Deep Learning Architectures for Scientific Machine Learning

A Comprehensive PyTorch Guide

Professor Vectorex Gradiens

Wednesday 3rd December, 2025

Version 1.0

Disclaimer

Important Notice: Apart from this paragraph, every single word in this document was written with the assistance of Large Language Models. This document is intended as an aid for learning PyTorch, in particular for implementing deep learning architectures presented in the course "AI in the Sciences and Engineering" held at ETH Zürich.

The content is extremely likely to contain errors, so use it with care, knowing that this isn't reviewed or approved by domain experts. Always verify critical implementations and consult primary sources for production use.

Giovanni Guidarini, December 2025

Acknowledgments: This textbook builds upon the collective wisdom of the PyTorch community, scientific machine learning researchers, and educators worldwide. Special thanks to the developers of PyTorch, the authors of foundational papers in deep learning, and students whose questions shaped this material.

Preface

My dear student,

Welcome. It is a distinct pleasure to guide you on this journey. You are already embedded in the world of scientific computing, a field of precision, rigor, and computational elegance. You will find that deep learning is not so different. At its heart, it is a powerful form of high-dimensional function approximation, built on the familiar foundations of linear algebra and calculus. The "magic" is simply that we have found a way to make these function approximators (*neural networks*) trainable on a massive scale.

Your lack of PyTorch experience is not a hindrance; it is an opportunity. You arrive with no bad habits. We will build your knowledge from the ground up, with the same care one takes in formulating a proof or designing a robust simulation. PyTorch is our "language" for this. It is expressive, powerful, and, once you are fluent, a genuine joy to use.

This textbook is comprehensive, spanning from the fundamental `torch.Tensor` to state-of-the-art architectures like Transformers and Neural Operators. Each chapter builds systematically, combining rigorous theory with practical implementation. You will not merely learn to use PyTorch—you will understand *why* it works the way it does, and you will develop the intuition to design your own architectures for scientific problems.

How to Use This Book:

- **Theory sections** provide mathematical foundations and intuition. Do not skip these—they are the bedrock upon which everything rests.
- **Implementation sections** show you the "PyTorch way"—idioms, best practices, and common pitfalls. Type out the code yourself; reading is not enough.
- **Exercises** range from elementary to research-level. Attempt all of them. Struggle is where learning happens.
- **Colored boxes** highlight key insights:

- Professor's Notes for expert intuition
- PyTorch Idioms for best practices
- Warnings for common mistakes
- Expert Tricks for advanced techniques
- Scientific Applications showing real-world use

Treat this not just as a technical exercise, but as an art. The code we write, like the LaTeX that renders this page, can and should be clean, precise, and beautiful.

Let us begin.

Professor Vectorex Gradiens
ETH Zürich
December 2025

Contents

Part I

Foundations of PyTorch

Chapter 1

Tensor Foundations and Computation

"In the beginning was the tensor, and the tensor was with gradient, and the tensor was computation."
— Ancient PyTorch Proverb

1.1 Introduction: Why Tensors?

Everything in PyTorch—from your input data (a simulation mesh, a time-series, an image) to the parameters of your neural network (weights and biases)—is represented as a **tensor**.

If you have used NumPy, you are already familiar with the core concept: the `ndarray`. A PyTorch tensor is, at its essence, a multi-dimensional array. So why not just use NumPy?

Three fundamental reasons:

1. **GPU Acceleration:** PyTorch tensors can be effortlessly moved to Graphics Processing Units (GPUs). This enables *massive* parallelization of computations, transforming operations that would take hours on a CPU into minutes or seconds. For scientific computing and deep learning at scale, this is essential.

2. **Automatic Differentiation:** This is the beating heart of modern deep learning. PyTorch can automatically compute gradients (vectors of partial derivatives) of any output with respect to any input. This mechanism, called `autograd`, enables gradient-based optimization. We dedicate Chapter 2 to this crucial concept.

3. **Ecosystem Integration:** PyTorch tensors integrate seamlessly with a vast ecosystem of neural network layers (`torch.nn`), optimizers (`torch.optim`), and

utilities specifically designed for deep learning workflows.

In this chapter, we build from first principles. We begin with the mathematical definition of tensors, progress through their computational properties, and culminate in mastering the PyTorch tensor API with all its subtleties and power.

