Mastering PyTorch and Lightning

A Step-by-Step Practical Guide with **QA**

Aghiles Kebaili

Mastering PyTorch and Lightning

A Step-by-Step Practical Guide with QA

Aghiles Kebaili

Portfolio: arksyd96.github.io

https://arksyd96.github.io

Companion GitHub repository:

code, notebooks, and practical resources

https://github.com/Arksyd96/mastering-pytorch-lightning-book

About the Author

Aghiles Kebaili

Aghiles Kebaili holds a PhD in Artificial Intelligence and specializes in deep learning, computer vision, and generative AI for medical imaging. He works as a Computer Vision Research Engineer at a renowned cancer research center in France, where he designs and evaluates advanced neural architectures for complex medical-image analysis problems.

His doctoral research explored generative models for predicting cancer progression from multimodal data. His broader expertise includes representation learning, variational autoencoders, diffusion models, multimodal image synthesis, tumor segmentation, and learning from limited clinical datasets. He has authored multiple peer-reviewed scientific publications as a primary author, including studies on deep generative data augmentation, medical-image synthesis, and predictive modeling of brain tumor evolution. He currently collaborates with Institut Curie in Paris on the design and development of generative deep-learning approaches for harmonizing multicenter PET images.

Drawing on his experience in deep learning, Python, and NumPy, and more than six years working with PyTorch and PyTorch Lightning, he brings the lessons learned from designing research-grade AI systems into this book. Its practical approach emphasizes clear architecture, reliable implementation patterns, testability, reproducibility, and scalable training, helping readers turn sound ideas into maintainable AI projects.

Research, Portfolio, and Companion Code

Portfolio: arksyd96.github.io
https://arksyd96.github.io

Google Scholar: Aghiles Kebaili's publication profile
https://scholar.google.com/citations?user=Sp3Q6LQAAAAJ

Companion repository: Mastering PyTorch and Lightning companion repository
https://github.com/Arksyd96/mastering-pytorch-lightning-book

Contents

MODULE 0:

THE INDUSTRIAL DEEP LEARNING LANDSCAPE

CHAPTER 0

Genesis, Architecture, and Environment Setup

0.1 The Origins: Why PyTorch?

Before writing a single line of code, it helps to understand the ecosystem surrounding the model. Deep learning engineering combines mathematics with software architecture, data movement, and hardware-aware execution.

In the early days of deep learning, frameworks like TensorFlow 1.x relied on *static computation graphs*. Engineers had to define the entire architecture of a neural network before feeding any data into it. If there was a bug, it was notoriously difficult to track down because the execution was locked in a compiled black box.

PyTorch was co-created by researchers at Facebook AI Research (FAIR) in 2016 and publicly launched in 2017. It helped popularize the **dynamic computation graph** and Define-by-Run workflow. The graph is created as Python operations execute. This makes it possible to inspect intermediate tensors, use standard Python debugging tools, and vary execution according to the input.

PyTorch has become widely used across AI research and production, bridging the gap between rapid prototyping and optimized deployment. Since the release of PyTorch 2.0, `torch.compile` has provided a native Python path to compiler optimization through technologies such as TorchInductor and Triton.

0.2 The Lightning Evolution: Scaling Deep-Learning Workflows

If PyTorch is so powerful, why do we need a framework like PyTorch Lightning?

Standard PyTorch scripts can become difficult to maintain as projects grow. Engineering concerns such as CPU/GPU transfers, training loops, distributed computing, and metric logging can become entangled with the research logic that defines the neural network itself.

PyTorch Lightning is a structured framework built on top of PyTorch. It encourages a modular design that separates engineering boilerplate from research code. In compatible projects, Trainer configuration can change devices, distributed strategies, and numerical precision without altering the model's core mathematical logic.

