PyTorch Essentials

Instructor: Chandrashekar Babu <training@chandrashekar.info>

https://www.chandrashekar.info/ | **https://www.slashprog.com/**

About This Training

Purpose

To provide a comprehensive, practical understanding of PyTorch fundamentals through detailed explanations and numerous hands-on exercises

Audience

Early-career ML engineers and data scientists looking to master PyTorch for deep learning applications

Prerequisites

Basic Python programming skills and fundamental understanding of machine learning concepts

This training emphasizes practical application and includes numerous code examples and exercises to reinforce learning. We'll progress from basic tensor operations to building complex neural networks, ensuring you gain both theoretical understanding and practical skills.

Training Agenda

01

PyTorch Foundations

Introduction, installation, tensor operations, GPU acceleration, and other key PyTorch features

02

Building Neural Networks

Typing workflow for building neural networks using PyTorch, common nn layers, loss functions and optimizers

03

Advanced Model Architectures

CNNs, RNNs, LSTM, Attention mechanisms and Transfer-Learning

04

Production Workflows

Model optimization, TorchScript, and deployment strategies.

05

Cutting-Edge Features

Distributed training, mixed precision, monitoring, miscellaneous tips and tricks.

About me

A FOSS Technologist and Corporate Trainer with 30+ years of experience in software development, technology consulting and corporate training.

A Pythonista since 1999

Also a polyglot with proficiency in many programming languages that include - Perl, Ruby, Tcl, PHP, Bash, Java, Clojure, C, Zig, Rust, x86 assembly language and many more…

Expertise in diverse technology domains in the Linux ecosystem (Linux Kernel / Device Driver development, MySQL / PostgreSQL Database systems, Apache web server administration and tuning, Web development using PHP, Rails, Flask, software architecture and design principles, agile / adaptive software methodologies).

AI & ML enthusiast following developments (with inclination towards Python frameworks and applied use-cases) since 2016.

To know more about me: kindly visit **https://www.chandrashekar.info/**

Your brief introductions please...

Your designation / role in your organization

Yous experience

Your comfort-level in Python

Your comfort-level in AI concepts and Machine-Learning

Are you already using PyTorch ? If yes, are there any specific expectations you wish to have covered in this course ?

Topics for Today

01

PyTorch Foundations

Introduction, installation, tensor operations, and key neural network concepts

02

Tensors & Autograd

Deep dive into tensor operations, GPU acceleration, and automatic differentiation

03

Building Models

Creating linear models, implementing neural networks from scratch

04

Training & Optimization

Loss functions, backpropagation, and optimizers in practical scenarios

Module 1: PyTorch Foundations

Building a solid understanding of PyTorch basics and neural network fundamentals

What is PyTorch?

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR) in 2016. It provides a flexible, intuitive interface for building and training neural networks.

At its core, PyTorch is designed to be **Pythonic**, meaning it integrates seamlessly with the Python programming language and ecosystem. This design philosophy makes it more accessible and natural to use for Python developers compared to some alternatives.

PyTorch has gained tremendous popularity in the research community and increasingly in industry applications due to its flexibility and ease of debugging.

Unlike some other frameworks that use static computation graphs, PyTorch uses a **dynamic computational graph** paradigm, allowing for more intuitive model development and easier debugging.

Key Features of PyTorch

Pythonic Interface

Intuitive and seamlessly integrated with Python, allowing for natural programming patterns

Dynamic Computation Graphs

Builds graphs on-the-fly, enabling flexibility during development and debugging

GPU Acceleration

Seamless transition between CPU and GPU execution for faster computation

Automatic Differentiation

Built-in autograd for automatic computation of gradients during backpropagation

Rich Ecosystem

Extensive libraries for computer vision (torchvision), NLP (torchtext), and more

Distributed Training

Native support for distributed training across multiple GPUs and machines

These features make PyTorch particularly well-suited for research prototyping and deployment of deep learning models in production environments.