1.2 Theoretical Foundations

1.2.1 Mathematical Definition

Tensor

A **tensor** is a multi-dimensional array of numerical values. Formally, a tensor of order n (also called rank n) is an element of the tensor product of n vector spaces:

$$\mathcal{T} \in V_1 \otimes V_2 \otimes \cdots \otimes V_n$$

More practically, for our purposes:

- A **scalar** is a 0D tensor (a single number): $x \in \mathbb{R}$
- A **vector** is a 1D tensor: $\boldsymbol{x} \in \mathbb{R}^n$
- A **matrix** is a 2D tensor: $\boldsymbol{X} \in \mathbb{R}^{m \times n}$
- A **3D tensor** might represent: $\boldsymbol{X} \in \mathbb{R}^{h \times w \times c}$ (image with height, width, channels)
- A **4D tensor** might represent: $\boldsymbol{X} \in \mathbb{R}^{b \times c \times h \times w}$ (batch of images)

The **shape** or **dimensions** of a tensor describe the size along each axis.

Why "Tensor" and Not Just "Array"?

The term "tensor" comes from differential geometry and physics, where tensors are geometric objects that transform in specific ways under coordinate changes. In deep learning, we use the term more loosely to mean "multi-dimensional array."

However, the connection is not entirely superficial: neural networks learn representations that should be invariant (or equivariant) to certain transformations—this is the geometric perspective on deep learning. When we design convolutional networks to be translation-equivariant or use group convolutions, we are encoding geometric structure into our "tensors."

1.2.2 Key Properties and Operations

Understanding tensor operations is crucial for both computational efficiency and numerical stability in scientific computing.

Element-wise Operations

Element-wise (or pointwise) operations apply a function independently to each element:

$$(\boldsymbol{A} + \boldsymbol{B})_{ij} = A_{ij} + B_{ij}$$

$$(\boldsymbol{A} \odot \boldsymbol{B})_{ij} = A_{ij} \cdot B_{ij} \quad \text{(Hadamard product)}$$

These operations are trivially parallelizable and form the basis of many neural network operations (ReLU, Sigmoid, etc.).

Reduction Operations

Reductions aggregate values along one or more dimensions:

$$\text{sum}(\boldsymbol{A}) = \sum_{i,j} A_{ij}, \quad \text{mean}(\boldsymbol{A}) = \frac{1}{mn} \sum_{i,j} A_{ij}$$

In deep learning, we frequently reduce over the batch dimension to compute average loss, or over spatial dimensions in global pooling.

Broadcasting

Broadcasting is perhaps the most powerful—and most misunderstood—feature of tensor operations. It allows operations between tensors of different shapes by automatically expanding dimensions.

Broadcasting Rules

Two tensors are **compatible for broadcasting** if, for each dimension (working right-to-left):

1. The dimensions are equal, OR
2. One of the dimensions is 1, OR
3. One of the tensors doesn't have this dimension (can be virtually expanded)

Example 1.2.1 (Broadcasting Examples).

$$\text{Shape } (3, 4) + \text{Shape } (4,) \to \text{Shape } (3, 4) \quad \checkmark$$
$$\text{Shape } (3, 1, 5) + \text{Shape } (3, 4, 1) \to \text{Shape } (3, 4, 5) \quad \checkmark$$
$$\text{Shape } (3, 4) + \text{Shape } (3,) \to \text{Error!} \quad \times$$

The last example fails because the shapes align as:

```
(3, 4)
   (3,)  <- This 3 aligns with 4, not compatible!
```

Silent Broadcasting Bugs

Broadcasting can cause subtle bugs that compile and run but produce incorrect results. Consider normalizing each feature of a dataset:

Wrong:

```python
X = torch.randn(100, 10)  # 100 samples, 10 features
mean = X.mean(dim=0)        # Shape: (10,)
X_normalized = X - mean     # Works! But wrong if you
    meant something else
```

Right (for per-sample normalization):

```python
mean = X.mean(dim=1, keepdim=True)  # Shape: (100, 1)
X_normalized = X - mean              # Broadcasts
    correctly
```

Always verify shapes explicitly during development!

1.2.3 Memory Layout and Contiguity

Understanding memory layout is critical for performance in scientific computing.

Contiguous Tensors

A tensor is **contiguous** if its elements are stored in a single, contiguous block of memory in row-major (C-style) order. Operations like `view()` require contiguous tensors, while operations like `reshape()` can work with non-contiguous tensors (by copying if necessary).