0.3 Industrial Use Cases: What Will You Build?

Mastering this stack opens the door to engineering state-of-the-art applications. Understanding the foundations laid in this ebook will provide you with the vocabulary and mechanics required to enter the field. Once you master tensors, computational graphs, and training loops, you will understand how the industry tackles standard problems across various domains:

- **Computer Vision:** Moving beyond basic tutorials to understand how neural networks process pixel arrays for image classification, object detection, and feature extraction.
- **Natural Language Processing (NLP):** Grasping the data pipelines and embedding mechanics that process text sequences for sentiment analysis or translation.
- **Structured Data & Time Series:** Utilizing deep learning for complex recommendation systems and predictive forecasting where traditional machine learning models reach their limits.

0.4 The GPU Matrix: Truth About CUDA & cuDNN

Hardware acceleration is essential for many deep learning workloads. Training efficiently often requires an accelerator. However, configuring a GPU environment is where many beginners struggle.

Industrial Reality Check

You will read countless tutorials telling you to manually download and install the NVIDIA CUDA Toolkit and cuDNN libraries on your operating system. **Do not do this.**

When you install PyTorch through an official binary distribution with CUDA support, the required CUDA runtime components are installed with the package. To get your GPU working, you must ensure that **your host machine's NVIDIA graphics driver supports the CUDA runtime used by PyTorch.**

1. Open your terminal and run `nvidia-smi`.
2. Look at the top right of the output for `CUDA Version: X.Y`. This is the *maximum* version of CUDA your current driver supports.
3. As a general rule, as long as the PyTorch version you download requires a CUDA version equal to or lower than the one shown by `nvidia-smi`, it will work out of the box.

> *Note:* Thanks to NVIDIA Forward Compatibility, a slightly older driver can sometimes run a newer CUDA runtime embedded in PyTorch, but keeping your driver updated to meet or exceed the PyTorch requirement remains the golden rule for avoiding industrial bugs.

0.5 Setting Up Your Development Environment

Let's set up a clean, professional workspace.

Follow Along with the Companion Repository

Keep the companion repository open while working through the hands-on chapters.
https://github.com/Arksyd96/mastering-pytorch-lightning-book
Running the corresponding code alongside the explanations makes it easier to reproduce each step, inspect intermediate results, and experiment with the complete examples.

Step 1: Virtual Environment Never install PyTorch in your global Python environment. Always use a virtual environment to avoid dependency conflicts.

```
python -m venv pytorch_env        # Or use conda create -n pytorch_env python
    =3.11
source pytorch_env/bin/activate   # On Linux/macOS
pytorch_env\Scripts\activate      # On Windows
```

Step 2: Installation Go to the official PyTorch installation selector.

https://pytorch.org/get-started/locally/

Copy the command generated for your operating system and accelerator. The available CUDA builds change over time. A current CUDA 12.6 wheel command follows this pattern:

```
pip install torch torchvision torchaudio --index-url https://download.pytorch.
    org/whl/cu126
```

Step 3: Verification Verify that PyTorch can successfully communicate with your hardware:

```python
import torch

print(f"PyTorch Version: {torch.__version__}")
print(f"CUDA Available: {torch.cuda.is_available()}")

if torch.cuda.is_available():
    print(f"Device Name: {torch.cuda.get_device_name(0)}")
```

Step 4: The Workspace : Jupyter Notebooks For the first two modules of this book, following our Active Learning methodology, we will heavily use Jupyter Notebooks. Notebooks allow you to execute Python code cell by cell, keeping tensors loaded in RAM while you modify the next block of code.

Install it within your virtual environment:

```
pip install jupyterlab
jupyter lab
```

Create a new notebook, initialize your first tensor, and get ready to dive into the core architecture of deep learning.

MODULE 1:

PYTORCH CORE FOUNDATIONS

CHAPTER 1

The Anatomy of Tensors

Tensors are the core data structure in PyTorch. A `torch.Tensor` is more than a multidimensional array: it combines a memory buffer with shape, stride, dtype, device, and Autograd metadata. Together, these properties determine how its values are stored, interpreted, and processed.