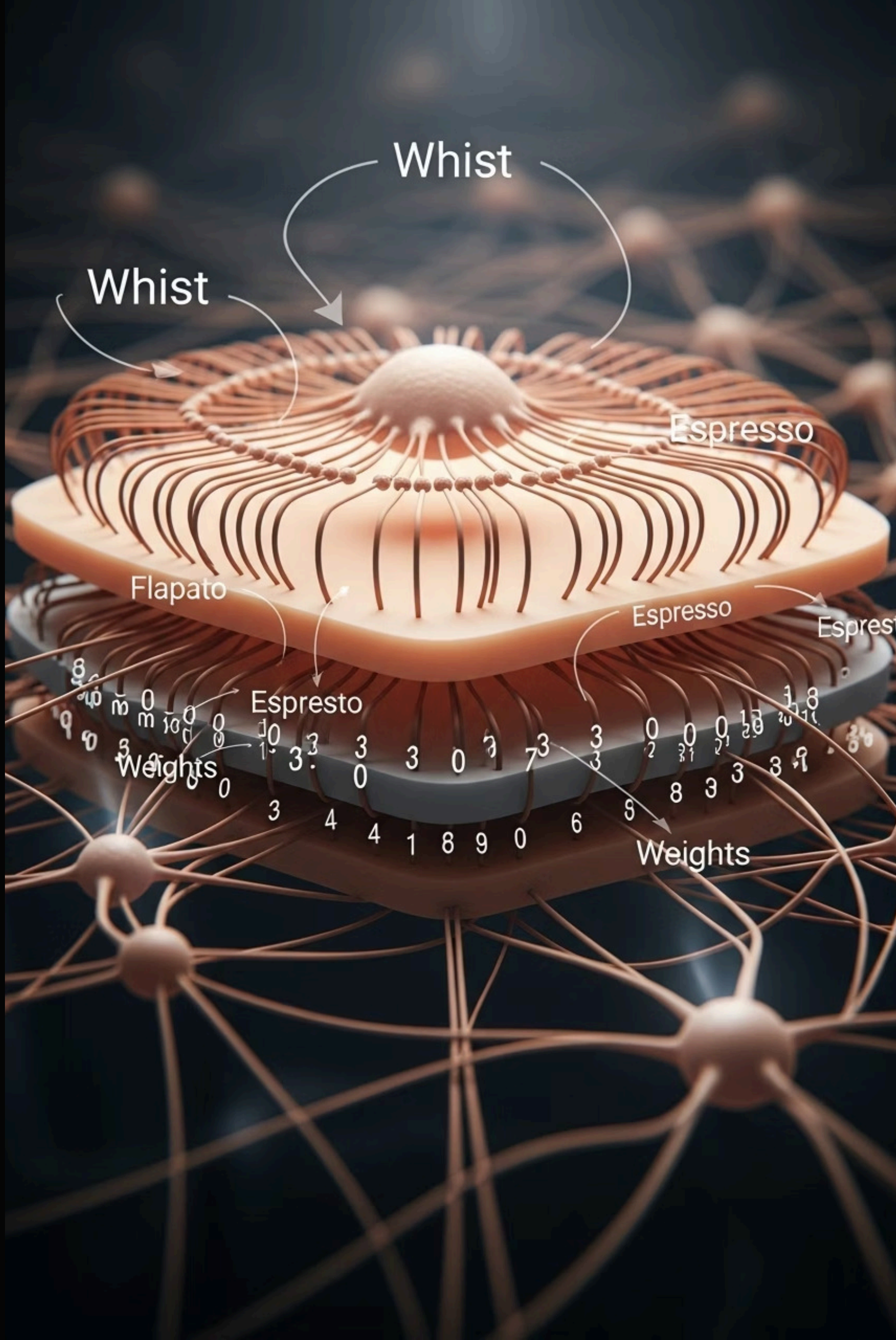
PyTorch vs. TensorFlow vs. Scikit-Learn

Feature	PyTorch	TensorFlow	Scikit-Learn
Primary Use Case	Deep learning research and production	Production-focused deep learning	Classical machine learning algorithms
Computation Graph	Dynamic (define-by-run)	Static with eager execution mode	No explicit computation graph
Learning Curve	Moderate (Pythonic)	Steeper (less Pythonic)	Gentle (highly Pythonic API)
Debugging	Simple (standard Python tools)	More complex	Very simple
Deployment	Improving with TorchScript	Excellent (TensorFlow Serving)	Basic (pickle serialization)
Mobile/Edge Support	PyTorch Mobile	TensorFlow Lite	Limited
Community Focus	Research-oriented	Industry and production	General ML practitioners

Understanding these differences helps you choose the right tool for your specific machine learning tasks. PyTorch excels in research and scenarios requiring flexibility, while TensorFlow may be preferred for production deployment at scale. Scikit-Learn remains the go-to for classical ML algorithms.

Key Components of Neural Networks

Understanding these fundamental building blocks is essential before diving into PyTorch implementation details. Each component plays a specific role in how neural networks learn from data.

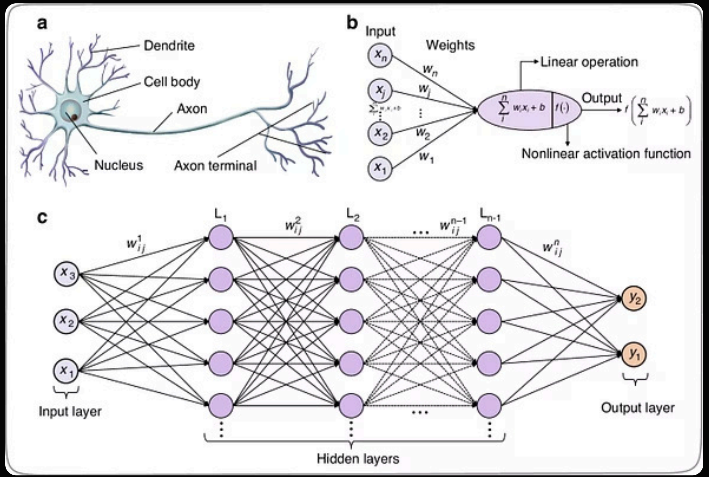


Neural Network Component: Neurons

A neuron (or perceptron) is the fundamental unit of computation in neural networks, inspired by biological neurons in the brain.