Memory Contiguity Rules

- **Transposing** makes tensors non-contiguous:

```python
x = torch.randn(3, 4)
y = x.t()   # Non-contiguous!
```

- **Check contiguity:** `tensor.is_contiguous()`
- **Make contiguous:** `tensor.contiguous()`
- **view() vs reshape():**
  - `view()` requires contiguous memory, returns a view (no copy)
  - `reshape()` works on any tensor, copies if necessary
  - Use `view()` when you know tensor is contiguous (faster, explicit)
  - Use `reshape()` when you're unsure (safer, more flexible)

1.2.4 Data Types and Numerical Precision

The choice of data type (`dtype`) has profound implications for both performance and numerical stability.

Table 1.1: Common PyTorch Data Types

PyTorch dtype	NumPy equivalent	Size	Range/Precision
`torch.float32`	`np.float32`	32 bits	$\sim 10^{-38}$ to 10^{38}, 7 digits
`torch.float64`	`np.float64`	64 bits	$\sim 10^{-308}$ to 10^{308}, 15 digits
`torch.float16`	`np.float16`	16 bits	$\sim 10^{-8}$ to 10^{4}, 3 digits
`torch.bfloat16`	—	16 bits	Same range as float32, 3 digits
`torch.int32`	`np.int32`	32 bits	-2^{31} to $2^{31}-1$
`torch.int64`	`np.int64`	64 bits	-2^{63} to $2^{63}-1$
`torch.bool`	`np.bool_`	8 bits	True or False

Float32 vs Float64 in Scientific ML

Traditional scientific computing often uses `float64` (double precision) for numerical stability. However, deep learning typically uses `float32` because:

1. **Memory:** Models can have billions of parameters; float32 halves memory usage
2. **Speed:** GPU operations are often 2-10× faster with float32
3. **Sufficient precision:** The stochastic nature of SGD acts as regularization; extra precision rarely helps

When to use float64:

- Physics-Informed Neural Networks (PINNs) computing high-order derivatives
- Ill-conditioned problems (e.g., inverting nearly-singular matrices)
- When numerical errors accumulate (very deep networks, long sequences)
- Scientific simulations requiring strict conservation laws

Modern approach: Mixed precision training uses float16 for speed and float32 for critical operations.

1.3 PyTorch Implementation Deep Dive

1.3.1 Creating Tensors: The Complete API

Let's explore every important way to create tensors in PyTorch, with rationale for each.

From Explicit Values

```python
import torch
import numpy as np

```

```python
# From Python lists
x = torch.tensor([1, 2, 3, 4])  # Shape: (4,), dtype inferred
    as int64
y = torch.tensor([[1.0, 2.0], [3.0, 4.0]])  # Shape: (2, 2),
    dtype: float32

# From NumPy arrays (shares memory if possible!)
np_array = np.array([1, 2, 3])
z = torch.from_numpy(np_array)  # Zero-copy if np_array is
    contiguous

# Explicitly specify dtype and device
x = torch.tensor([1, 2, 3], dtype=torch.float32, device='cuda
    ')
```

torch.tensor() vs torch.Tensor()

- `torch.tensor()` — Function that creates a new tensor, infers dtype
- `torch.Tensor()` — Class constructor, always creates `float32` tensor

Prefer `torch.tensor()` for clarity:

```python
x = torch.tensor([1, 2, 3])       # Infers dtype=int64
y = torch.Tensor([1, 2, 3])       # Forces dtype=float32
    (!!)
```

Initialization Patterns

```python
# Zeros and ones
zeros = torch.zeros(3, 4)                      # All zeros
ones = torch.ones(2, 3, dtype=torch.float64) # All ones
identity = torch.eye(5)                        # Identity
    matrix

# Uninitialized (faster, but contains garbage!)
uninitialized = torch.empty(1000, 1000)  # Use when you'll
    overwrite immediately

# From ranges
arange = torch.arange(0, 10, 2)       # [0, 2, 4, 6, 8]
linspace = torch.linspace(0, 1, 5)    # [0.0, 0.25, 0.5, 0.75,
    1.0]
```

Random Initialization

Random initialization is crucial for breaking symmetry in neural networks.

```python
# Uniform distribution [0, 1)
uniform = torch.rand(3, 4)

# Standard normal distribution N(0, 1)
normal = torch.randn(3, 4)

# Random integers
randint = torch.randint(0, 10, (3, 4))  # Random ints in [0,
    10)

# Random permutation (useful for shuffling)
perm = torch.randperm(10)  # Random permutation of 0..9
```