Chapter Roadmap

We will build the tensor abstraction from the bottom up. First, we will inspect the relationship between tensor metadata and storage. We will then compare copied and shared memory, control numerical precision, and move data between devices. The second half of the chapter focuses on views, dimensions, strides, and physical memory layouts.

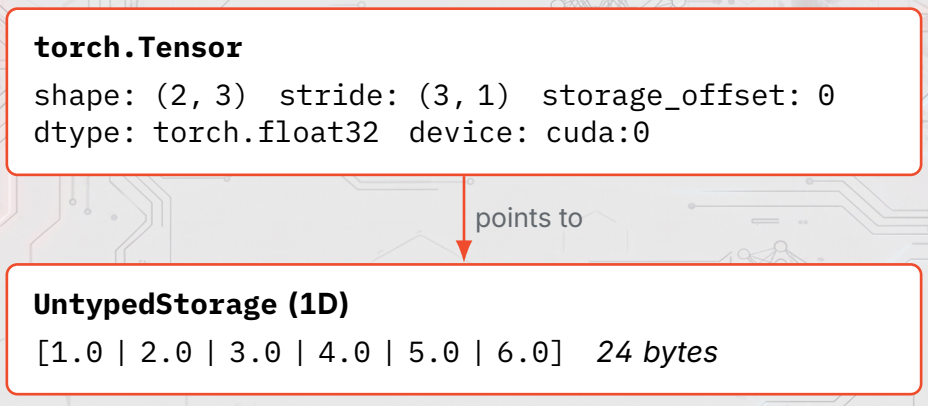
1 Internal Architecture & Memory Allocation

1.1 Step 1: Connect the Tensor Header to Storage

A PyTorch tensor is decoupled into two distinct entities:

1. **The `Storage` (or `UntypedStorage`):** A contiguous, one-dimensional sequential block of memory containing the raw bytes of data.
2. **The Tensor Header:** A logical view over this storage that defines how to interpret the linear byte stream. It stores:
   - **shape (or `size`):** The dimensions of the tensor $(D_0, D_1, \ldots, D_{N-1})$.
   - **`stride`:** The memory step size, meaning the number of elements to skip in the `Storage` to advance one step along a specific dimension.
   - **`storage_offset`:** The index of the first element in the `Storage`.
   - **`dtype`:** The scalar type and binary encoding of the elements.
   - **`device`:** The target hardware device: CPU, CUDA GPU, MPS, and others.

The following diagram visualizes the strict separation between the logical tensor header and the underlying storage:



Before comparing tensor-creation functions, inspect the main properties of a small tensor:

```python
import torch

matrix = torch.arange(6, dtype=torch.float32).reshape(2, 3)

print("Shape:", matrix.shape)                  # (2, 3)
print("Stride:", matrix.stride())              # (3, 1)
print("Storage offset:", matrix.storage_offset())  # 0
print("Dtype:", matrix.dtype)                  # torch.float32
print("Device:", matrix.device)                # cpu
print("Allocated data bytes:",
      matrix.numel() * matrix.element_size())   # 24
```

This is a useful first step when debugging tensor code. Shape describes the logical dimensions, stride describes how indexing moves through storage, and `numel()` multiplied by `element_size()` gives the number of bytes used by the tensor elements.

1.2 Step 2: Distinguish Copies from Shared Memory

PyTorch provides several ways to create tensors, with different copying and memory-sharing behavior:

- **`torch.tensor(data)`:** Allocates new storage and copies the source data.
- **`torch.as_tensor(data)` / `torch.from_numpy(arr)`:** Reuses the source memory when possible. The resulting tensor and source array can then observe the same mutations.
- **`torch.empty(*size)`:** Allocates storage without initializing its values. Its initial contents are undefined and must be overwritten before use.

- **`torch.zeros(*size)` / `torch.ones(*size)` / `torch.randn(*size)`:** Allocates and initializes values to zero, one, or samples from a normal distribution $\mathcal{N}(0, 1)$.

