PART I · TENSOR FLUENCY — CHAPTER 2

What a Tensor Really Is

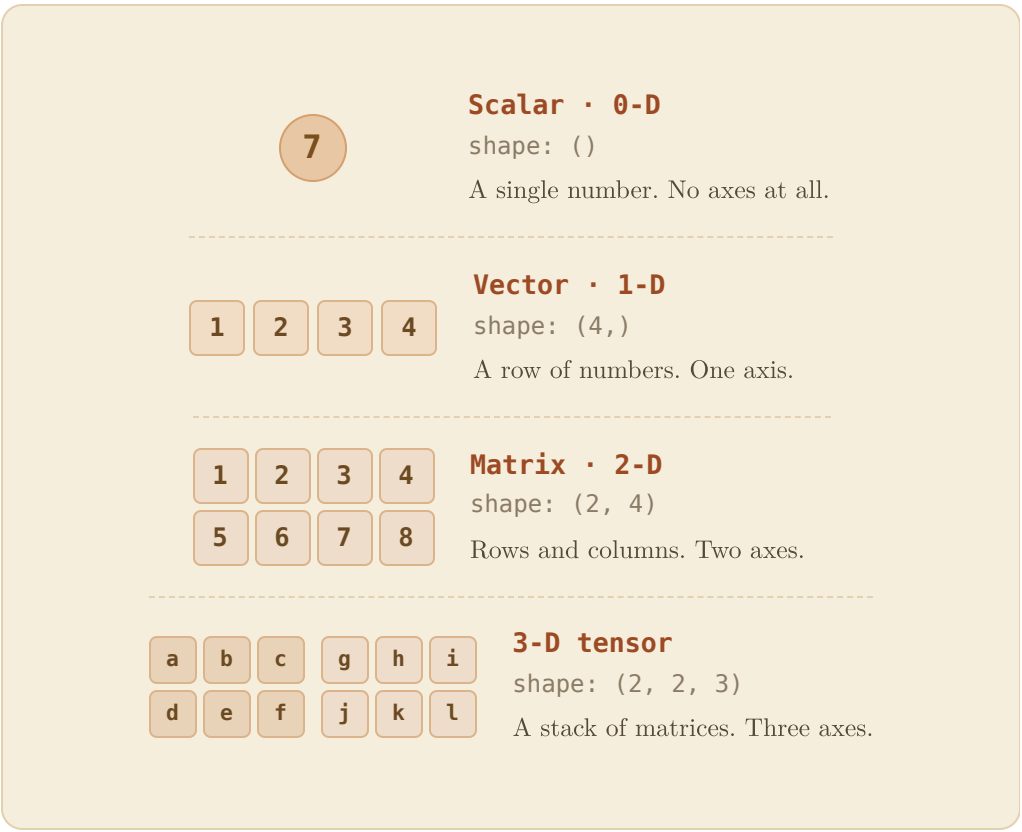
A tensor is just a box of numbers with a shape. Everything else follows from that.

The word *tensor* can sound intimidating — it shows up in physics, differential geometry, and a lot of heavy mathematics. Forget all of that. In PyTorch, a tensor is the simplest thing in the world: a container of numbers arranged in a regular grid, with a shape that tells you how those numbers are organized. A single number is a tensor. A row of numbers is a tensor. A spreadsheet of numbers is a tensor. A stack of spreadsheets is a tensor. The only thing that changes is how many dimensions the grid has.

This chapter walks you through that idea from the ground up: one dimension at a time, then the three facts PyTorch keeps about every tensor — its *shape*, its *dtype*, and its *device* — and, finally, the quiet truth underneath: how those numbers actually sit in memory, and what a *stride* is. You do not need to memorize the memory model today, but meeting it now means it will feel familiar the first time a reshape fails and you need to understand why.

The dimension ladder

Every tensor has a number of dimensions, which PyTorch calls its `ndim`. The ladder below shows the first four rungs. There is no upper limit — you can have five, six, or more dimensions — but almost everything you meet in practice lives on these four.



