MCDA 5511

Current Practices in Computing and Data Science

Introduction to PyTorch

Part 1: Tensors, Operations & Autograd

Somto Muotoe

January 2026

Tutorial Overview

This is a **two-part series** on PyTorch fundamentals.

Part 1: Foundations (This Week)

- What is PyTorch?
- Tensors: creation, attributes, types
- Tensor operations & broadcasting
- Reshaping & memory
- Indexing & slicing
- PyTorch vs NumPy
- Automatic differentiation (Autograd)

Part 2: Neural Networks (Next Week)

- The `nn.Module` class
- Building network architectures
- Loss functions & optimizers
- The training loop
- Data handling with DataLoader
- Model evaluation

Setup

Clone the repository and install dependencies:

```
git clone https://github.com/smuotoe/mcda-pytorch-tutorial.git
cd mcda-pytorch-tutorial
uv sync
```

Prerequisites

- Python 3.10+
- Basic Python programming
- Familiarity with NumPy is helpful but not required

Resources

- PyTorch Documentation
- PyTorch 60-Minute Blitz

What is PyTorch?

PyTorch: A Deep Learning Framework

What is it?

- Open-source deep learning framework
- Developed by Meta AI (formerly Facebook)
- Released in 2016, now under Linux Foundation

Key Features

- **Pythonic API** - feels like native Python
- **GPU acceleration** - seamless CUDA support
- **Automatic differentiation** - gradients computed automatically

The PyTorch Ecosystem

```
PyTorch Core
    |
    +-- torchvision (images)
    +-- torchaudio (audio)
    +-- torchtext (NLP)
    +-- torch.distributed (multi-GPU)
    +-- TorchServe (deployment)
```

Industry Adoption

- Used by Tesla, OpenAI, Microsoft, Hugging Face
- Dominant in ML research and growing in production

Sources: Ecosystem, Case Studies

Why PyTorch?

PyTorch vs TensorFlow

Aspect	PyTorch	TensorFlow
Learning curve	Easier	Steeper
Research adoption	Dominant	Less common
API style	Pythonic, intuitive	More verbose

Both frameworks are capable of the same tasks. PyTorch is often preferred for learning and research due to its simpler API.

Source: PyTorch vs TensorFlow Comparison (OpenCV)

What Makes PyTorch Beginner-Friendly

Intuitive API

- Code reads like standard Python
- Operations behave as you'd expect

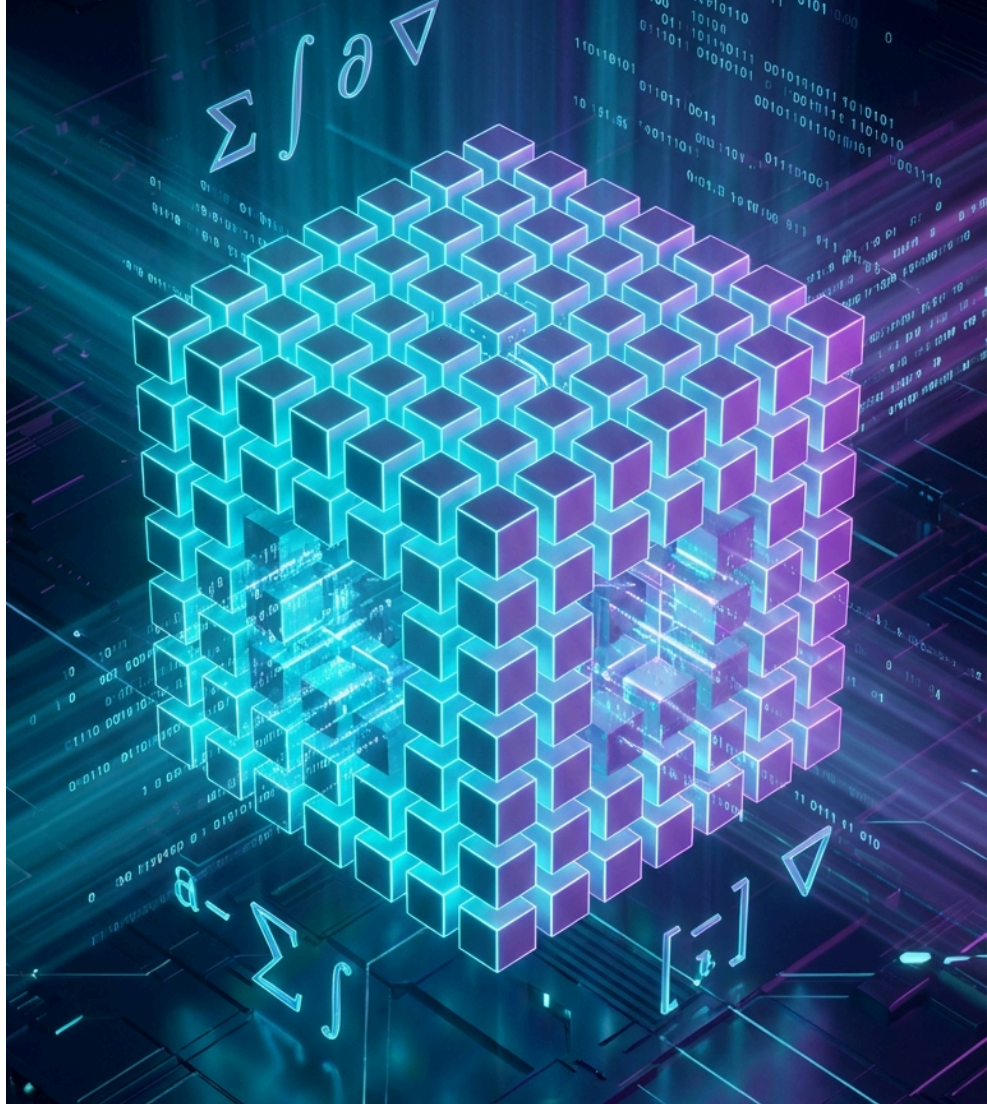
Easy to Experiment

- Interactive development in notebooks
- Quick iteration on model changes

Great Documentation

- Extensive official tutorials
- Active community support

Tensors: The Core Data Structure



What is a Tensor?

A **tensor** is an n-dimensional array - the fundamental data structure in PyTorch.

Scalar

0-dimensional

```
torch.tensor(5)
```

Shape: `()`

Vector

1-dimensional

```
torch.tensor([1, 2, 3])
```

Shape: `(3,)`

Matrix

2-dimensional

```
torch.tensor([
    [1, 2],
    [3, 4]
])
```

Shape: `(2, 2)`

3D Tensor

3-dimensional

```
torch.rand(2, 3, 4)
```

Shape: `(2, 3, 4)`

Real-world examples: Images are 3D tensors (height x width x channels), batches of images are 4D tensors.