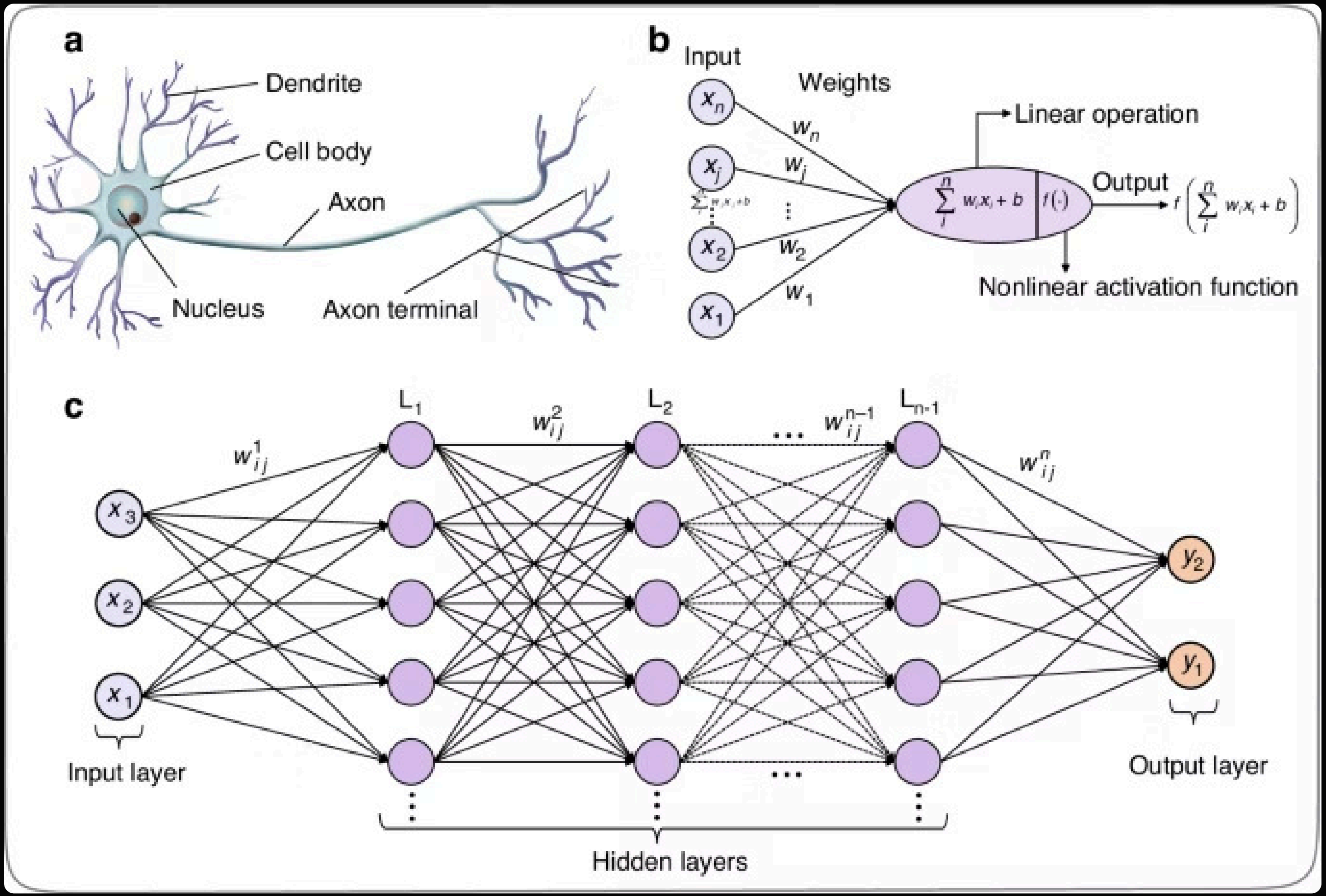
Neuron Components:

- Inputs ($x_1, x_2, ..., x_n$): Data features or outputs from previous layer neurons
- Weights ($w_1, w_2, ..., w_n$): Parameters that determine the importance of each input
- Bias (b): An additional parameter that allows the model to fit the data better
- Summation function: Computes weighted sum ($\sum w_i x_i + b$)
- Activation function: Introduces non-linearity (e.g., ReLU, sigmoid, tanh)



The output of a neuron is typically computed as: $y = f(\sum w_i x_i + b)$, where f is the activation function. In PyTorch, we'll implement this calculation using tensor operations.

Neural Network Component: Layers



Input Layer

Receives the raw input data (features) and passes it to the first hidden layer without transformation. The number of neurons typically equals the number of input features.

Example: For an image of 28×28 pixels, the input layer would have 784 neurons (28×28=784).

Hidden Layers

Perform computations and transfer information from the input layer to the output layer. Deep networks have multiple hidden layers. Each neuron connects to all neurons in adjacent layers.

Example: A network might have hidden layers with 128, 64, and 32 neurons, progressively extracting higher-level features.

Output Layer

Produces the final prediction or classification. The number of neurons depends on the task (e.g., binary classification: 1 neuron; 10-class classification: 10 neurons).

Example: For MNIST digit classification (0-9), the output layer would have 10 neurons, each representing the probability of a specific digit.

In PyTorch, we can define these layers using `torch.nn.Linear` or by creating custom layer classes that inherit from `torch.nn.Module`.

Getting Started with PyTorch

Now that we understand the key neural network concepts, let's set up our PyTorch environment and start working with tensors - the fundamental data structure in PyTorch.