Reproducible Randomness

For reproducible experiments (critical in scientific computing):

```python
# Set global seed
torch.manual_seed(42)

# For CUDA
torch.cuda.manual_seed(42)
torch.cuda.manual_seed_all(42)  # For multi-GPU

# For full reproducibility (may impact performance)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

Note: Full reproducibility across different hardware is not guaranteed. Use seeds for debugging and validation, not for critical production randomness.

Creating Tensors with Same Shape

Often you need a new tensor with the same shape as an existing one:

```python
x = torch.randn(3, 4)

# Same shape, filled with zeros/ones
zeros_like = torch.zeros_like(x)
ones_like = torch.ones_like(x)

# Same shape, random
rand_like = torch.rand_like(x)
randn_like = torch.randn_like(x)

# Can override dtype
int_version = torch.zeros_like(x, dtype=torch.int64)
```

1.3.2 Tensor Operations: The Idioms

Indexing and Slicing

PyTorch indexing works like NumPy, but with some powerful extensions:

```python
x = torch.randn(4, 5, 6)

# Basic slicing
first_row = x[0]          # Shape: (5, 6)
submatrix = x[1:3, :, 2:4]  # Shape: (2, 5, 2)

# Advanced indexing
indices = torch.tensor([0, 2, 3])
selected = x[indices]    # Shape: (3, 5, 6)

# Boolean masking
mask = x > 0
positive_values = x[mask]  # 1D tensor of positive values

# Ellipsis (...) for all remaining dimensions
batch_first = x[..., 0]  # Equivalent to x[:, :, 0]
```

In-Place Operations

Operations ending with underscore (_) modify tensors in-place:

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

# Out-of-place (creates new tensor)
y = x.add(1)      # x unchanged, y = x + 1

# In-place (modifies x)
x.add_(1)         # x modified: x = x + 1

# Common in-place ops
x.zero_()         # Fill with zeros
x.fill_(3.14)     # Fill with value
x.clamp_(0, 1)    # Clamp to range
```

When to use in-place:

- Memory savings when working with large tensors
- Optimizer updates: `param.data.sub_(lr * grad)`

When NOT to use in-place:

- On tensors that require gradients (breaks autograd!)
- When you need the original value later
- In complex computational graphs (debugging nightmare)

Shape Manipulation

```
x = torch.randn(2, 3, 4)

# View vs Reshape (discussed earlier)
y = x.view(6, 4)        # Requires contiguous, returns view
z = x.reshape(6, 4)     # Always works, may copy

# Squeeze and unsqueeze (add/remove dimensions of size 1)
a = torch.randn(1, 3, 1, 4)
b = a.squeeze()          # Shape: (3, 4) - removes all size-1
   dims
c = a.squeeze(0)         # Shape: (3, 1, 4) - removes only dim
   0
d = c.unsqueeze(1)       # Shape: (3, 1, 1, 4) - adds dim at
   position 1

# Transpose (swap two dimensions)
x = torch.randn(2, 3, 4)
y = x.transpose(0, 1)    # Shape: (3, 2, 4)
z = x.transpose(-1, -2) # Last two dims: (2, 4, 3)
```

```python

# Permute (general reordering of dimensions)
x = torch.randn(2, 3, 4, 5)  # (batch, channel, height, width
    )
y = x.permute(0, 2, 3, 1)    # (batch, height, width, channel
    )

# Flatten
x = torch.randn(2, 3, 4)
y = x.flatten()          # Shape: (24,)
z = x.flatten(1)         # Flatten from dim 1: (2, 12)
```

Dimension Ordering Conventions

Different deep learning frameworks use different conventions:

PyTorch (and many CV models):

- Images: (Batch, Channels, Height, Width) — `(B, C, H, W)`
- Sequences: (Batch, Sequence, Features) — `(B, S, F)`
- or (Sequence, Batch, Features) — `(S, B, F)` for RNNs

TensorFlow/Keras:

- Images: (Batch, Height, Width, Channels) — `(B, H, W, C)`

The PyTorch convention aligns with cuDNN's optimized implementations. Always check documentation when using pre-trained models!

Concatenation and Stacking

```python
x = torch.randn(2, 3)
y = torch.randn(2, 3)
z = torch.randn(2, 3)

# Concatenate along existing dimension
cat0 = torch.cat([x, y, z], dim=0)  # Shape: (6, 3)
cat1 = torch.cat([x, y, z], dim=1)  # Shape: (2, 9)

# Stack (creates new dimension)
stack0 = torch.stack([x, y, z], dim=0)  # Shape: (3, 2, 3)
stack1 = torch.stack([x, y, z], dim=1)  # Shape: (2, 3, 3)

# Split (opposite of cat)
chunks = torch.chunk(cat0, 3, dim=0)  # List of 3 tensors,
    each (2, 3)
split = torch.split(cat0, [2, 2, 2], dim=0)  # Custom split
    sizes
```

1.3.3 GPU Operations and Device Management

Moving computation to GPU is trivial in PyTorch, but has important subtleties.