The following example shows the difference between sharing a NumPy array and creating an independent copy:

```python
import numpy as np
import torch

# 1. Zero-copy vs explicit allocation
np_array = np.array([1.0, 2.0, 3.0], dtype=np.float32)

# Memory sharing: modifying t_shared modifies np_array
t_shared = torch.from_numpy(np_array)
# Alternative: torch.as_tensor(np_array)

# Explicit copy: allocation of a new independent Storage
t_copy = torch.tensor(np_array)

# Verifying memory sharing
t_shared[0] = 99.0
print(f"NumPy array after mutation: {np_array[0]}")  # 99.0
print(f"Copied tensor (independent): {t_copy[0]}")   # 1.0

assert np_array[0] == t_shared[0].item()
assert t_copy[0].item() == 1.0
```

The mutation confirms that `from_numpy()` reused the NumPy array's memory. This can avoid a copy, but changes made through either object are visible to the other. Use an explicit copy when that shared behavior is not desirable.

1.3 Step 3: Control dtype and Memory Footprint

The choice of `dtype` affects memory usage, transfer bandwidth, numerical range, and access to specialized hardware such as Tensor Cores.

Note on `torch.bool`

In standard PyTorch storage, each `torch.bool` element occupies one byte. Boolean values do not have the sign, exponent, and mantissa fields used by floating-point formats. Specialized implementations, including custom Triton kernels, may bit-pack masks when a smaller representation is required.

dtype	Size	Sign / Exponent / Mantissa	Primary use case
`torch.float32`	32 bits (4 B)	1 / 8 / 23 bits	Standard deep learning precision (FP32).
`torch.float64`	64 bits (8 B)	1 / 11 / 52 bits	Scientific computing, physical simulations, and nonlinear optimization.
`torch.float16`	16 bits (2 B)	1 / 5 / 10 bits	Legacy mixed precision. Prone to underflow and overflow and commonly paired with loss scaling.
`torch.bfloat16`	16 bits (2 B)	1 / 8 / 7 bits	Modern mixed precision with the dynamic range of FP32.
`torch.int64`	64 bits (8 B)	Signed integer	Class indices, embeddings, and tensor indexing.
`torch.int32`	32 bits (4 B)	Signed integer	Low-level indexing, masks, and CPU memory economy.
`torch.bool`	8 bits (1 B)	N/A	Binary masking, attention masks, and conditional selection.

The next experiment makes the memory tradeoff measurable rather than abstract. Both tensors contain the same number of elements, but BF16 uses half as many bytes as FP32:

```python
import torch

# Compare identical shapes stored with different dtypes.
weights_fp32 = torch.empty((1024, 1024), dtype=torch.float32)
weights_bf16 = torch.empty((1024, 1024), dtype=torch.bfloat16)

fp32_bytes = weights_fp32.numel() * weights_fp32.element_size()
bf16_bytes = weights_bf16.numel() * weights_bf16.element_size()
print(fp32_bytes, bf16_bytes)
assert bf16_bytes == fp32_bytes // 2

# Cast explicitly when an operation requires another dtype.
x = torch.arange(10, dtype=torch.int64)
x_float = x.to(dtype=torch.float32)
assert x_float.dtype == torch.float32
```

Lower precision reduces storage and data-transfer costs, but it also changes rounding behavior and numerical range. Choosing a dtype is therefore both a performance decision and a numerical one.

1.4 Step 4: Move Tensors Between Host and Accelerator

The interaction between host RAM (CPU) and VRAM (GPU) represents one of the major bottlenecks in deep learning. Modern industrial pipelines also use multi-GPU systems rather than a single monolithic device.

