically and exactly, and a **GPU** runs them at ludicrous speed. *Previously*: you earned a deep respect for backpropagation — and calluses. Your mission: teach a network to read handwriting, then point it at *your own*.

- ☐ Meet **tensors** — arrays that live on a GPU and track history
- ☐ Let **autograd** compute gradients for you
- ☐ Train a real network on **MNIST** handwriting
- ☐ Watch a **GPU** demolish a CPU at matrix multiply
- ☐ Upgrade to a **convolutional** network (CNN)
- ☐ Draw your own digit and **unlock the Digit Reader**

2 Field guide the concepts

Tensors and autograd. A **tensor** is a NumPy array with two superpowers: it can live on a **GPU**, and it remembers its own history. Mark it `requires_grad=True`, do some math, call `.backward()`, and PyTorch replays the chain rule to leave the exact gradient in `.grad` — the same job yesterday's nudge method did, but in **two passes instead of 130**. Almost every network in the world then trains with the **same five-step loop**. Burn it into memory:



repeat for every batch, every epoch — even GPT trains on this skeleton

The five lines that train (almost) all of AI. Under the hood it's all the `@` matrix multiply from Day 4.

MNIST, CNNs, and the GPU. You'll train on **MNIST** — 60,000 training and 10,000 test handwritten digits — in **batches**, for a few **epochs** (full passes), with the **Adam** optimizer. A plain MLP flattens the image; a **convolutional** network keeps its 2-D shape and scores higher. And because every step is matrix multiplication, a **GPU** that does thousands of multiplies at once leaves the CPU in the dust.

A precise reading

MNIST has 60,000 training and 10,000 test images; the test images do not train the model. Autograd applies the chain rule to recorded tensor operations, subject to floating-point precision. GPU speed depends on matrix size, hardware and transfer overhead; small workloads can be faster on the CPU.

3 Lab missions check each off in the notebook

1 Your first tensors

Matrix-multiply a 3×4 and a 4×2 tensor with `@`.

Expect: `C` is 3×2 — the shared inner dimension cancels.

2 Differentiate by yourself (well, by PyTorch)

Build `y = x**3 + 2*x` at `x = 3`, call `y.backward()`.

Expect: `x.grad` is 29 ($3x^2 + 2$ at $x=3$).

3 Assemble the MLP

`nn.Sequential`: Flatten → Linear(784,128) → ReLU → Linear(128,10).

Expect: a four-layer model, moved onto `DEVICE`.

4 Count the parameters

Sum `p.numel()` over `mlp.parameters()`.

Expect: exactly 101,770 tunable knobs.

5 Complete the training loop

Fill the five gaps: `zero_grad` → forward → loss → backward → step.

Expect: the printed loss goes down each epoch.

6 Write `evaluate`

A `torch.no_grad()` loop counting correct `argmax` predictions.

Expect: MLP test accuracy ≥ 0.95 (~97%).

7 Time the matmul

Time `A @ B` on CPU vs GPU (with `cuda.synchronize()`).

Expect: the GPU obliterates the CPU on big matrices.

8 Train the CNN by reusing your loop

Instantiate `SmallCNN` and call the same `train_model`.

Expect: accuracy climbs a couple of points above the MLP.

4 Cheat sheet PyTorch essentials

<code>torch.from_numpy(a) · t.numpy()</code>	Bridge between NumPy and torch
<code>x = torch.tensor(2., requires_grad=True)</code>	Track history so autograd can differentiate
<code>y.backward(); x.grad</code>	Fill in every gradient in one backward pass
<code>tensor.to(DEVICE)</code>	Move data / model onto the GPU (or CPU)
<code>nn.Sequential(nn.Flatten(), nn.Linear..., nn.ReLU...)</code>	Stack layers front to back
<code>nn.CrossEntropyLoss()</code> · <code>optim.Adam(..., lr=1e-3)</code>	Classification loss · the optimizer
<code>zero_grad</code> → <code>model(x)</code> → <code>loss</code> → <code>backward</code> → <code>step</code>	The five-line training loop
<code>with torch.no_grad(): logits.argmax(dim=1)</code>	Evaluate without tracking gradients
<code>DataLoader(ds, batch_size=128, shuffle=True)</code>	Feed data in shuffled batches
<code>torch.jit.trace(model, ex).save("...pt")</code>	Export a frozen TorchScript model

5 Unlock your Studio module

Digit Reader

Export the trained CNN as a self-contained TorchScript file the Studio can load without your training code. It must write exactly:

```
ai_studio/models/digit_reader.pt
```

Input contract: a (1, 1, 28, 28) tensor, values 0–1, **white digit on black** (MNIST style — **ToTensor** only, no normalization). Then:

```
$ python ai_studio/app.py
```

6 Mini-glossary

tensor

A NumPy-like array that can run on a GPU and track its own history.

epoch

One full pass over the whole training set.

optimizer

The rule (e.g. Adam) that turns gradients into weight updates.

TorchScript

A frozen, portable copy of a model, saved as a `.pt` file.

autograd

PyTorch's automatic gradients via one backward pass of the chain rule.

batch

A small group of examples processed together in one step.

CUDA / GPU

Hardware that does thousands of multiplies in parallel.

7 Show & tell

1. What test accuracy did your **MLP** reach — and how many more digits did the **CNN** get right?
2. Draw three digits in your Studio's **Digit Reader**. Which one fooled it, and why do you think it slipped?
3. In the GPU race, how many times faster was the biggest **matmul**? Tie that back to why GPUs matter.