Basic GPU Usage

```python
# Check if CUDA is available
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Create tensor directly on GPU
x_gpu = torch.randn(1000, 1000, device='cuda')

# Move existing tensor to GPU
x_cpu = torch.randn(1000, 1000)
x_gpu = x_cpu.to('cuda')
# or
x_gpu = x_cpu.cuda()  # Shorthand, but less flexible

# Move back to CPU
x_cpu = x_gpu.to('cpu')
# or
x_cpu = x_gpu.cpu()

# Check device
print(x_gpu.device)  # cuda:0
```

Device-Agnostic Code

Write code that works on both CPU and GPU:

```python
# At the top of your script
device = torch.device('cuda' if torch.cuda.is_available
    () else 'cpu')

# Create tensors
x = torch.randn(100, 100, device=device)

# Or move existing tensors
x = x.to(device)

# For neural network models
model = MyModel()
model.to(device)

# All subsequent operations inherit device
y = model(x)  # y will be on same device as x
```

Multi-GPU Considerations

```python
# Check number of GPUs
num_gpus = torch.cuda.device_count()

# Specify which GPU
x = torch.randn(100, 100, device='cuda:0')  # First GPU
y = torch.randn(100, 100, device='cuda:1')  # Second GPU

# Operations between tensors on different GPUs will fail!
# z = x + y  # ERROR!

# Must move to same device first
y = y.to('cuda:0')
z = x + y  # Now works
```

GPU Memory Management

GPUs have limited memory. Here are strategies for large models:

```python
# Clear CUDA cache
torch.cuda.empty_cache()

# Check memory usage
print(torch.cuda.memory_allocated() / 1e9)  # GB
print(torch.cuda.memory_reserved() / 1e9)   # GB

# Delete tensors explicitly
del large_tensor
torch.cuda.empty_cache()

# Use gradient checkpointing for very deep networks
# (recompute activations instead of storing them)
from torch.utils.checkpoint import checkpoint

# Context manager to avoid tracking gradients (saves
    memory)
with torch.no_grad():
    output = model(input)  # No computational graph
    built
```

Memory optimization strategies:

1. Use mixed precision (`torch.cuda.amp`)
2. Reduce batch size
3. Use gradient accumulation
4. Enable gradient checkpointing
5. Use `.detach()` to remove tensors from computational graph

1.3.4 Interoperability with NumPy

```python
import numpy as np

# NumPy to PyTorch (zero-copy if possible)
np_array = np.array([[1, 2], [3, 4]])
torch_tensor = torch.from_numpy(np_array)

# Modifying one modifies the other (they share memory!)
np_array[0, 0] = 999
print(torch_tensor[0, 0])  # 999

# PyTorch to NumPy (also shares memory if on CPU)
torch_tensor = torch.randn(2, 3)
np_array = torch_tensor.numpy()
```

```

# For GPU tensors, must move to CPU first
gpu_tensor = torch.randn(2, 3, device='cuda')
np_array = gpu_tensor.cpu().numpy()
```

Memory Sharing Gotcha

When converting between NumPy and PyTorch, they often share memory:

```
np_arr = np.array([1, 2, 3])
torch_arr = torch.from_numpy(np_arr)

np_arr[0] = 999
print(torch_arr)  # tensor([999, 2, 3]) - SHARED MEMORY!
```

To avoid this, explicitly copy:

```
torch_arr = torch.tensor(np_arr)  # Creates new copy
# or
torch_arr = torch.from_numpy(np_arr.copy())
```

1.3.5 Counterintuitive Patterns and Gotchas

Broadcasting Surprises

```
# Intending to normalize each sample
X = torch.randn(100, 10)  # 100 samples, 10 features
mean = X.mean(dim=0)        # Shape: (10,) - mean per feature
X_norm = X - mean           # Broadcasts correctly

# But what if you transpose?
X_T = X.t()                 # Shape: (10, 100)
X_T_norm = X_T - mean       # Broadcasts, but probably not
    what you want!
# mean (10,) broadcasts to (10, 100) by copying across dim 1
```

Solution: Always use `keepdim=True` when reducing to preserve dimensionality:

```
mean = X.mean(dim=0, keepdim=True)  # Shape: (1, 10)
X_norm = X - mean                    # Unambiguous
    broadcasting
```