Creating Tensors

From Data

```
import torch

# From a Python list
t1 = torch.tensor([1, 2, 3])

# From nested lists (matrix)
t2 = torch.tensor([[1, 2], [3, 4]])

# From NumPy array
import numpy as np
arr = np.array([1, 2, 3])
t3 = torch.from_numpy(arr)
```

Factory Functions

```
# Zeros and ones
zeros = torch.zeros(3, 4)      # 3×4 of zeros
ones = torch.ones(2, 3)        # 2×3 of ones

# Random values
rand = torch.rand(2, 2)        # uniform [0, 1)
randn = torch.randn(2, 2)      # normal dist

# Sequences
seq = torch.arange(0, 10, 2)   # [0, 2, 4, 6, 8]
lin = torch.linspace(0, 1, 5)  # 5 points 0 to 1

# Like another tensor
like = torch.zeros_like(rand)  # same shape
```

Tensor Attributes

Every tensor has three key attributes:

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

print(t.shape)   # torch.Size([3, 4]) - dimensions
print(t.dtype)   # torch.float32 - data type
print(t.device)  # cpu - where it lives
```

Shape

The size of each dimension.

```
t.shape      # torch.Size([3, 4])
t.size()     # same thing
t.ndim       # 2 (number of dims)
t.numel()    # 12 (total elements
```

Data Type (dtype)

Numeric precision.

```
torch.float32  # default for floa
torch.float64  # double precision
torch.int64    # default for ints
torch.int32    # 32-bit integer
torch.bool     # boolean
```

Device

CPU or GPU location.

```
t.device           # cpu
t.to('cuda')       # move to GPU
t.to('cpu')        # move to CPU
t.cuda()           # shorthand
t.cpu()            # shorthand
```

Data Types and Casting

Data type (dtype): How values are stored in memory, trading precision for speed/memory. **Casting:** Converting a tensor from one dtype to another.

Common dtypes

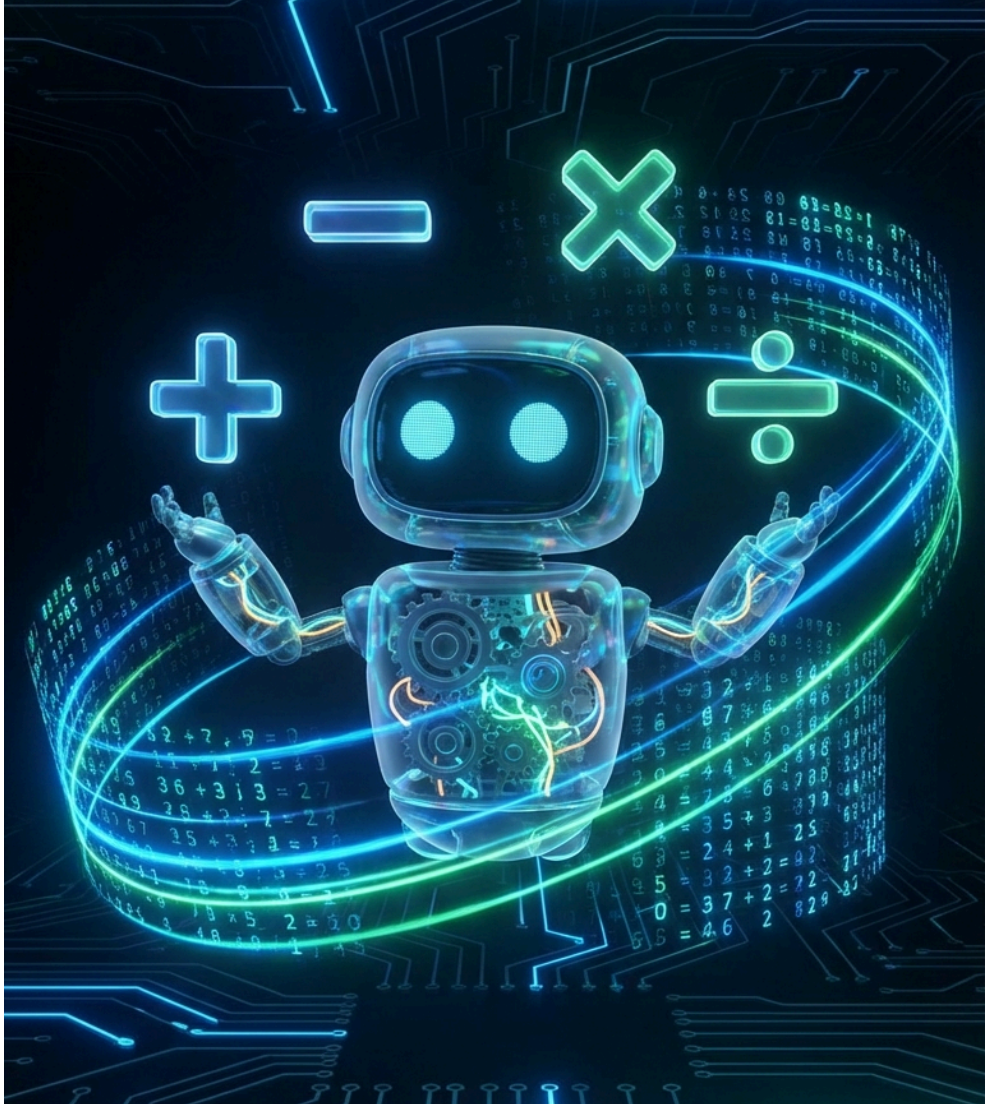
Type	Size	Use Case
`float32`	4 bytes	Default for neural nets
`float16`	2 bytes	Mixed precision training
`bfloat16`	2 bytes	Same range as float32, less precision
`float64`	8 bytes	High precision math
`int64`	8 bytes	Indices, labels
`int32`	4 bytes	Smaller indices
`bool`	1 byte	Masks

Casting Between Types

```
t = torch.tensor([1, 2, 3])

# Using .to()
t.to(torch.float32)
t.to(torch.float16)
t.to(torch.bfloat16)

# Shorthand methods
t.float()   # float32
t.half()    # float16
t.double()  # float64
t.int()     # int32
t.long()    # int64
```

Warning: Casting to int truncates, doesn't round: `tensor([1.7]).int()` gives `[1]`

Tensor Operations



Element-wise Operations

Operations applied to each element independently.

Arithmetic

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# Using operators
a + b        # tensor([5, 7, 9])
a - b        # tensor([-3, -3, -3])
a * b        # tensor([4, 10, 18])
a / b        # tensor([0.25, 0.4, 0.5])
a ** 2       # tensor([1, 4, 9])

# Using functions (equivalent)
torch.add(a, b)
torch.sub(a, b)
torch.mul(a, b)
torch.div(a, b)
```

With Scalars

```
a = torch.tensor([1, 2, 3])

a + 10       # tensor([11, 12, 13])
a * 2        # tensor([2, 4, 6])
a / 2        # tensor([0.5, 1.0, 1.5])
```

Mathematical Functions

```
t = torch.tensor([1.0, 4.0, 9.0])

torch.sqrt(t)    # [1, 2, 3]
torch.exp(t)     # e^x for each x
torch.log(t)     # natural log
torch.abs(t)     # absolute value
torch.sin(t)     # trigonometric
```

Reduction Operations

Operations that reduce dimensions by aggregating values.