Each rung adds one axis. A 3-D tensor is just two matrices stacked; a 4-D tensor is a batch of those stacks.

```
scalar = torch.tensor(7)
vector = torch.tensor([1, 2, 3, 4])
matrix = torch.tensor([[1,2,3,4],
                       [5,6,7,8]])
cube   = torch.randn(2, 2, 3)

print(scalar.ndim, vector.ndim,
      matrix.ndim, cube.ndim)
```

OUTPUT

```
0 1 2 3
```

TRY IT

Create a 4-D tensor with `torch.zeros(2, 3, 4, 5)`. Print its `.ndim` and `.shape`. Can you say in plain English what each of the four numbers in the shape means — how many of *what*?

The three facts: shape, dtype, device

Every tensor carries exactly three pieces of metadata, and checking them is how you stay out of trouble. You already met *device* in Chapter 1. Now we add the other two.

```
t = torch.tensor([[1.0, 2.0],
                  [3.0, 4.0]])

print(t.shape)    # torch.Size([2, 2])
print(t.dtype)    # torch.float32
print(t.device)   # cpu
```

Shape tells you how the numbers are arranged: two rows, two columns. **Dtype** (data type) tells you what kind of number each cell holds — here, a 32-bit floating-point number. **Device** tells you where the tensor lives — on the CPU or on a GPU. Together, these three facts answer the question "what am I working with?" and they are the first thing to print whenever something goes wrong.

Dtype: picking the right number type

Not every number needs the same amount of precision. PyTorch lets you choose, and the default — `float32` — is the right one for almost everything you will do in this book. Here are the types you will actually encounter:

DTYPE	WHAT IT STORES	WHEN YOU'LL USE IT
`torch.float32`	32-bit decimal	Default. Model weights, inputs, losses.
`torch.float64`	64-bit decimal	Rare. Extra precision for numerical checks.
`torch.float16`	16-bit decimal	Mixed-precision training on GPUs.
`torch.int64`	64-bit integer	Class labels, indices.
`torch.bool`	True / False	Masks, comparisons.

You can always check with `.dtype` and convert with `.to(torch.float32)` or `.float()`.

```
labels = torch.tensor([0, 1, 2])
print(labels.dtype)          # torch.int64

floats = labels.float()      # convert to float32
print(floats.dtype)          # torch.float32
```

WATCH OUT

When you create a tensor from integers like `[1, 2, 3]`, PyTorch defaults to `int64`. When you create from decimals like `[1.0, 2.0]`, it defaults to `float32`. Mixing the two in an operation often produces a dtype-mismatch error. The fix: cast one of them explicitly with `.float()` or `.long()`.

Shape: learning to read it

The shape is a tuple of integers, one per dimension, and it tells you the size along each axis. Reading it is a skill you will use a hundred times a day, so it is worth getting comfortable now.

```
img_batch = torch.randn(32, 3, 28, 28)
print(img_batch.shape)
```

OUTPUT

```
torch.Size([32, 3, 28, 28])
```

Read it left to right: 32 images, each with 3 color channels, each channel 28 pixels tall and 28 pixels wide. This convention — *batch, channels, height, width*, or **NCHW** — is the standard layout for image data in PyTorch, and you will see it constantly.

```
print(img_batch.shape[0])   # 32 — batch size
print(img_batch.shape[-1])  # 28 — last dim (width)
print(img_batch.size(1))    # 3  — channels
```

You can index into `.shape` like any tuple, and `.size(dim)` does the same thing. Both are useful; use whichever reads more clearly in context.