The View Trap

```
x = torch.randn(2, 3)
y = x.t()  # Transpose - now non-contiguous

```

```python
# This fails!
try:
    z = y.view(6)
except RuntimeError as e:
    print(f"Error: {e}")  # view size not compatible

# Must make contiguous first
z = y.contiguous().view(6)  # Works
# or use reshape (does this automatically)
z = y.reshape(6)  # Works
```

When to Use View

Despite the gotcha, `view()` is preferable to `reshape()` when:

1. You *know* the tensor is contiguous
2. You want to catch bugs (reshape silently copies)
3. Performance is critical (view is guaranteed zero-copy)

Use `reshape()` for robust code that handles any input. Use `view()` for performance-critical sections after verifying contiguity.

In-Place Operations and Autograd

```python
# This breaks autograd!
x = torch.randn(3, 4, requires_grad=True)
x.add_(1)  # In-place modification
y = x.sum()
y.backward()  # RuntimeError: one of the variables needed for
    gradient
                # computation has been modified by an inplace
    operation
```

Rule: Never use in-place operations on tensors that require gradients unless you know exactly what you're doing.

1.4 Best Practices for Scientific ML

1.4.1 Choosing Data Types

1.4.2 Memory Management Strategy

1. **Profile first:** Use `torch.cuda.memory_summary()` to identify bottlenecks
2. **Delete intermediate tensors:** Use `del` and `torch.cuda.empty_cache()`
3. **Gradient accumulation:** For large batch sizes, accumulate over multiple forward passes

Table 1.2: Dtype Selection Guide

Use Case	Recommended dtype
Standard deep learning	`float32`
PINNs, high-order derivatives	`float64`
Large models (inference)	`float16`/`bfloat16`
Integer indices, labels	`int64`
Masks, boolean logic	`bool`

4. **Mixed precision:** Use automatic mixed precision (AMP) for 2× memory savings

5. **Data loading:** Use `pin_memory=True` and `num_workers > 0`

1.4.3 Numerical Stability Considerations

Avoiding Numerical Instability

Common issues in scientific ML:

1. Underflow in exponentials:

```python
# Bad: can underflow for large negative x
def softmax_naive(x):
    exp_x = torch.exp(x)
    return exp_x / exp_x.sum(dim=-1, keepdim=True)

# Good: numerically stable
def softmax_stable(x):
    x_max = x.max(dim=-1, keepdim=True)[0]
    exp_x = torch.exp(x - x_max)  # Subtract max for stability
    return exp_x / exp_x.sum(dim=-1, keepdim=True)
```

2. Log of small numbers:

```python
# Bad: log(0) = -inf
loss = -torch.log(probabilities)

# Good: add small epsilon
loss = -torch.log(probabilities + 1e-8)

# Best: use built-in stable versions
loss = F.nll_loss(log_probabilities, targets)
```

3. Gradient clipping for exploding gradients:

```python
# Clip gradients to prevent explosion
torch.nn.utils.clip_grad_norm_(model.parameters(),
    max_norm=1.0)
```

1.5 Practical Exercises

1.5.1 Elementary Exercises

1.1: Tensor Creation and Inspection

Objective: Master basic tensor creation and property inspection.

Tasks:

1. Create a tensor of shape $(4, 5, 6)$ filled with random values from $\mathcal{N}(0, 1)$.
2. Inspect and print: shape, dtype, device, number of elements, whether it

requires gradients.

3. Convert it to `float64`, then to `int32`, observing what happens to values.
4. If you have a GPU, move the tensor to GPU and back to CPU.
5. Verify the tensor is contiguous. Transpose it and check again.

Expected output: Understanding of tensor properties and how operations affect them.

1.2: Broadcasting Mastery

Objective: Understand and predict broadcasting behavior.

Tasks: Given the following tensors:

```
A = torch.randn(3, 1, 5)
B = torch.randn(1, 4, 5)
C = torch.randn(5)
D = torch.randn(3, 4)
```

1. Without running code, predict the output shape of: `A + B`, `A + C`, `B + C`
2. Verify your predictions by running the code.
3. Try `A + D`. Will it work? Why or why not?
4. Create a tensor `E` such that `D + E` broadcasts to shape $(3, 4, 5)$.

Learning goal: Internalize broadcasting rules to avoid silent bugs.

1.3: Indexing and Slicing Operations

Objective: Master advanced indexing techniques.