Basic Reductions

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

t.sum()      # 21 (all elements)
t.mean()     # 3.5
t.max()      # 6
t.min()      # 1
t.prod()     # 720 (product)
```

Along an Axis

```
# Sum along rows (axis 0)
t.sum(dim=0)     # tensor([5, 7, 9])

# Sum along columns (axis 1)
t.sum(dim=1)     # tensor([6, 15])

# Keep dimensions
t.sum(dim=1, keepdim=True)  # shape (2, 1)
```

Finding Indices

```
t = torch.tensor([3, 1, 4, 1, 5])

t.argmax()       # 4 (index of max)
t.argmin()       # 1 (index of min)

# Along axis
m = torch.tensor([[1, 5, 2],
                  [4, 3, 6]])
m.argmax(dim=1)  # [1, 2] (per row)
```

Statistical

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

t.std()      # standard deviation
t.var()      # variance
t.median()   # median value
```

Matrix Operations

Matrix Multiplication

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

# Three equivalent ways
a @ b                  # preferred
torch.matmul(a, b)     # explicit
torch.mm(a, b)         # 2D only

# Result:
# [[1*5+2*7, 1*6+2*8],
#  [3*5+4*7, 3*6+4*8]]
# = [[19, 22],
#    [43, 50]]
```

Note: `*` is element-wise, `@` is matrix multiplication.

Other Matrix Operations

```
m = torch.tensor([[1, 2],
                  [3, 4]], dtype=torch.float32)

# Transpose
m.T              # or m.t() or m.transpose(0, 1)

# Inverse (square matrices)
torch.inverse(m)

# Determinant
torch.det(m)     # -2.0

# Matrix-vector multiplication
v = torch.tensor([1, 2])
m @ v            # tensor([5, 11])
```

Dot Product (vectors)

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
torch.dot(a, b)  # 32 (1*4 + 2*5 + 3*6)
```

Comparison Operations

Element-wise Comparisons

```
a = torch.tensor([1, 2, 3, 4])
b = torch.tensor([2, 2, 2, 2])

a > b       # tensor([False, False, True, True])
a ⩾ b       # tensor([False, True, True, True])
a < b       # tensor([True, False, False, False])
a == b      # tensor([False, True, False, False])
a ≠ b       # tensor([True, False, True, True])
```

With Scalars

```
a > 2       # tensor([False, False, True, True])
```

Logical Operations

```
x = torch.tensor([True, True, False])
y = torch.tensor([True, False, False])

x & y       # tensor([True, False, False])
x | y       # tensor([True, True, False])
~x          # tensor([False, False, True])
```

torch.where

```
a = torch.tensor([1, 2, 3, 4])

# Select based on condition
torch.where(a > 2, a, torch.zeros_like(a))
# tensor([0, 0, 3, 4])

# Useful for conditional assignment
torch.where(a > 2, a * 10, a)
# tensor([1, 2, 30, 40])
```

Broadcasting

What is Broadcasting?

Broadcasting allows operations on tensors of **different shapes** by automatically expanding dimensions.

```
# Without broadcasting - shapes must match exactly
a = torch.tensor([1, 2, 3])
b = torch.tensor([10, 10, 10])
a + b  # tensor([11, 12, 13])

# With broadcasting - scalar expands to match
a = torch.tensor([1, 2, 3])
a + 10  # tensor([11, 12, 13])  # 10 broadcasts to [10, 10, 10]
```

Why Broadcasting Matters

- **Memory efficient**: No need to create expanded copies
- **Concise code**: Express operations naturally
- **Performance**: Optimized at C++ level

Broadcasting Rules

Two tensors are **broadcastable** if, comparing dimensions **right-to-left**:

1. The dimensions are equal, OR
2. One of them is 1, OR
3. One of them doesn't exist (missing dimensions are treated as 1)

For shape `(2, 3, 4)`, we compare: 4 first, then 3, then 2. The rightmost dimension is called the "trailing" dimension.

Valid Broadcasting

```
# (3,) + (1,) → (3,)
torch.tensor([1, 2, 3]) + torch.tensor([10])
# Result: [11, 12, 13]

# (2, 3) + (3,) → (2, 3)
a = torch.ones(2, 3)
b = torch.tensor([1, 2, 3])
a + b  # [[2, 3, 4], [2, 3, 4]]

# (2, 1) + (1, 3) → (2, 3)
a = torch.tensor([[1], [2]])      # (2, 1)
b = torch.tensor([[10, 20, 30]])  # (1, 3)
a + b  # [[11, 21, 31], [12, 22, 32]]
```

Invalid Broadcasting

```
# (3,) + (2,) → ERROR
a = torch.tensor([1, 2, 3])
b = torch.tensor([1, 2])
a + b  # RuntimeError!
# Neither is 1, and they're not equal

# (2, 3) + (2,) → ERROR
a = torch.ones(2, 3)
b = torch.tensor([1, 2])
a + b  # RuntimeError!
# Trailing dims: 3 vs 2 - incompatible
```

Tip: Check shapes with `a.shape` and `b.shape` when debugging.

