PyTorch: Storage, Shape, and Axis-Moving

v0.3 · June 2026

Matthew Willetts, with assistance from Codex and Claude

A reference for the operations that change a tensor's shape or axis order: what copies, what doesn't, and the precise semantics of each.

The three things that define a tensor

A `torch.Tensor` is not just a multi-dim array of values. It is three things wrapping a flat storage:

1. **Storage** — a flat 1D block of memory holding the actual numbers.
2. **Shape** — how to interpret those numbers as a multi-dim grid.
3. **Strides** — how many storage elements to skip when each dim index increments by 1.

Multiple tensors can share one storage with different shapes and strides — that is what views are.

Row-major (C-order) layout

PyTorch and NumPy default to **row-major** layout: the last dim is the "fast" one, and rows are contiguous in storage.

For `X = torch.arange(6).reshape(2, 3)`:

```
X = [[0, 1, 2],
     [3, 4, 5]]
Storage:  [0, 1, 2, 3, 4, 5]
Shape:    (2, 3)
Strides:  (3, 1)
```

`Strides = (3, 1)` says: to move one row, advance 3 in storage; to move one column, advance 1.

A tensor is **contiguous** when its strides match what you would expect for a freshly-allocated row-major layout. Operations that change axis order break contiguity by leaving strides that no longer match storage order.

The reshape rule

> **`reshape(...)` walks the logical tensor in row-major order, then lays out the new shape filling row-major.**

The key word is *logical*. Reshape iterates through the tensor as if it had been freshly materialised into a contiguous buffer, and uses that walk to populate the new shape.

For a contiguous tensor, the logical row-major walk matches storage order, so `reshape` is just metadata — no copy.

For a non-contiguous tensor (typically after a transpose / permute), the logical row-major walk does not correspond to a stride pattern over the original storage, so PyTorch **materialises a copy** into a fresh contiguous buffer first, then reshapes.

`view` vs `reshape`

- **`view(...)`**: requires the tensor's strides to permit the new shape without a copy. Errors otherwise. Never copies.
- **`reshape(...)`**: permissive. Returns a view if it can; copies if it cannot. Convenient but hides the cost.

```
X = torch.arange(6).reshape(2, 3)          # contiguous

X.view(3, 2)                               # works - view
X.reshape(3, 2)                            # works - view (no copy needed)

X.T.view(-1)                               # RuntimeError - strides incompatible
X.T.contiguous().view(-1)                  # works - explicit copy via contiguous()
X.T.reshape(-1)                            # works - silent copy
```

For hot-path code where allocations matter, prefer `view` and call `.contiguous()` explicitly so copies are visible. For prototyping, `reshape` is fine.

Axis-moving operations (the catalogue)

All of these are metadata changes — they swap shape and strides without moving data. They typically produce non-contiguous tensors.

op	what it does	example
`.T`	2D transpose; reverses dims for >2D (deprecated for >2D)	`X.T`
`.mT`	matrix transpose — swap the last two dims	`X.mT`
`.t()`	2D transpose, method form	`X.t()`
`.transpose(d0, d1)`	swap two specific dims	`X.transpose(0, 2)`
`.swapaxes(d0, d1)` / `.swapdims(d0, d1)`	aliases for transpose	`X.swapaxes(0, 1)`
`.permute(*dims)`	reorder all dims at once	`X.permute(2, 0, 1)`
`.movedim(src, dst)`	move dim(s) from src to dst, slide rest	`X.movedim(0, -1)`
`.unsqueeze(dim)`	insert a length-1 dim	`X.unsqueeze(0)`
`.squeeze(dim)`	remove a length-1 dim	`X.squeeze(0)`
`.flatten(start, end)`	collapse contiguous dim range	`X.flatten(1, 3)`

Notes:

- `permute` takes a tuple listing the *source* dim for each new position: `permute(2, 0, 1)` means "new dim 0 is old dim 2; new dim 1 is old dim 0; ...".
- `movedim` is the ergonomic "take this one dim and put it over there" form: `movedim(0, -1)` slides dim 0 to the last position. Also accepts tuples: `movedim((0, 1), (-2, -1))`.
- `squeeze` / `unsqueeze` preserve contiguity (they only insert / remove length-1 dims).
- `flatten` on a non-contiguous tensor will copy, like reshape.