Tasks: Create a tensor `X` of shape $(10, 8, 6)$ with random values.

1. Extract the first 5 elements along dimension 0.
2. Extract every other element along dimension 1 for all of dimension 0.
3. Use boolean masking to extract all elements greater than 0.5.
4. Use `torch.where()` to create a tensor where elements > 0 are replaced with 1 and elements ≤ 0 with -1.
5. Use advanced indexing to extract elements at indices $[0, 3, 7]$ along dimension 0 and indices $[1, 4]$ along dimension 1.

Expected skills: Comfort with PyTorch's rich indexing API.

1.5.2 Intermediate Exercises

1.4: Implementing Batch Matrix Operations

Objective: Work with batched operations common in deep learning.

Tasks:

1. Create a batch of 32 matrices, each of size $(10, 10)$, with random values.
2. Implement batched matrix multiplication: multiply each matrix by its

transpose.
3. Compute the determinant of each matrix in the batch (use `torch.linalg.det`).
4. For each matrix, compute eigenvalues and eigenvectors (use `torch.linalg.eig`).
5. Identify matrices that are ill-conditioned (condition number > 100). Hint: Use `torch.linalg.cond`.

Learning goal: Understanding batched linear algebra operations crucial for scientific ML.

1.5: Memory-Efficient Operations

Objective: Learn to work with large tensors without running out of memory.
Context: You need to compute pairwise distances between 10,000 points in 128-dimensional space. The naive approach would create a $(10000, 10000)$ matrix, requiring $\sim$400MB.
Tasks:

1. Implement the naive version:

```python
def pairwise_distances_naive(X):
    # X: (N, D)
    # Return: (N, N) matrix of Euclidean distances
    pass

```

2. Implement a memory-efficient version that processes in chunks:

```python
def pairwise_distances_chunked(X, chunk_size=1000):
    # Process chunk_size rows at a time
    # Return: same (N, N) matrix
    pass

```

3. Compare memory usage using `torch.cuda.memory_allocated()` (if GPU available).
4. Benchmark speed: which is faster and why?
5. **Bonus:** Implement using `torch.cdist` and compare.

Learning goal: Strategies for working with large-scale data.

1.6: Custom Reduction Operations

Objective: Implement custom reductions for scientific computing.
Tasks: Implement the following custom reduction operations from scratch (without using built-in PyTorch functions for the core computation):

1. **Geometric mean:** $(\prod_{i=1}^{n} x_i)^{1/n}$
   - Hint: Use log-sum-exp trick for numerical stability
2. **Harmonic mean:** $\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$
3. **Trimmed mean:** Mean after removing top and bottom 10% of values
   - Use `torch.kthvalue` or `torch.quantile`
4. **Weighted standard deviation:** Given weights w_i, compute $\sqrt{\frac{\sum w_i(x_i - \bar{x})^2}{\sum w_i}}$

Each function should:

- Accept a `dim` argument for reduction axis
- Accept a `keepdim` argument
- Handle numerical edge cases gracefully
- Work on GPU tensors

Learning goal: Deep understanding of reduction operations and numerical stability.

1.5.3 Advanced Exercises

1.7: Implementing Einstein Summation

Objective: Master `torch.einsum`, the Swiss Army knife of tensor operations.

Background: Einstein summation notation is a compact way to express tensor operations. PyTorch's `torch.einsum` implements this.

Tasks:

1. **Basics:** Express these operations using `einsum`:
   - Dot product: $\boldsymbol{a} \cdot \boldsymbol{b}$
   - Matrix multiplication: $\boldsymbol{AB}$
   - Batch matrix multiplication: $\boldsymbol{A}_i\boldsymbol{B}_i$ for $i = 1, \ldots, N$
   - Trace: $\mathrm{Tr}(\boldsymbol{A}) = \sum_i A_{ii}$
   - Transpose: $\boldsymbol{A}^\top$
2. **Scientific computing:** Implement these using `einsum`:
   - Bilinear form: $\boldsymbol{x}^\top \boldsymbol{Ay}$
   - Tensor contraction: Sum over indices (i, j) of $T_{ijk} \cdot S_{ijl}$
   - Outer product of three vectors: $T_{ijk} = a_i b_j c_k$
3. **Deep learning:** Implement attention mechanism using einsum:

```
# Q, K, V: (batch, heads, seq_len, d_k)
# Output: (batch, heads, seq_len, d_k)
# Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) V

```

4. **Performance:** Compare einsum vs explicit operations for batched matrix multiplication. Which is faster?