The following example selects an available device and uses pinned host memory only for a CUDA transfer. Combined with `non_blocking=True`, pinned memory allows the runtime to perform an asynchronous copy when the surrounding workload supports it:

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
tensor = torch.zeros((4, 512, 512), dtype=torch.float32)

if device.type == "cuda":
    host_tensor = tensor.pin_memory()
    device_tensor = host_tensor.to(device, non_blocking=True)
else:
    device_tensor = tensor.to(device)

print(device_tensor.device)
assert device_tensor.shape == tensor.shape
```

Production Preview: DeviceMesh

Distributed tensor layouts extend the same device concept across several processes. A two-dimensional `DeviceMesh`, for example, can describe separate data-parallel and tensor-parallel axes. Executable examples are postponed until the scaling chapters, where process launch, rank assignment, and collective communication are introduced together.

Checkpoint: Tensor Construction

At this point, you should be able to explain whether an operation allocates or shares memory, calculate a tensor's element storage in bytes, identify its dtype and device, and recognize why host-to-device transfer policy belongs to the data pipeline.

2 Advanced Dimension & Stride Manipulation

Memory-Layout Roadmap

The examples in this section separate two questions: did an operation change the tensor's logical dimensions, its physical memory layout, or both? We will begin with a contiguous matrix, inspect how strides change, resolve incompatible views, and then apply the same reasoning to image and feature tensors.

2.1 Step 1: Read Strides and Calculate an Offset

For an N-dimensional tensor with shape $(d_0, d_1, \ldots, d_{N-1})$, the stride S_k is the number of storage elements crossed when index i_k increases by one.

The linear address in the `Storage` for the element $(i_0, i_1, \ldots, i_{N-1})$ is calculated as

$$\text{Index}_{\text{storage}} = \texttt{storage\_offset} + \sum_{k=0}^{N-1} i_k S_k.$$

A tensor is strictly C-contiguous if its stride along the k-th dimension is the product of all subsequent dimensions:

$$S_k = \prod_{j=k+1}^{N-1} d_j, \qquad S_{N-1} = 1.$$

Let us observe how strides behave in practice when creating and reshaping a simple matrix:

```python
x = torch.arange(6, dtype=torch.int32).reshape(2, 3)

print("Shape:", x.shape)                # torch.Size([2, 3])
print("Strides:", x.stride())           # (3, 1)
print("Contiguous:", x.is_contiguous())  # True

row, column = 1, 2
offset = (
    x.storage_offset()
    + row * x.stride(0)
    + column * x.stride(1)
)
print("Calculated offset:", offset)       # 5
assert x[row, column] == x.reshape(-1)[offset]
```

The offset calculation turns the stride equation into a concrete address. Moving one row advances three elements, while moving one column advances one element.

2.2 Step 2: Compare Contiguous and Channels-Last Layouts

For some supported convolutional workloads, the standard C-contiguous format (B, C, H, W) is not the fastest physical layout. PyTorch supports the **Channels Last** memory format, which can improve performance when the model, kernels, accelerator, and numerical precision are compatible.

In this layout, the tensor is not C-contiguous, but it is contiguous according to the Channels Last specification. Its logical shape remains (B, C, H, W) even though its physical memory strides follow a channels-last arrangement.

The snippet below switches to Channels Last without altering the logical semantic shape of the tensor:

```python
import torch

# Standard C-contiguous layout (NCHW)
vision_tensor = torch.randn(32, 3, 224, 224)
print("Is C-contiguous:", vision_tensor.is_contiguous())  # True

# Convert to Channels Last for optimized Conv2D kernels
channels_last_tensor = vision_tensor.to(memory_format=torch.channels_last)