Broadcasting Patterns

Row/Column Operations

```
# Subtract row mean from each row
data = torch.tensor([[1, 2, 3],
                     [4, 5, 6]], dtype=torch.float32)

row_mean = data.mean(dim=1, keepdim=True)  # (2, 1)
centered = data - row_mean
# [[-1, 0, 1],
#  [-1, 0, 1]]

# Divide each column by its max
col_max = data.max(dim=0).values  # (3,)
normalized = data / col_max
```

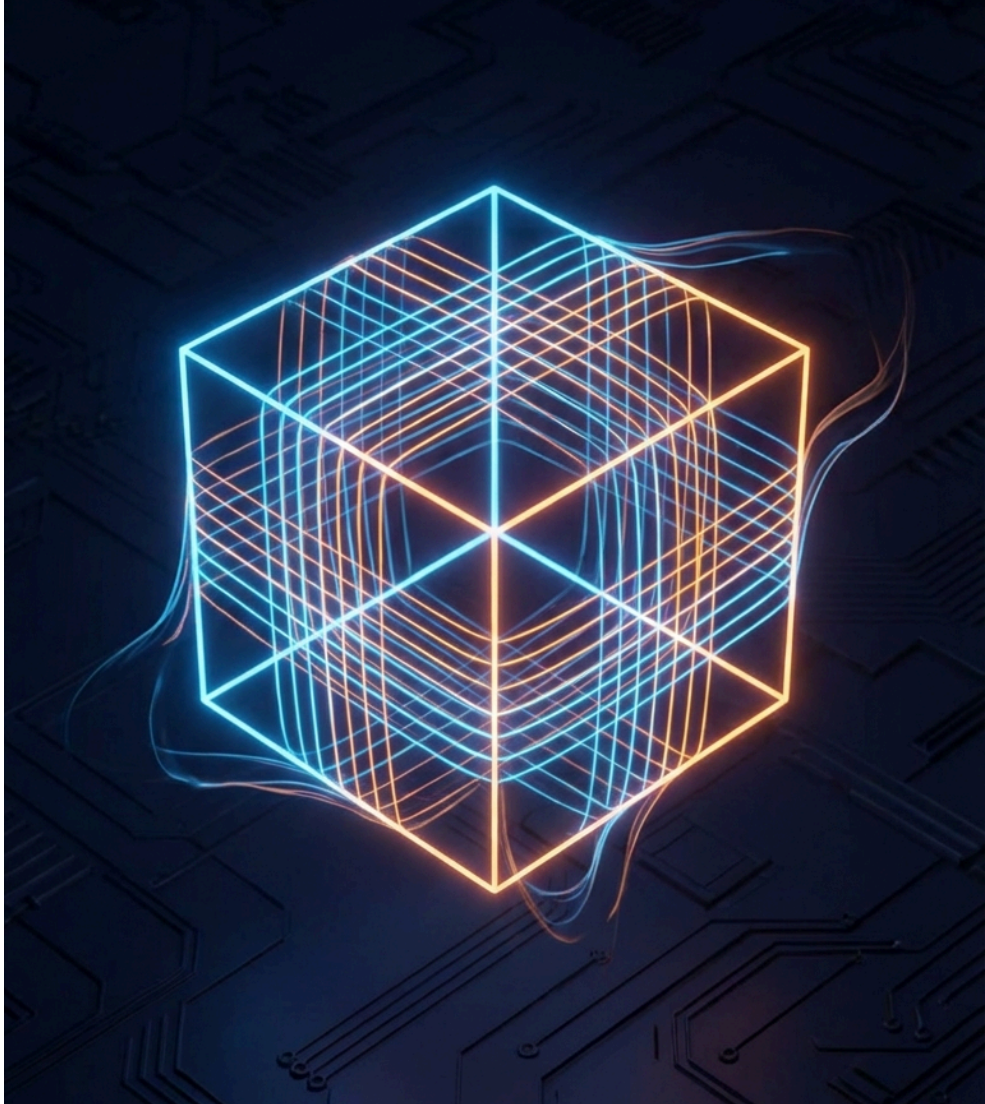
Outer Product via Broadcasting

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([10, 20])

# Reshape to enable broadcasting
# (3, 1) * (1, 2) → (3, 2)
outer = a.unsqueeze(1) * b.unsqueeze(0)
# [[10, 20],
#  [20, 40],
#  [30, 60]]
```

Common Pitfall

```
# Accidentally broadcasting when you didn't mean to
a = torch.rand(100, 1)
b = torch.rand(1, 100)
c = a + b  # Shape: (100, 100) - 10,000 elements!
```

Reshaping and Memory



Reshaping Tensors

reshape and view

```
t = torch.arange(12)  # [0, 1, 2, ..., 11]

# Reshape to 3×4
t.reshape(3, 4)
# [[ 0,  1,  2,  3],
#  [ 4,  5,  6,  7],
#  [ 8,  9, 10, 11]]

# Use -1 to infer dimension
t.reshape(3, -1)   # same as (3, 4)
t.reshape(-1, 6)   # shape (2, 6)

# view is similar but requires contiguous memory
t.view(3, 4)
```

reshape vs view

- `view` : faster, but requires contiguous tensor
- `reshape` : always works, may copy data

squeeze and unsqueeze

```
# Remove dimensions of size 1
t = torch.zeros(1, 3, 1, 4)
t.squeeze().shape        # (3, 4)
t.squeeze(0).shape       # (3, 1, 4)
t.squeeze(2).shape       # (1, 3, 4)

# Add dimension of size 1
t = torch.zeros(3, 4)
t.unsqueeze(0).shape     # (1, 3, 4)
t.unsqueeze(1).shape     # (3, 1, 4)
t.unsqueeze(-1).shape    # (3, 4, 1)
```

flatten

```
t = torch.zeros(2, 3, 4)
t.flatten().shape        # (24,)
t.flatten(1).shape       # (2, 12) - keep first dim
```

Transpose and Permute

Transpose (2D matrices)

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

m.T              # shape (3, 2)
m.t()            # same
m.transpose(0, 1)  # same

# Result:
# [[1, 4],
#  [2, 5],
#  [3, 6]]
```

transpose() on Higher Dimensions

```
# transpose() swaps exactly two dimensions
t = torch.zeros(2, 3, 4, 5)
t.transpose(1, 3).shape  # (2, 5, 4, 3)
# Dims 1 and 3 swapped: 3⟷5

# For 2D, these are equivalent:
m.T
m.transpose(0, 1)
```

Permute (reorder all dimensions)

```
t = torch.zeros(2, 3, 4)  # (batch, height, width)

# Specify new order of ALL dimensions
t.permute(0, 2, 1).shape  # (2, 4, 3)
t.permute(2, 1, 0).shape  # (4, 3, 2)
```

Common Use: Channel Ordering

```
# PyTorch default: channels first (C, H, W)
# Some libraries expect: channels last (H, W, C)

img = torch.zeros(3, 224, 224)  # (C, H, W)
img.permute(1, 2, 0).shape      # (H, W, C)

# Batch of images
batch = torch.zeros(32, 3, 224, 224)  # (N, C, H, W)
batch.permute(0, 2, 3, 1).shape       # (N, H, W, C)
```

Memory: Views vs Copies

Understanding when data is shared is crucial for performance and avoiding bugs.