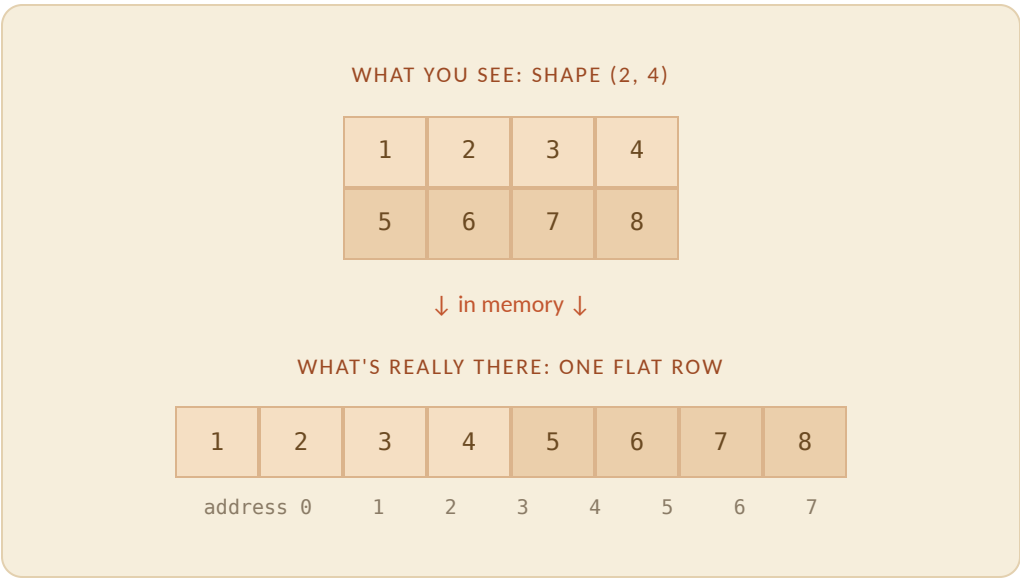
TRY IT

Create `torch.randn(8, 512)`. Without running it, say what `.shape`, `.ndim`, and `.shape[0]` will print. Then run it and check.

Under the hood: memory is flat

Here is the quiet truth that makes reshape, view, transpose, and contiguity all make sense: no matter how many dimensions a tensor *looks* like it has, the actual numbers sit in a single, flat row in memory. A 2×4

matrix is eight numbers in a line. The shape is just a lens PyTorch holds over that line to let you see it as rows and columns.

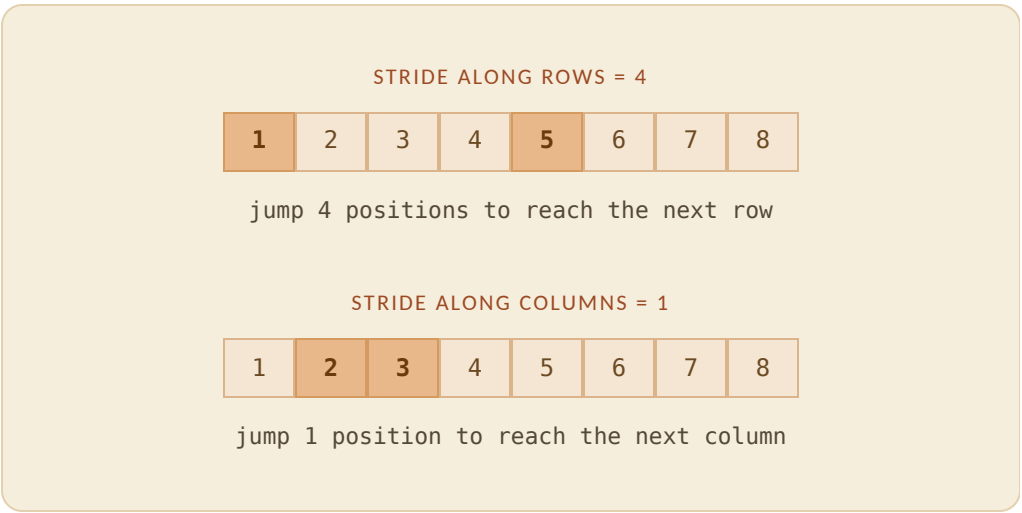


Row-major order: row 0 first, then row 1, laid out end to end.