Installing PyTorch Environment

Using pip:

```
pip install torch torchvision torchaudio torchtext
```

Using conda:

```
conda install pytorch torchvision torchaudio -c pytorch
```

With specific CUDA version (for GPU):

```
pip install torch==2.8.0+cu118 torchvision==0.19.0+cu118 \
    torchaudio==2.8.0+cu118 -f
https://download.pytorch.org/whl/torch_stable.html
```

Using pipenv (recommended):

```
# Install pipenv if you don't have it
pip install pipenv

# Create a new environment with PyTorch
pipenv install torch torchvision torchaudio

# Activate the environment
pipenv shell
```

Using pipenv creates a reproducible environment with a Pipfile that locks your dependencies. This ensures consistent behavior across different installations.

Choose the installation method that best fits your workflow. For this training, we'll use PyTorch version 2.8.0, but the concepts apply to other recent versions as well.

Verifying PyTorch Installation

```
import torch
import torchvision

# Print PyTorch version
print(f"PyTorch version: {torch.__version__}")

# Set device based on GPU availability (PyTorch 2.5.1+)
device = torch.accelerator.current_accelerator().type  \
        if torch.accelerator().is_available()         \
        else "cpu"
print(f"Using device: {device}")

# Set device based on GPU availability for older versions of PyTorch (e.g. CUDA)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# If GPU is available, print additional information
print("Number of devices:", torch.accelerator.device_count())
print("Current device (index):", torch.accelerator.current_device_index())
print("Current device:", torch.accelerator.current_accelerator())

# Device info for "mps" (Metal Performance Shaders - MacBook) if available
if torch.mps.is_available():
    print("Number of devices:", torch.mps.device_count())
    print(f"Current allocated memory: {torch.mps.current_allocated_memory():,} bytes")
    print(f"Driver allocated memory: {torch.mps.driver_allocated_memory():,} bytes")
    print(f"Recommended max memory: {torch.mps.recommended_max_memory():,} bytes")

# Device info for "CUDA" (NVIDIA GPUs) if available
if torch.cuda.is_available():
    print(f"Current CUDA device: {torch.cuda.current_device()}")
    print(f"Device name: {torch.cuda.get_device_name(0)}")
    print(f"Device memory: {torch.cuda.get_device_properties(0).total_memory / 1e9:.2f} GB")
    print(f"CUDA version: {torch.version.cuda}")

# Create a simple tensor to test
x = torch.rand(5, 3)
print(f"Random tensor:\n{x}")

# Move tensor to available device
x = x.to(device)
print(f"Tensor on {device}:\n{x}")
```

Run this code to verify that PyTorch is correctly installed and to check if you have GPU support. If PyTorch can utilize your GPU, the "Using device" line will either show "cuda" / "mps" / others; otherwise, it will show "cpu". The tensor operations should work regardless of the device.

This verification step is crucial before proceeding with any deep learning work, as GPU acceleration can make training significantly faster.

Understanding Tensor Dimensions

In PyTorch, tensor is a common term used for representing 0-d, 1-d, 2-d, n-d arrays

Scalar (0D Tensor)

A single value

```
scalar = torch.tensor(7)
print(scalar)  # tensor(7)
print(scalar.shape)  # ()
print(scalar.ndim)   # 0
```

Vector (1D Tensor)

A list of values

```
vector = torch.tensor([1, 2, 3])
print(vector)  # tensor([1, 2, 3])
print(vector.shape)  # (3,)
print(vector.ndim)   # 1
```

Matrix (2D Tensor)

A table of values

```
matrix = torch.tensor([[1, 2], [3, 4]])
print(matrix)  # tensor([[1, 2], [3, 4]])
print(matrix.shape)  # (2, 2)
print(matrix.ndim)   # 2
```

Tensor (3D+ Tensor)