Views (Shared Memory)

```
a = torch.tensor([1, 2, 3, 4])  # shape (4,)
b = a.view(2, 2)  # b is a VIEW of a, reshaped to (2, 2)
# b = [[1, 2],
#      [3, 4]]

b[0, 0] = 99
print(a)  # tensor([99, 2, 3, 4]) - a changed!

# These create views:
# view(), reshape(), transpose(), permute()
# squeeze(), unsqueeze(), slicing, expand()
```

Copies (Independent Memory)

```
a = torch.tensor([1, 2, 3, 4])
b = a.clone()  # b is a COPY

b[0] = 99
print(a)  # tensor([1, 2, 3, 4]) - unchanged

# These create copies:
# - clone()
# - contiguous() (when needed)
# - reshape() (sometimes, when non-contiguous)
# - to() when changing device/dtype
```

Check if Shared

```
a = torch.tensor([1, 2, 3, 4])
b = a.view(2, 2)
print(a.data_ptr() == b.data_ptr())  # True
```

Contiguous Memory

What is Contiguous?

Tensor elements stored sequentially in memory, matching logical order.

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

# Memory layout: [1, 2, 3, 4, 5, 6]
a.is_contiguous()  # True

b = a.T  # Transpose
# Logical: [[1, 4], [2, 5], [3, 6]]
# Memory still: [1, 2, 3, 4, 5, 6]
b.is_contiguous()  # False
```

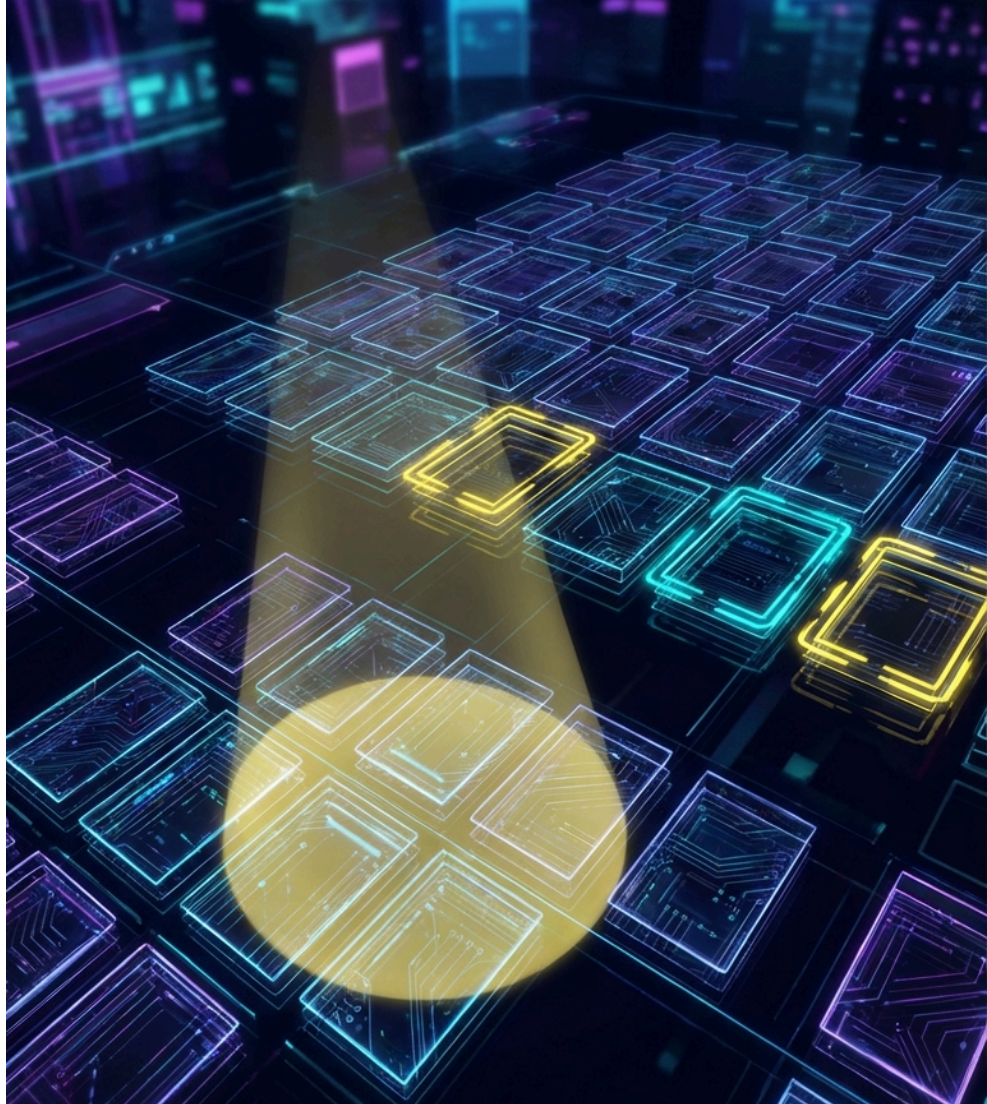
Why Does it Matter?

```
b = a.T

# view() requires contiguous
b.view(6)  # RuntimeError!

# Two solutions:
b.reshape(6)              # works, may copy
b.contiguous().view(6)    # explicit copy
```

Operations that Break Contiguity

```
# These return non-contiguous views:
t.T
t.transpose(0, 1)
t.permute( ... )
t[::2]  # stride slicing
```

Indexing and Slicing



Basic Indexing

PyTorch indexing works like NumPy - zero-indexed, supports negative indices.

1D Indexing

```
t = torch.tensor([10, 20, 30, 40, 50])

t[0]      # 10 (first element)
t[-1]     # 50 (last element)
t[2]      # 30 (third element)
```

2D Indexing

```
m = torch.tensor([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

m[0, 0]    # 1 (top-left)
m[1, 2]    # 6 (row 1, col 2)
m[-1, -1]  # 9 (bottom-right)

# Get entire row/column
m[0]       # tensor([1, 2, 3])
m[:, 0]    # tensor([1, 4, 7])
```

Slicing Syntax

```
# [start:stop:step] - stop is exclusive

t = torch.arange(10)  # [0, 1, 2, ..., 9]

t[2:5]     # [2, 3, 4]
t[:3]      # [0, 1, 2]
t[7:]      # [7, 8, 9]
t[::2]     # [0, 2, 4, 6, 8]
t[::-1]    # [9, 8, 7, ..., 0] (reversed)
```

2D Slicing