PyTorch (like C and NumPy) stores tensors in *row-major* order: the last axis varies fastest. So the first row fills addresses 0–3, the second row fills 4–7. This is just a convention, but it matters because it determines which operations can reinterpret the memory without copying it (like `view`) and which cannot.

Strides: how PyTorch navigates the flat row

If memory is flat, how does PyTorch find the element at row 1, column 2? With a small, elegant idea called a *stride*. The stride tells you how many positions to jump in memory to move one step along each axis.



For a (2, 4) tensor stored row-major, the strides are (4, 1).

```
m = torch.tensor([[1,2,3,4],
                  [5,6,7,8]])

print(m.shape)     # torch.Size([2, 4])
print(m.stride())  # (4, 1)
```

The tuple `(4, 1)` says: to move one step down (next row), jump 4 elements; to move one step right (next column), jump 1. That is all PyTorch needs to turn a flat memory block into any shape you ask for. You will see strides again when we talk about `view` vs `reshape`, `transpose`, and contiguity — they are the single idea underneath all of those.

WATCH OUT

You do not need to work with strides directly in everyday code. They exist so that operations like `transpose` and `view` can rearrange a tensor without copying its data — which is fast and memory-efficient, but occasionally means a "non-contiguous" tensor cannot be `view`-ed, only `reshape`-ed. We will cover that in detail in Chapter 6.

Numel and storage size

Two quick helpers round out the picture. `.numel()` returns the total number of elements — the product of all dimensions — and `.element_size()` tells you how many bytes each one takes.

```
t = torch.randn(3, 4)

print(t.numel())          # 12  (3 × 4)
print(t.element_size())   # 4 bytes (float32)
print(t.numel() * t.element_size(),
      "bytes")            # 48 bytes total
```

These are useful later when you want to estimate how much memory a model or a batch of data will consume — and they give you a concrete sense of scale that shape alone does not.

Putting it all together

Here is the complete mental model so far. A tensor is a grid of numbers. PyTorch remembers three things about it: its *shape* (how the grid is organized), its *dtype* (what kind of number), and its *device* (where it lives). Under the hood, the grid is really a flat row of numbers, and the *stride* is the recipe for navigating it. That is the whole foundation — everything from here forward is just learning to work with it fluently.

THE RULE

1. A tensor is a grid of numbers with a *shape* — one integer per axis.
2. Every tensor carries three facts: **shape**, **dtype**, **device**. Print them first when debugging.
3. `ndim` counts the axes. `numel()` counts the total elements.
4. Memory is flat. The **stride** tells PyTorch how to navigate it — which is why some reshapes are free and others require a copy.
5. The default dtype is `float32` for decimals, `int64` for integers. Cast explicitly when mixing them.

END OF CHAPTER 2

Exercises

1. Create tensors of dimensions 0 through 4 and print each one's `.ndim`, `.shape`, and `.numel()`. Verify that `numel` equals the product of the shape.
2. Create `torch.tensor([1, 2, 3])` and `torch.tensor([1.0, 2.0, 3.0])`. Print each tensor's `.dtype`. Why are they different? Convert the integer tensor to `float32` and confirm the dtype changed.
3. Create a (3, 5) tensor of random numbers. Print its `.stride()`. Explain in one sentence what each number in the stride tuple means.
4. Calculate by hand how many bytes a `torch.float32` tensor of shape (64, 3, 224, 224) would consume. Then check with `.numel() * .element_size()`. Does your answer match?
5. Create a (2, 3) tensor. Print its `.storage()` — what do you notice about the format? How does it compare to what you see when you print the tensor normally?

INTERVIEW-STYLE PROBLEM

An interviewer asks: "You have a tensor of shape `(32, 3, 28, 28)`. What does each dimension represent, and how many total numbers does this tensor contain?" Answer in two to three sentences. Then they follow up: "What is its stride, and why does the stride matter?" Give a one-sentence answer.