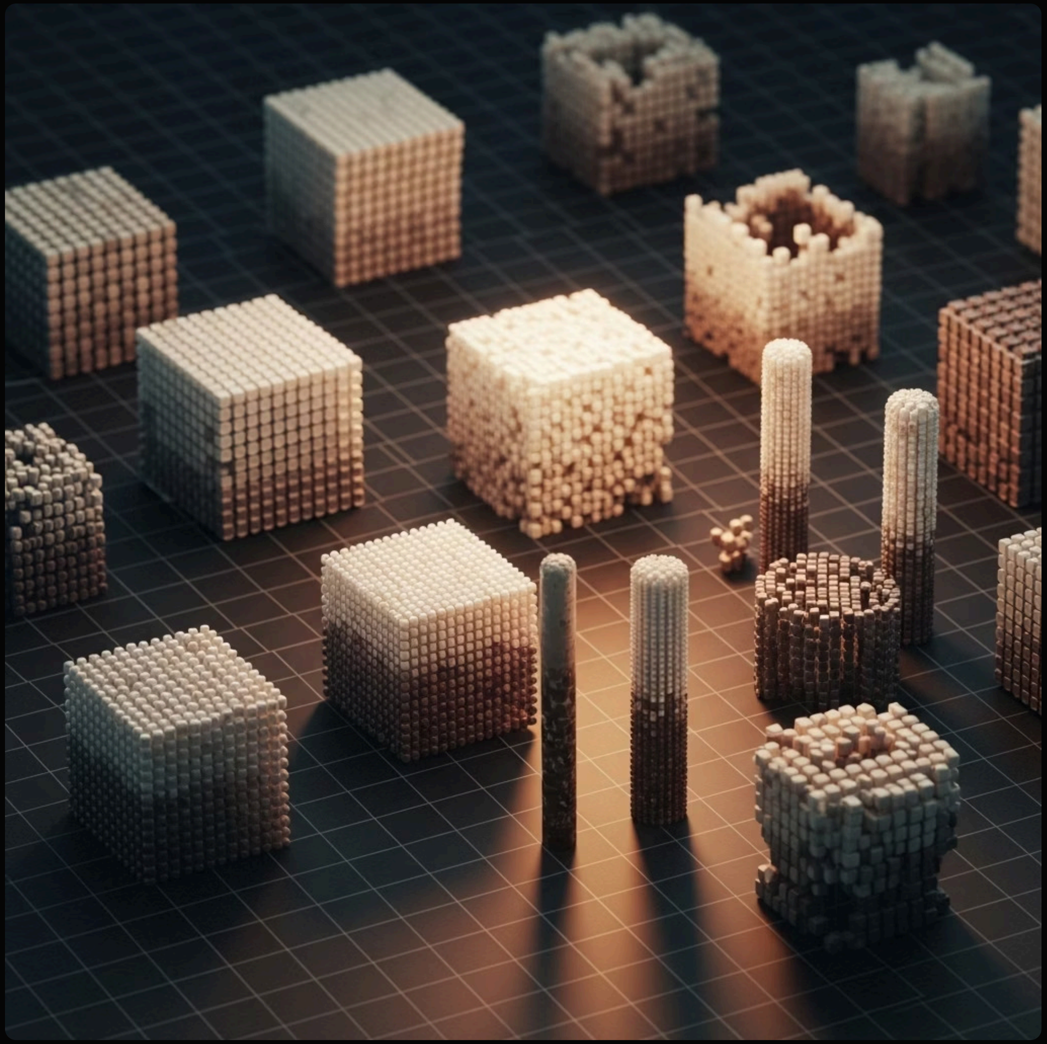
Multidimensional arrays

```
tensor_3d = torch.rand(2, 3, 4)
print(tensor_3d.shape)  # (2, 3, 4)
print(tensor_3d.ndim)   # 3
```

Understanding tensor dimensions is crucial for deep learning. For example, a batch of images might be represented as a 4D tensor with dimensions [batch_size, channels, height, width]. You can access a tensor's shape using the `.shape` attribute and its number of dimensions using `.ndim`.

Creating Basic Tensors

```
import torch

# Print the version of PyTorch
print(torch.__version__)

# Uninitialized tensor (arbitrary values)
x1 = torch.empty(3, 4)
print(f"Empty tensor:\n{x1}\n")

# Zeros tensor
x2 = torch.zeros(2, 3)
print(f"Zeros tensor:\n{x2}\n")

# Ones tensor
x3 = torch.ones(2, 2)
print(f"Ones tensor:\n{x3}\n")

# Tensor with specific values
x4 = torch.tensor([2.5, 3.5])
print(f"Specific values tensor:\n{x4}\n")

# Random tensor (uniform distribution [0,1])
x5 = torch.rand(2, 3)
print(f"Random tensor:\n{x5}\n")

# Random tensor (normal distribution μ=0, σ=1)
x6 = torch.randn(2, 3)
print(f"Normal distribution tensor:\n{x6}\n")
```



Key Points:

- `torch.empty()` creates a tensor with uninitialized values
- `torch.zeros()` and `torch.ones()` create tensors filled with 0s and 1s
- `torch.tensor([])` creates a tensor from a Python list or array
- `torch.rand()` creates tensors with random values from a uniform distribution [0,1]
- `torch.randn()` creates tensors with values from a normal distribution (μ=0, σ=1)

Tensor Attributes and Properties

```
import torch

# Create a tensor
x = torch.randn(3, 4, device="cpu", dtype=torch.float32)

# Basic attributes
print(f"Tensor: {x}")
print(f"Shape: {x.shape}")          # Size: (3, 4)
print(f"Dimensions: {x.ndim}")      # Number of dimensions: 2
print(f"Size: {x.size()}")          # Another way to get shape: torch.Size([3, 4])
print(f"Data type: {x.dtype}")      # Data type: torch.float32
print(f"Device: {x.device}")        # Device: cpu

# Memory information
print(f"Element size in bytes: {x.element_size()}")  # Size of each element in bytes
print(f"Total elements: {x.numel()}")                # Total number of elements: 12
print(f"Memory usage: {x.element_size() * x.numel()} bytes")  # Total memory usage

# Other properties
print(f"Requires gradient: {x.requires_grad}")  # False by default
print(f"Is contiguous: {x.is_contiguous()}")    # Whether tensor is stored contiguously in memory
```

Understanding tensor properties is essential for debugging, memory optimization, and ensuring your models work correctly. Pay special attention to `shape`, `dtype`, and `device`, as mismatches in these properties are common sources of errors in PyTorch code.