```
m = torch.arange(12).reshape(3, 4)
# [[ 0,  1,  2,  3],
#  [ 4,  5,  6,  7],
#  [ 8,  9, 10, 11]]

m[:2, :2]    # top-left 2×2
m[1:, 2:]    # bottom-right 2×2
m[:, ::2]    # every other column
```

Boolean Indexing (Masking)

Select elements based on a condition.

Creating Masks

```
t = torch.tensor([1, 5, 3, 8, 2, 9])

# Condition creates boolean tensor
mask = t > 4
print(mask)  # [False, True, False, True, False, True]

# Use mask to select elements
t[mask]      # tensor([5, 8, 9])

# Or directly
t[t > 4]     # tensor([5, 8, 9])
```

Multiple Conditions

```
# Combine with & (and), | (or)
t[(t > 2) & (t < 8)]  # [5, 3]
t[(t < 3) | (t > 7)]  # [1, 8, 2, 9]
```

2D Masking

```
m = torch.tensor([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# Mask flattens the result
m[m > 5]     # tensor([6, 7, 8, 9])

# Mask specific rows/columns
row_mask = torch.tensor([True, False, True])
m[row_mask]  # rows 0 and 2
```

Assigning with Masks

```
t = torch.tensor([1, 5, 3, 8, 2])
t[t > 4] = 0
print(t)     # tensor([1, 0, 3, 0, 2])
```

Advanced Indexing

Index with List/Tensor

```
t = torch.tensor([10, 20, 30, 40, 50])
indices = torch.tensor([0, 2, 4])

t[indices]   # tensor([10, 30, 50])

# With lists
t[[0, 2, 4]] # same result
```

2D Fancy Indexing

```
m = torch.tensor([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# Select specific elements
rows = torch.tensor([0, 1, 2])
cols = torch.tensor([2, 1, 0])
m[rows, cols]  # tensor([3, 5, 7])

# Select specific rows
m[[0, 2]]      # rows 0 and 2
```

torch.where for Conditional Selection

```
a = torch.tensor([1, 2, 3, 4, 5])
b = torch.tensor([10, 20, 30, 40, 50])

# Choose from a or b based on condition
torch.where(a > 3, a, b)
# tensor([10, 20, 30, 4, 5])

# Find indices where condition is true
torch.where(a > 3)
# (tensor([3, 4]),)  # indices 3 and 4
```

gather (for specific selections)

```
t = torch.tensor([[1, 2], [3, 4]])
idx = torch.tensor([[0, 0], [1, 0]])

torch.gather(t, dim=1, index=idx)
# [[1, 1], [4, 3]]  # per-row column selection
```

PyTorch vs NumPy

Similarities

PyTorch's tensor API is heavily inspired by NumPy.

Creation

```
# NumPy
np.array([1, 2, 3])
np.zeros((3, 4))
np.ones((2, 3))
np.arange(10)
np.linspace(0, 1, 5)

# PyTorch
torch.tensor([1, 2, 3])
torch.zeros(3, 4)
torch.ones(2, 3)
torch.arange(10)
torch.linspace(0, 1, 5)
```

Operations

```
# NumPy
a + b
a * b
a @ b
np.sum(a)
np.mean(a)
a.reshape(3, 4)

# PyTorch
a + b
a * b
a @ b
torch.sum(a)  # or a.sum()
torch.mean(a) # or a.mean()
a.reshape(3, 4)
```

Most NumPy code translates directly to PyTorch - just replace `np` with `torch` and `array` with `tensor`.

Key Differences

GPU Support

```
# NumPy - CPU only
a = np.array([1, 2, 3])

# PyTorch - CPU or GPU
a = torch.tensor([1, 2, 3])
a_gpu = a.to('cuda')  # move to GPU
a_gpu = a.cuda()      # shorthand
```

Automatic Differentiation

```
# NumPy - no gradients
a = np.array([1.0, 2.0])
# Can't compute gradients

# PyTorch - gradients tracked
a = torch.tensor([1.0, 2.0], requires_grad=True)
b = (a ** 2).sum()
b.backward()
print(a.grad)  # tensor([2., 4.])
```

In-place Operations

```
# PyTorch has explicit in-place ops (trailing _)
a = torch.tensor([1, 2, 3])
a.add_(10)     # modifies a in-place
a.mul_(2)      # modifies a in-place
a.zero_()      # sets all to zero

# Be careful with autograd!
a = torch.tensor([1.0], requires_grad=True)
a.add_(1)      # RuntimeError if gradient needed
```

Naming Conventions

```
# NumPy          # PyTorch
np.ndarray       torch.Tensor
a.ndim           a.dim()
np.concatenate   torch.cat
np.stack         torch.stack
axis=0           dim=0
```

Converting Between NumPy and PyTorch

NumPy to PyTorch

```
import numpy as np
import torch

arr = np.array([1, 2, 3])