# Logical shape remains (B, C, H, W), but physical strides change.
print("Is C-contiguous:", channels_last_tensor.is_contiguous())  # False
print(
    "Is Channels-Last contiguous:",
    channels_last_tensor.is_contiguous(memory_format=torch.channels_last),
)  # True

assert channels_last_tensor.shape == vision_tensor.shape
assert channels_last_tensor.stride() != vision_tensor.stride()
```

The dimension order remains NCHW, while the physical strides change. This is different from `permute()`, which reorders the logical axes seen by the program.

2.3 Step 3: Choose Between `view()`, `reshape()`, and `contiguous()`

- **`view(*shape)`:** Returns a different shape over the same storage. The requested shape must be compatible with the current size and strides. Otherwise, PyTorch raises a `RuntimeError`.
- **`contiguous()`:** Returns the original tensor when its layout is already compatible. Otherwise, it allocates contiguous storage and copies the data.
- **`reshape(*shape)`:** Returns a view when possible and performs a copy when the existing strides cannot represent the requested shape.

This example shows why a transposed tensor may be incompatible with `view()`, and how `reshape()` or `contiguous()` handles that layout:

```python
import torch

# Create a (2, 3) matrix
A = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)

# Transposition changes the strides without moving a byte
A_t = A.t()
print("A_t shape:", A_t.shape)       # (3, 2)
print("A_t strides:", A_t.stride())  # (1, 3), non-contiguous

# Attempting an incompatible view raises RuntimeError
try:
    A_t.view(-1)
except RuntimeError as error:
    print(f"Error caught: {error}")

```

```
# Solution 1: force contiguity before viewing
flat_view = A_t.contiguous().view(-1)

# Solution 2: reshape handles the copy automatically when needed
flat_reshaped = A_t.reshape(-1)

assert torch.equal(flat_view, flat_reshaped)
```

Use `view()` when the result must share storage with the source tensor. Use `reshape()` when either a view or a copy is acceptable. Call `contiguous()` when the layout conversion and possible copy should be explicit.

2.4 Step 4: Reorder Dimensions with `transpose()` and `permute()`

- **`transpose(dim0, dim1)`:** Swaps exactly two specific dimensions.
- **`permute(*dims)`:** Rearranges all dimensions according to an arbitrary permutation order.

The following example uses different height and width values so that each transformation is visible in the printed shape:

```
images = torch.randn(32, 3, 224, 160)

swapped_hw = images.transpose(2, 3) # Swap the height and width axes
print(swapped_hw.shape)  # (32, 3, 160, 224)

nhwc = images.permute(0, 2, 3, 1) # Rearrange NCHW to NHWC logically
print(nhwc.shape)  # (32, 224, 160, 3)
```

2.5 Step 5: Add and Remove Singleton Dimensions

- **`unsqueeze(dim)`:** Inserts a singleton dimension of size one at the specified index.
- **`squeeze(dim=None)`:** Removes every dimension of size one, or only the specified dimension when `dim` is passed.

The following code injects a singleton dimension of length one and removes it again afterward:

```
feature_map = torch.randn(64, 128)  # (Batch, Features)

# Inject a sequence/time dimension: (Batch, 1, Features)
expanded_1 = feature_map.unsqueeze(1)
# Idiomatic injection via None / np.newaxis
expanded_2 = feature_map[:, None, :]
# Remove the singleton dimension
collapsed = expanded_1.squeeze(dim=1)  # Returns to (64, 128)

assert expanded_1.shape == expanded_2.shape and torch.equal(collapsed, feature_map)
```

2.6 Step 6: Flatten and Restore Structured Features

Spatial features are often flattened before a fully connected layer. `unflatten()` can restore the original structure when the dimensions are known:

```
import torch

# Convolutional features: (Batch=16, Channels=64, H=7, W=7)
conv_out = torch.randn(16, 64, 7, 7)

# Flatten for ingestion into a Linear layer: (B, C * H * W)
flat = torch.flatten(conv_out, start_dim=1)  # (16, 3136)

# Inverse structured unfolding
unflat = flat.unflatten(dim=1, sizes=(64, 7, 7))  # (16, 64, 7, 7)

assert torch.equal(unflat, conv_out)
```

Checkpoint: Views and Layouts

Shape answers what the program sees. Stride answers how indices move through storage. Contiguity answers whether a requested traversal matches a recognized dense layout. Keeping these three ideas separate makes reshaping errors much easier to diagnose.

3 Gotchas & Reality Checks: Production Pitfalls

Gotcha 1: Graph Accumulation and VRAM Memory Leaks

During training, `loss` is a tensor whose `grad_fn` links it to the current Autograd graph. Accumulating loss tensors retains references to graphs from previous batches, including intermediate values saved for the backward pass. In contrast, `loss.item()` extracts the scalar value without retaining its Autograd history, making it suitable for metric aggregation. Computational graphs and `grad_fn` are covered in more detail in Chapter 3.

The following helper makes this behavior observable by counting the Autograd nodes reachable from a tensor. It returns zero for non-tensor values and for tensors that are not connected to a graph:

```
import torch

# Count the unique Autograd nodes reachable from a tensor.
def count_graph_nodes(tensor):
    if not isinstance(tensor, torch.Tensor) or tensor.grad_fn is None:
        return 0

    visited_nodes = set()

    def traverse(grad_fn):
```

```
        if grad_fn is None or grad_fn in visited_nodes:
            return
        visited_nodes.add(grad_fn)

        # Each entry contains the next function and an output index.
        for next_fn, _ in getattr(grad_fn, "next_functions", []):
            traverse(next_fn)

    traverse(tensor.grad_fn)
    return len(visited_nodes)
```

The next example compares metric accumulation with and without `item()`. The first count depends on the model architecture and the number of processed batches because `graph_total` remains connected to their graphs. The second count is zero because `scalar_total` has no Autograd history. This helper shows how the graph grows. It does not measure VRAM consumption or the memory occupied by saved intermediate values.

```
graph_total = 0.0
# Accumulating loss tensors retains their Autograd graphs.

for data, target in dataloader:
    output = model(data)
    loss = criterion(output, target)
    graph_total = graph_total + loss

# item() extracts scalar values with no Autograd history.
scalar_total = 0.0

for data, target in dataloader:
    output = model(data)
    loss = criterion(output, target)
    scalar_total += loss.item()

graph_nodes = count_graph_nodes(graph_total)
scalar_nodes = count_graph_nodes(scalar_total)

print("Nodes kept by graph_total:", graph_nodes)
print("Nodes kept by scalar_total:", scalar_nodes)

assert graph_nodes > 0
assert scalar_nodes == 0
```

Gotcha 2: In-Place Operations and Autograd Errors

Operations ending with an underscore, such as `add_()`, `mul_()`, and `zero_()`, modify an existing tensor instead of creating a separate result. This can reduce temporary allocations, but it requires care when Autograd has saved that tensor for the backward pass.

In the following example, `objective` depends on the original value of `y`. Modifying `y` before `backward()` makes the saved value invalid:

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x * 2
objective = y.pow(2).sum()  # Backward saves information about y.

# The saved tensor is modified before backward uses it.
y.add_(1.0)

try:
    objective.backward()
except RuntimeError as error:
    print(f"Autograd detected the in-place modification: {error}")
```

PyTorch tracks a version counter for tensors used by Autograd. When a saved tensor changes unexpectedly, the backward pass raises an error instead of computing gradients from inconsistent values. Prefer out-of-place operations unless the memory benefit of an in-place update is necessary and has been verified.

Gotcha 3: Implicit Typing from NumPy to PyTorch

NumPy commonly creates floating-point arrays with dtype `float64`, while most PyTorch modules use `float32` parameters by default. `torch.from_numpy()` preserves the NumPy dtype, so the resulting tensor may not match the model.