Contiguous vs Non-contiguous Tensors

Tensors can become non-contiguous in memory by performing an operation that changes the logical order of elements without changing their physical memory layout, such as transposing a tensor.

```
import torch

# Create a contiguous 2D tensor
a = torch.arange(12).reshape(3, 4)
print("Original contiguous tensor:", a)
print("Is contiguous:", a.is_contiguous())

# Transpose the tensor, creating a non-contiguous view
b = a.T
print("\nTransposed (non-contiguous) tensor:", b)
print("Is contiguous:", b.is_contiguous())

# Attempting to use .view() on a non-contiguous tensor will raise an error
try:
    b.view(-1)
except RuntimeError as e:
    print(f"\nError: {e}")

# Make the non-contiguous tensor contiguous using .contiguous()
c = b.contiguous()
print("\nTensor after calling .contiguous():")
print(c)
print("Is contiguous:", c.is_contiguous())

# Now .view() will work
print("\nViewing the now contiguous tensor:")
print(c.view(-1))
```

PyTorch Tensor Features

NumPy-like Interface

PyTorch tensors have an API similar to NumPy arrays, making them familiar to data scientists. This includes indexing, slicing, and many array manipulation functions.

```
# NumPy-like slicing
x = torch.randn(5, 3)
print(x[1:4, 0:2])  # Rows 1-3, columns 0-1
```

Automatic Differentiation

PyTorch tensors can track operations for automatic gradient computation, essential for neural network training.

```
x = torch.ones(2, 2, requires_grad=True)
y = x * 2
z = y.mean()
z.backward()  # Computes gradients
print(x.grad)  # Shows d(z)/d(x)
```

GPU Acceleration

Tensors can be moved between CPU and GPU seamlessly for accelerated computation.

```
# Move tensor to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
x = torch.randn(10000, 10000)
x = x.to(device)  # Move to GPU
y = x * x        # Operation runs on GPU
```

Dynamic Computation Graphs

PyTorch builds graphs on-the-fly during execution, allowing for more intuitive debugging and flexible model architecture.

```
def dynamic_layer(x, weight):
 if x.sum() > 0:
 return x * weight
 else:
 return x + weight
```

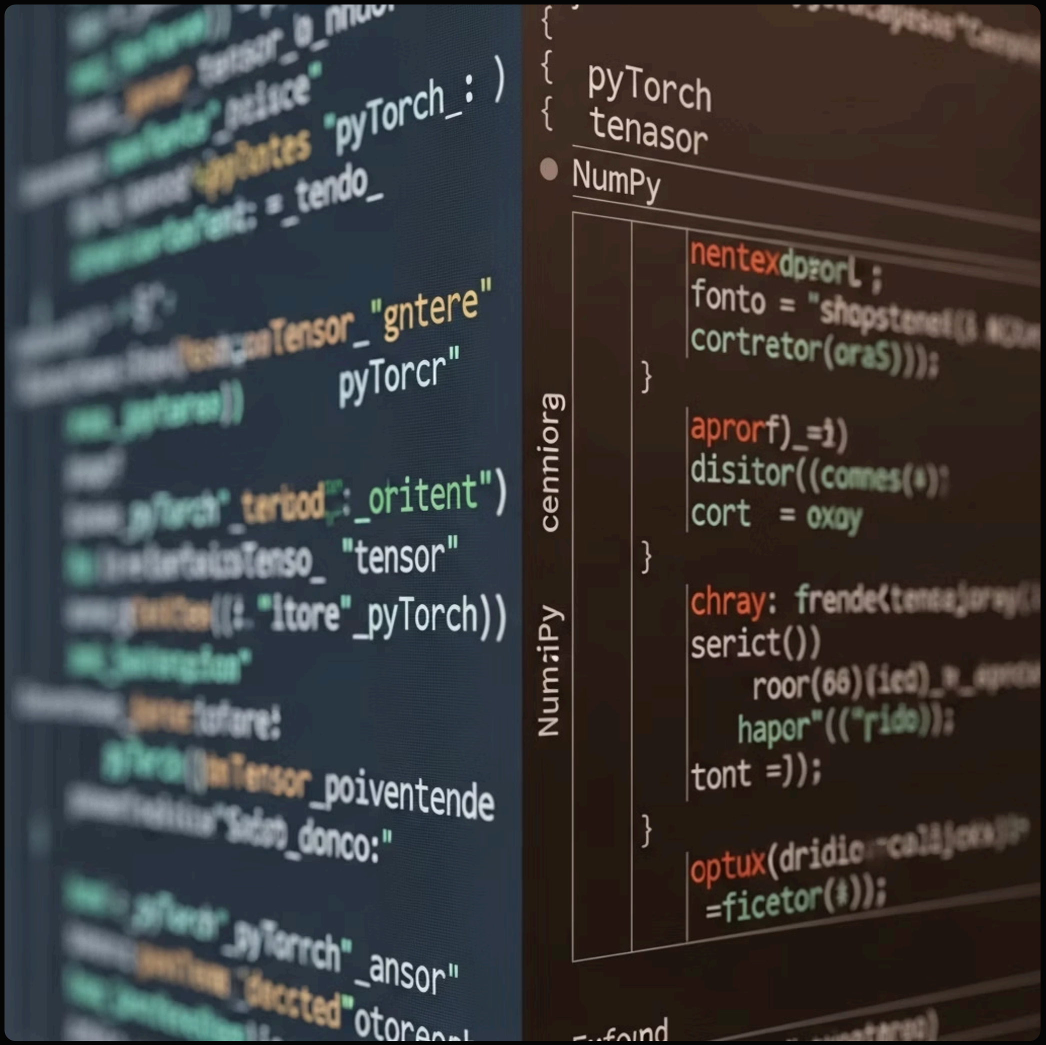
Tensors vs. NumPy Arrays

Similarities:

- Both represent multi-dimensional arrays
- Similar APIs for indexing, slicing, and reshaping
- Both support broadcasting operations
- Efficient memory usage and vectorized operations

Key Differences:

- PyTorch tensors support automatic differentiation
- Tensors can utilize GPU acceleration
- PyTorch optimizes for deep learning operations
- NumPy has more comprehensive scientific computing functions



Converting Between Tensors and NumPy:

```
import torch
import numpy as np

# NumPy array to PyTorch tensor
numpy_array = np.random.rand(3, 4)
torch_tensor = torch.from_numpy(numpy_array)

# PyTorch tensor to NumPy array
tensor = torch.rand(3, 4)
numpy_array = tensor.numpy()

# Note: Conversions share memory when possible
# (changes in one can affect the other)
```

NumPy-like Operations in PyTorch

Mathematical Operations

```
# Element-wise operations
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
print(a + b)       # tensor([5, 7, 9])
print(a * b)       # tensor([4, 10, 18])

# Matrix multiplication
x = torch.randn(2, 3)
y = torch.randn(3, 4)
z = torch.matmul(x, y)  # or x @ y
print(z.shape)     # torch.Size([2, 4])
```

Broadcasting

```
# Broadcasting operations
x = torch.randn(5, 3)
y = torch.randn(3)
z = x + y  # y is broadcast to match x's shape
print(z.shape)  # torch.Size([5, 3])

# Broadcasting with unsqueezed dimensions
batch = torch.randn(32, 10)
scale = torch.randn(10).unsqueeze(0)  # shape: [1, 10]
scaled = batch * scale  # scale broadcast to [32, 10]
```

Reshaping & Transposing

```
# Reshaping
x = torch.randn(12)
y = x.reshape(3, 4)  # or x.view(3, 4)
print(y.shape)  # torch.Size([3, 4])

# Transposing
z = y.transpose(0, 1)  # or y.t() for 2D
print(z.shape)  # torch.Size([4, 3])

# Permuting dimensions
a = torch.randn(2, 3, 4)
b = a.permute(2, 0, 1)
print(b.shape)  # torch.Size([4, 2, 3])
```

Flattening & Stacking

```
# Flattening
x = torch.randn(2, 3, 4)
flat = x.flatten() # All dimensions
print(flat.shape) # torch.Size([24])
flat2 = x.flatten(1) # Only from dim 1
print(flat2.shape) # torch.Size([2, 12])

# Stacking tensors
a = torch.randn(3, 4)
b = torch.randn(3, 4)
c = torch.stack([a, b]) # New dimension
print(c.shape) # torch.Size([2, 3, 4])
d = torch.cat([a, b], dim=0) # Concatenate
print(d.shape) # torch.Size([6, 4])
```

Advanced Tensor Operations

Aggregation Operations:

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

# Sum, mean, max, min
print(x.sum())      # tensor(21)
print(x.mean())     # tensor(3.5000)
print(x.max())      # tensor(6)
print(x.min())      # tensor(1)

# Along specific dimension
print(x.sum(dim=0))  # tensor([5, 7, 9])
print(x.mean(dim=1)) # tensor([2., 5.])

# With keepdim to preserve dimensions
print(x.sum(dim=1, keepdim=True))
# tensor([[6], [15]])
```

Element-wise Operations:

```
# Apply function to each element
y = torch.randn(3, 4)
print(torch.abs(y))      # Absolute value
print(torch.sqrt(torch.abs(y)))  # Square root of abs
print(torch.exp(y))      # Exponential
print(torch.log(torch.abs(y)))   # Natural log

# Comparison operations
z = torch.randn(3, 4)
print(y > z)        # Element-wise comparison
print(torch.eq(y, z))  # Equality comparison
print(torch.all(y > 0))  # Are all elements > 0?
print(torch.any(y > 0))  # Are any elements > 0?
```

These operations allow for efficient manipulation of tensor data, whether for preprocessing inputs, implementing model architectures, or analyzing outputs. Most operations support batch processing, making them highly efficient for deep learning workflows.

Converting Between PyTorch and NumPy

```
import torch
import numpy as np

# Creating a NumPy array
numpy_array = np.random.rand(3, 4).astype(np.float32)
print(f"NumPy array:\n{numpy_array}")
print(f"Type: {type(numpy_array)}")

# Convert NumPy array to PyTorch tensor
pytorch_tensor = torch.from_numpy(numpy_array)
print(f"\nPyTorch tensor from NumPy:\n{pytorch_tensor}")
print(f"Type: {type(pytorch_tensor)}")