# Method 1: from_numpy (shares memory!)
t1 = torch.from_numpy(arr)

# Method 2: torch.tensor (copies data)
t2 = torch.tensor(arr)

# Shared memory means changes propagate
arr[0] = 99
print(t1)  # tensor([99, 2, 3]) - changed!
print(t2)  # tensor([1, 2, 3]) - unchanged
```

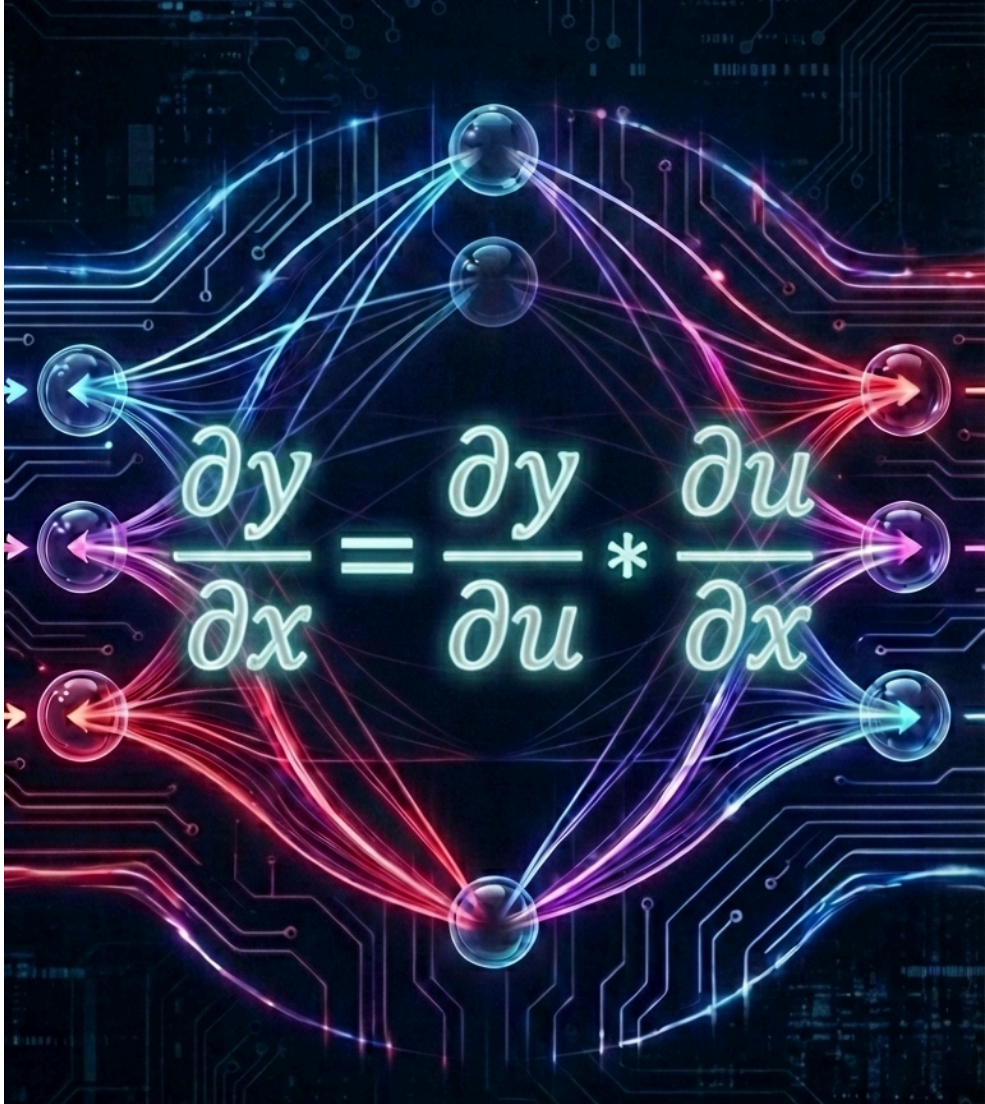
PyTorch to NumPy

```
t = torch.tensor([1, 2, 3])

# Method 1: .numpy() (shares memory on CPU)
arr1 = t.numpy()

# If tensor is on GPU, must move to CPU first
t_gpu = t.cuda()
arr2 = t_gpu.cpu().numpy()

# If tensor requires grad, must detach first
t_grad = torch.tensor([1.0], requires_grad=True)
arr3 = t_grad.detach().numpy()
```

Common Pattern

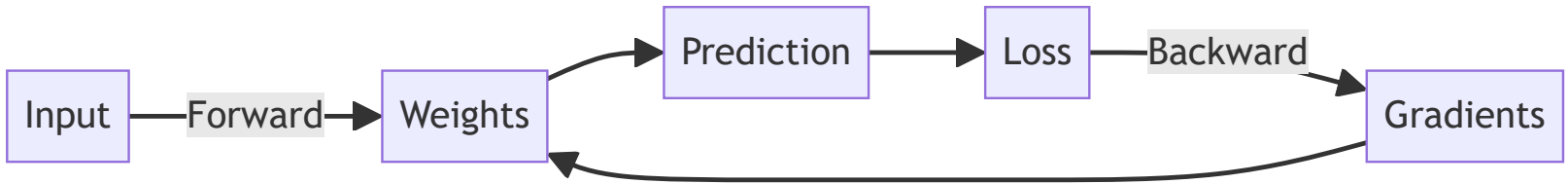
```
# Safe conversion (always works)
arr = tensor.detach().cpu().numpy()
```

Automatic Differentiation (Autograd)



Why Do We Need Gradients?

Neural networks learn by adjusting weights to minimize a loss function.



The Learning Process

1. Make a prediction (forward pass)
2. Compute how wrong it is (loss)
3. Figure out how to adjust weights (gradients)
4. Update weights (optimization)
5. Repeat

Gradients Tell Us

- **Direction**: Which way to adjust each weight
- **Magnitude**: How much to adjust

Without automatic differentiation, you'd compute gradients by hand - impractical for millions of parameters.

requires_grad: Tracking Computations

Enabling Gradient Tracking

```
# By default, no tracking
a = torch.tensor([1.0, 2.0, 3.0])
print(a.requires_grad)  # False

# Enable tracking at creation
b = torch.tensor([1.0, 2.0], requires_grad=True)
print(b.requires_grad)  # True

# Enable on existing tensor
a.requires_grad_(True)  # in-place
# or
a = a.requires_grad_(True)
```

What Gets Tracked?

```
x = torch.tensor([2.0], requires_grad=True)
y = x ** 2      # y = 4.0
z = y * 3       # z = 12.0

# PyTorch builds a computation graph
print(y.grad_fn)  # <PowBackward0>
print(z.grad_fn)  # <MulBackward0>

# z "remembers" how it was computed
# This graph enables automatic differentiation
```

Only for Floats

```
# Integer tensors can't track gradients
t = torch.tensor([1, 2], requires_grad=True)
# RuntimeError: only floating point tensors
```

Computing Gradients with backward()

Simple Example

```
x = torch.tensor([2.0], requires_grad=True)

# Forward pass: y = x^2
y = x ** 2

# Backward pass: compute dy/dx
y.backward()

# Access gradient
print(x.grad)  # tensor([4.0])
# dy/dx = 2x, at x=2: 2*2 = 4
```

With Multiple Variables

```
x = torch.tensor([2.0], requires_grad=True)
w = torch.tensor([3.0], requires_grad=True)

y = x * w  # y = 6.0
y.backward()

print(x.grad)  # tensor([3.0]) - dy/dx = w = 3
print(w.grad)  # tensor([2.0]) - dy/dw = x = 2
```

Chain Rule in Action

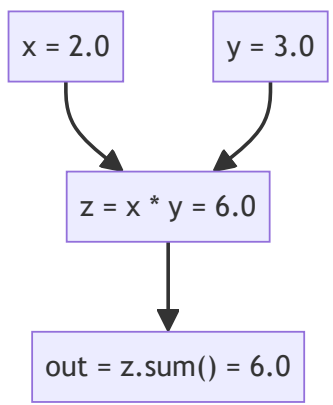
```
x = torch.tensor([2.0], requires_grad=True)

# y = (x^2 + 1)^3
y = (x ** 2 + 1) ** 3

y.backward()
print(x.grad)  # tensor([300.0])