The following example checks the imported dtype and converts it before running the layer:

```
import numpy as np
import torch

f32_model = torch.nn.Linear(10, 5)  # Model expects torch.float32 inputs

raw_data = np.random.randn(10, 10)        # np.float64 by default
tensor_data = torch.from_numpy(raw_data)  # torch.float64

try:
    forward_f32 = f32_model(tensor_data)  # This will raise a RuntimeError due to dtype
     mismatch
except RuntimeError as error:
    print(f"Error occurred: {error}")

# Explicitly cast before passing the tensor into the model
tensor_data = tensor_data.to(torch.float32)
forward_f32 = f32_model(tensor_data)  # This will now work correctly
assert forward_f32.dtype == torch.float32

print("Forward pass successful with dtype:", forward_f32.dtype)  # torch.float32
```

4 QA: Multiple Choice Questionnaire

Question 1: What is the fundamental memory difference between `torch.from_numpy(arr)` and `torch.tensor(arr)`?

- ☐ **A)** Both allocate new memory, but `from_numpy` operates faster.
- ☐ **B)** `torch.from_numpy` shares the memory block, while `torch.tensor` copies the data into a new buffer.
- ☐ **C)** `torch.from_numpy` can only create CPU tensors, whereas `torch.tensor` defaults to the GPU.
- ☐ **D)** `torch.from_numpy` creates an immutable tensor.

Question 2: Why can `x.transpose(0,1).view(-1)` raise a `RuntimeError`?

- ☐ **A)** `transpose()` deletes the underlying data.
- ☐ **B)** `transpose()` alters the strides without moving the bytes, making the requested `view()` incompatible with the resulting layout.
- ☐ **C)** `view(-1)` is mathematically undefined for transposed tensors.
- ☐ **D)** `transpose()` casts the tensor to `float64`.

Question 3: Why is `bfloat16` often preferred to `float16` for training large networks?

- ☐ **A)** It is twice as fast on every CPU architecture.
- ☐ **B)** It has a larger mantissa and therefore greater decimal precision.
- ☐ **C)** It retains the exponent size and dynamic range of `float32`, reducing overflow and underflow risks.
- ☐ **D)** It natively supports dynamic graph allocation inside CUDA cores.

Question 4: What host-memory property is normally required to overlap a CPU-to-CUDA copy reliably with other work?

- ☐ **A)** The network must not contain recurrent layers.
- ☐ **B)** The source tensor should reside in page-locked (pinned) host memory.
- ☐ **C)** The target must be a dedicated NVIDIA Tensor Core.
- ☐ **D)** The transferred tensor must be one-dimensional.

Question 5: How does `permute()` differ from `to(memory_format=torch.channels_last)`?

- ☐ **A)** `permute()` changes the logical dimension order, while `to()` changes the physical layout and preserves the logical shape.
- ☐ **B)** They are identical operations.
- ☐ **C)** `permute()` is in-place, while `to()` clones the computational graph.
- ☐ **D)** Channels Last is only supported on Apple Silicon.

Answers & Explanations

Answer 1: B

`torch.from_numpy(arr)` creates a tensor that shares the existing NumPy memory block. This is an $O(1)$ operation without a data copy, so mutations are visible from both objects. `torch.tensor(arr)` allocates an independent buffer and copies the values.

Answer 2: B

`transpose()` changes the tensor's dimensions and strides without physically reordering its `Storage`. The requested flattened `view()` is therefore incompatible with that stride layout. `reshape()` can perform a copy automatically, while `contiguous()` explicitly creates a compatible storage layout.

Answer 3: C

`float16` uses five exponent bits, which limits its numerical range. `bfloat16` uses the same eight-bit exponent as `float32`, reducing the risk of overflow and underflow during training. The tradeoff is a shorter mantissa and lower numerical precision.

Answer 4: B

Pinned host memory allows CUDA to perform efficient asynchronous transfers from the CPU. Use `tensor.pin_memory()` for an individual tensor or `pin_memory=True` in a `DataLoader`. Passing `non_blocking=True` requests an asynchronous copy, but useful overlap still depends on the surrounding execution pipeline.

Answer 5: A

`permute()` reorders the logical dimensions, changing the shape from (B, C, H, W) to (B, H, W, C). A Channels Last conversion preserves the logical (B, C, H, W) shape and changes only the physical strides used to store the data.