Learning goal: Fluency with einsum for concise, efficient tensor operations.

1.8: Building a Sparse Tensor System

Objective: Work with sparse tensors for large-scale scientific computing.

Context: Many scientific problems involve sparse matrices (e.g., finite element methods, graph neural networks). PyTorch supports sparse tensors.

Tasks:

1. Create a sparse matrix representation of the 2D Laplacian operator on a 100×100 grid:

$$\nabla^2 u_{i,j} = u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}$$

2. Implement sparse matrix-vector multiplication using `torch.sparse`.
3. Solve the Poisson equation $\nabla^2 u = f$ using conjugate gradient method with your sparse Laplacian.
4. **Performance analysis:**
   - Compare memory usage: sparse vs dense representation

- Compare computation time: sparse vs dense matrix-vector product
- At what sparsity level does sparse representation become advantageous?

5. **Advanced:** Implement Incomplete Cholesky preconditioning for faster convergence.

Learning goal: Working with sparse tensors for large-scale scientific problems.

1.5.4 Hints

Hints for Exercise 1.4

- For batched operations, the batch dimension is typically dimension 0
- `torch.bmm` performs batched matrix multiplication
- For transpose in batched setting: `X.transpose(-2, -1)`
- Condition number: ratio of largest to smallest singular value

Hints for Exercise 1.5

- Euclidean distance: $\|\boldsymbol{x}-\boldsymbol{y}\|_2 = \sqrt{\sum_i (x_i - y_i)^2}$
- Vectorized form: $\|\boldsymbol{x}-\boldsymbol{y}\|_2^2 = \|\boldsymbol{x}\|_2^2 + \|\boldsymbol{y}\|_2^2 - 2\boldsymbol{x}\cdot\boldsymbol{y}$
- Process in chunks: iterate over rows, compute distances to all other points for each chunk
- Use `torch.no_grad()` context to avoid building computational graph

Hints for Exercise 1.7

- Einsum syntax: `torch.einsum('ij,jk->ik', A, B)` for matrix multiplication
- Repeated indices are summed over (Einstein convention)
- For attention, you'll need: `'bhqd,bhkd->bhqk'` for $\boldsymbol{Q}\boldsymbol{K}^\top$
- Use `timeit` module for accurate benchmarking

Hints for Exercise 1.8

- Sparse tensor format: `torch.sparse_coo_tensor(indices, values, size)`
- For 2D Laplacian, each row has at most 5 non-zero entries
- Conjugate gradient: iterative method, needs only matrix-vector products
- Preconditioning: $M^{-1}\boldsymbol{A}\boldsymbol{x} = M^{-1}\boldsymbol{b}$ where $M \approx \boldsymbol{A}$ is easier to invert

1.6 Summary and Looking Forward

In this chapter, we built a comprehensive understanding of PyTorch tensors from first principles. We covered:

- **Mathematical foundations:** Tensors as multi-dimensional arrays, broadcasting rules, memory layout
- **PyTorch API:** Creation, manipulation, indexing, and shape operations
- **Device management:** CPU vs GPU, memory optimization strategies
- **Numerical considerations:** Dtype selection, stability tricks, performance optimization
- **Best practices:** Idioms, common pitfalls, debugging strategies

Key takeaways:

1. Tensors are the fundamental data structure—master them, and everything else follows
2. Broadcasting is powerful but dangerous—always verify shapes explicitly
3. Memory management matters—profile before optimizing
4. Numerical stability is critical in scientific computing—use stable implementations
5. Modern PyTorch encourages functional style—avoid in-place ops on gradient tensors

Before Moving Forward

If you feel overwhelmed, that's normal. Tensor operations are the vocabulary of deep learning. Just as you can't write poetry without knowing words, you can't build neural networks without fluency in tensors.

Recommended practice:

- Spend a week working exclusively with tensors—no neural networks yet
- Implement familiar algorithms (matrix factorization, PCA, k-means) using only PyTorch tensors
- Debug every dimension mismatch error until you can predict shapes in your sleep
- Read PyTorch source code: `https://github.com/pytorch/pytorch`

The investment pays dividends. When you encounter bugs in Chapter 10 (Transformers) involving attention masks with shape $(B, 1, S, S)$ broadcasting with logits of shape (B, H, S, S), you'll debug it in seconds instead of hours.

In the next chapter, we tackle the second pillar of PyTorch: **automatic differentiation**.

We will see how PyTorch builds computational graphs, computes gradients efficiently, and enables the gradient-based optimization that powers all modern deep learning.

Onward!