# Manual calculation:
# Let u = x^2 + 1
# y = u^3
# dy/dx = dy/du * du/dx
#       = 3u^2 * 2x
#       = 3(x^2+1)^2 * 2x
#       = 3(5)^2 * 4 = 300
```

The Computation Graph

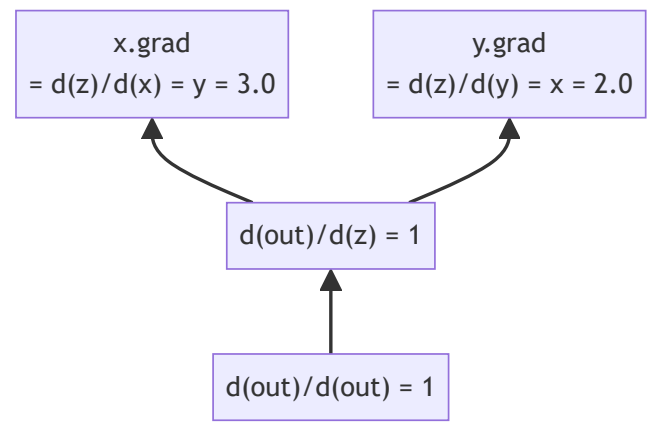
PyTorch builds a **dynamic computation graph** as operations execute.

```
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)

z = x * y        # z depends on x and y
out = z.sum()    # scalar output

out.backward()   # traverse graph backwards
```

Forward Pass (values flow down)



Backward Pass (gradients flow up)



Gradient Accumulation and zero_grad()

Gradients **accumulate** by default - they add up across multiple `backward()` calls.

The Accumulation "Problem"

```
x = torch.tensor([2.0], requires_grad=True)

# First backward
y1 = x ** 2
y1.backward()
print(x.grad)  # tensor([4.0])

# Second backward (gradients ADD)
y2 = x ** 3
y2.backward()
print(x.grad)  # tensor([16.0]) - not 12!
# 4.0 (from y1) + 12.0 (from y2) = 16.0
```

The Solution: zero_grad()

```
x = torch.tensor([2.0], requires_grad=True)

# First backward
y1 = x ** 2
y1.backward()
print(x.grad)  # tensor([4.0])

# Clear gradients before next backward
x.grad = None  # or x.grad.zero_()

# Second backward (fresh start)
y2 = x ** 3
y2.backward()
print(x.grad)  # tensor([12.0]) - correct!
```

Why accumulate by default? It simulates larger batch sizes when GPU memory is limited. Instead of processing 64 samples at once, process 4 mini-batches of 16 samples, accumulating gradients, then update weights once. The accumulated gradient approximates what you'd get from the full batch.

Disabling Gradient Tracking

Sometimes you don't want gradients (inference, evaluation, preprocessing).

torch.no_grad() Context

```
x = torch.tensor([2.0], requires_grad=True)

# Inside no_grad, operations don't track
with torch.no_grad():
    y = x ** 2
    print(y.requires_grad)  # False

# Back to tracking outside
z = x ** 2
print(z.requires_grad)  # True
```

Why Use no_grad()?

- **Faster**: No graph construction overhead
- **Less memory**: No saved tensors for backward
- **Required for**: Evaluation, inference, weight updates

detach()

```
x = torch.tensor([2.0], requires_grad=True)
y = x ** 2

# Detach from graph
y_detached = y.detach()
print(y_detached.requires_grad)  # False

# y_detached shares data but no gradients
```

torch.inference_mode() (PyTorch 1.9+)

```
# Even faster than no_grad for pure inference
with torch.inference_mode():
    y = model(x)
# Can't even enable requires_grad inside
```

Autograd: Putting It Together

A complete example showing the gradient computation workflow:

```
# Simulate learning a simple function: y = 2x + 1
# We'll learn the weight (w) and bias (b)

w = torch.tensor([0.0], requires_grad=True)  # start at 0
b = torch.tensor([0.0], requires_grad=True)  # start at 0

x = torch.tensor([1.0, 2.0, 3.0])  # inputs
y_true = torch.tensor([3.0, 5.0, 7.0])  # targets (2*x + 1)

# Training step
y_pred = w * x + b                  # forward pass
loss = ((y_pred - y_true) ** 2).mean()  # MSE loss

loss.backward()                     # compute gradients

print(f"w.grad: {w.grad}")          # gradient w.r.t. w
print(f"b.grad: {b.grad}")          # gradient w.r.t. b

# In real training, we'd update: w = w - lr * w.grad
# That's what optimizers do (next week!)
```

This is the foundation of all neural network training - next week we'll see how `nn.Module` and optimizers build on this.

Exercise Preview

Part 1 Exercises

Practice what you've learned with hands-on tensor exercises.

Exercise 1: Creating Tensors

- Create tensors from lists
- Use factory functions
- Specify data types

Exercise 2: Tensor Operations

- Element-wise arithmetic
- Matrix multiplication
- Reduction operations

Exercise 3: Reshaping

- Reshape tensors
- Understand views vs copies
- Use squeeze/unsqueeze

Exercise 4: Indexing & Slicing

- Basic indexing
- Slicing patterns
- Boolean masking

Exercise 5: Autograd Basics

- Enable gradient tracking
- Compute gradients
- Use no_grad context

File: `exercises/part1-exercises.ipynb` - Open in Jupyter Notebook or VS Code

Summary: Part 1

What We Covered

- **Tensors** are n-dimensional arrays, PyTorch's core data structure
- **Creation**: `torch.tensor()`, `zeros()`, `ones()`, `rand()`, etc.
- **Attributes**: `shape`, `dtype`, `device`
- **Operations**: element-wise, reductions, matrix ops
- **Broadcasting**: automatic shape expansion
- **Reshaping**: `view`, `reshape`, `squeeze`, `unsqueeze`
- **Memory**: views share data, clones copy
- **Indexing**: NumPy-style, boolean masks, advanced indexing

Key Takeaways

- **Autograd** tracks operations and computes gradients automatically
- `requires_grad=True` enables tracking
- `backward()` computes gradients via chain rule
- `zero_grad()` clears accumulated gradients
- `torch.no_grad()` disables tracking for efficiency

Next Week (Part 2)

- Building neural networks with `nn.Module`
- Loss functions and optimizers
- The complete training loop
- Model evaluation

End of Part 1

Next Week: Building & Training Neural Networks

- **nn.Module** - Define layers & forward pass
- **Training Loop** - Forward, loss, backward, step
- **Optimizers** - SGD, Adam & learning rates

Questions? Bring them to the session!