`transpose` vs `reshape` to the same shape

These produce **same-shape but completely different tensors**:

```
X = torch.arange(120).reshape(2, 3, 4, 5)

Y = X.transpose(0, 2)        # shape (4, 3, 2, 5)
Z = X.reshape(4, 3, 2, 5)    # shape (4, 3, 2, 5)
```

The mapping is governed by completely different rules:

- **Transpose** preserves data semantics. `Y[c, b, a, d] == X[a, b, c, d]` — same element, relabelled coordinates.
- **Reshape** preserves the flat byte stream. `Z[c, b, a, d] == X.flatten()[c*30 + b*10 + a*5 + d]` — whichever element falls at that flat index under the new shape's row-major traversal.

Spot check:

```
X[1, 2, 3, 4]   # = 119
Y[3, 2, 1, 4]   # = X[1, 2, 3, 4] = 119
Z[3, 2, 1, 4]   # = X.flatten()[119] = 119

Y[3, 2, 0, 4]   # = X[0, 2, 3, 4] = 59
Z[3, 2, 0, 4]   # = X.flatten()[114] = 114
```

Same coordinate, different elements.

Why this matters

If `X` is a rollout buffer of shape `(T, N, obs_dim)`:

- `X.transpose(0, 1)` → `(N, T, obs_dim)`, **data preserved**, semantically still "obs of env n at time t."
- `X.reshape(N, T, obs_dim)` → `(N, T, obs_dim)`, **data scrambled**, the new dim 0 is no longer "env index."

If you write the reshape when you meant the transpose, your training loop runs without error and produces garbage.

Which op to use

- **Want to relabel dims while keeping the same data at each coordinate?** → `transpose` / `permute` / `movedim`.
- **Want to reinterpret the flat byte stream under a new shape?** → `view` / `reshape`.
- **Want to do both** (e.g., transpose then flatten)? → transpose first, then `reshape`. The reshape will copy under the hood, producing a contiguous buffer in the new logical order.

When does a copy happen?

op	copies?
`transpose`, `permute`, `swapdims`, `movedim`	no (metadata only)
`unsqueeze`, `squeeze`	no
`view` on a compatible-strided tensor	no
`view` on an incompatible-strided tensor	**errors** (no copy attempted)
`reshape` when strides permit	no
`reshape` when strides forbid (e.g., after permute)	**yes, silently**
`flatten` on contiguous tensor	no
`flatten` on non-contiguous tensor	yes
`.contiguous()` on contiguous tensor	no
`.contiguous()` on non-contiguous tensor	**yes, explicitly**
`.clone()`	yes, always
`.to(other_device)` (different device)	yes
`.to(other_dtype)` (different dtype)	yes

The rule: a copy happens when the operation needs a new storage layout. Pure axis relabelling does not. Anything that materialises a fresh row-major buffer (`contiguous`, `clone`, reshape-after-permute, dtype/device

change) does.

Practical rules

- Default to `view` when you know the tensor is contiguous and you only want to merge / split dims.
- Reach for `.contiguous().view(...)` when you want the copy to be visible in code.
- Use `reshape` for prototyping; replace with `.contiguous().view(...)` when reviewing for hot paths.
- For complex reorderings, use `einops.rearrange` — it makes the intent legible:

```
from einops import rearrange

# Instead of: x.permute(0, 2, 1, 3).contiguous().view(B, T, H * D)
x = rearrange(x, 'b h t d -> b t (h d)')
```

Same underlying ops.

- For any shape change after a transpose / permute / slice, assume a copy will happen unless you have specifically reasoned about stride compatibility.

The numpy footnote

NumPy's `reshape` accepts an `order=` parameter (`'C'` for row-major, `'F'` for column-major) that controls the traversal direction. PyTorch does not — it only supports row-major. F-order behaviour can be emulated in PyTorch by transposing before *and after* the reshape (reshaping to the reversed target shape in between).

tl;dr

- A tensor is **storage + shape + strides**. Axis-moving ops change shape and strides; storage stays.
- **`reshape` walks the logical tensor row-major** and lays out the new shape row-major. For contiguous tensors this is a view; for non-contiguous tensors it copies.
- **`view` is `reshape` without the copy option.** Errors if a copy would be needed.
- **Transpose preserves data semantics; reshape reinterprets the flat logical element stream.** They produce different tensors even at the same shape.
- The only way a copy happens is when an op needs a new storage layout (`reshape` after a permute, `contiguous`, dtype/device change, `clone`, `flatten` of non-contiguous).
- Use `view` (or `.contiguous().view()`) when allocations should be visible; use `reshape` when convenience matters; use `einops.rearrange` when the reorder is complex enough to be unreadable.