# Modify the NumPy array (affects the tensor if memory is shared)
numpy_array[0, 0] = 100
print(f"\nModified NumPy array:\n{numpy_array}")
print(f"PyTorch tensor after NumPy modification:\n{pytorch_tensor}")

# Creating a PyTorch tensor
tensor = torch.randn(3, 4)
print(f"\nOriginal PyTorch tensor:\n{tensor}")

# Convert PyTorch tensor to NumPy array
numpy_from_tensor = tensor.numpy()
print(f"\nNumPy array from PyTorch:\n{numpy_from_tensor}")
print(f"Type: {type(numpy_from_tensor)}")

# Modify the tensor (affects the NumPy array if memory is shared)
tensor[0, 0] = 200
print(f"\nModified PyTorch tensor:\n{tensor}")
print(f"NumPy array after tensor modification:\n{numpy_from_tensor}")

# Important note: GPU tensors must be moved to CPU before conversion
gpu_tensor = torch.randn(3, 4)
if torch.cuda.is_available():
    gpu_tensor = gpu_tensor.cuda()
    cpu_tensor = gpu_tensor.cpu()  # Move to CPU first
    numpy_from_gpu = cpu_tensor.numpy()
    print(f"\nNumPy array from GPU tensor:\n{numpy_from_gpu}")
```

Key Point: When converting between PyTorch tensors and NumPy arrays, they often share the same underlying memory for efficiency. This means that changes to one can affect the other, as demonstrated above. However, this sharing only happens for CPU tensors - GPU tensors must be moved to CPU before conversion.

Advantages of PyTorch Tensors Over NumPy Arrays

GPU Acceleration

PyTorch tensors can seamlessly utilize CUDA-enabled GPUs for parallel computation, providing significant speedups (10-100x) for large operations.

```
# Example: Matrix multiplication speedup
import time

# CPU computation
x_cpu = torch.randn(5000, 5000)
y_cpu = torch.randn(5000, 5000)
start = time.time()
z_cpu = torch.matmul(x_cpu, y_cpu)
cpu_time = time.time() - start

# GPU computation (if available)
if torch.cuda.is_available():
    x_gpu = x_cpu.cuda()
    y_gpu = y_cpu.cuda()
    start = time.time()
    z_gpu = torch.matmul(x_gpu, y_gpu)
    torch.cuda.synchronize()  # Wait for GPU
    gpu_time = time.time() - start
    print(f"CPU time: {cpu_time:.2f}s")
    print(f"GPU time: {gpu_time:.2f}s")
    print(f"Speedup: {cpu_time/gpu_time:.1f}x")
```

Automatic Differentiation

PyTorch tensors can track their own computation history, enabling automatic gradient calculation - the foundation of neural network training.

```
# Example: Computing gradients automatically
x = torch.tensor([2.0, 3.0], requires_grad=True)
y = x**2 + 3*x + 1
print(f"y = {y}")  # tensor([11., 19.], grad_fn=...)

# Compute gradient of sum(y) with respect to x
y.sum().backward()
print(f"dy/dx = {x.grad}")  # tensor([7., 9.])

# Compare with analytical derivative:
# d(x^2 + 3x + 1)/dx = 2x + 3
# At x=2: 2(2) + 3 = 7
# At x=3: 2(3) + 3 = 9
```

Deep Learning Optimizations

PyTorch includes specialized operations optimized for deep learning that aren't available in NumPy.

```
import torch.nn.functional as F

# Example: Deep learning specific operations
inputs = torch.randn(32, 10)  # 32 samples, 10 features
weights = torch.randn(20, 10)  # 20 outputs, 10 inputs
bias = torch.randn(20)

# Linear layer (optimized matrix multiplication + bias)
outputs = F.linear(inputs, weights, bias)
print(f"Linear layer output shape: {outputs.shape}")

# Activation functions
relu_outputs = F.relu(outputs)
print(f"After ReLU: {relu_outputs.shape}")

# Softmax for classification
probs = F.softmax(outputs, dim=1)
print(f"Probability distribution: {probs.sum(dim=1)}")
```

Dynamic Computation Graph

PyTorch builds computation graphs dynamically during execution, allowing for more flexible control flow and easier debugging.

```
# Example: Dynamic control flow in PyTorch
def dynamic_network(x, condition):
 if condition:
 return x**2
 else:
 return x**3

# Works with autograd
x = torch.tensor([2.0], requires_grad=True)
y1 = dynamic_network(x, True) # x^2
y2 = dynamic_network(x, False) # x^3

print(f"y1 = {y1}")
y1.backward()
print(f"dy1/dx = {x.grad}") # Should be 2*x = 4

x.grad.zero_() # Reset gradient
y2.backward()
print(f"dy2/dx = {x.grad}") # Should be 3*x^2 = 12
```

Features Exclusive to PyTorch Tensors

1. CUDA Support

PyTorch provides seamless GPU acceleration that NumPy doesn't offer:

```
# Check for CUDA availability
print(f"CUDA available: {torch.cuda.is_available()}")

if torch.cuda.is_available():
    # List available GPUs
    print(f"Number of GPUs: {torch.cuda.device_count()}")
    for i in range(torch.cuda.device_count()):
        print(f"GPU {i}: {torch.cuda.get_device_name(i)}")

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

    # Move existing tensor to GPU
    x_cpu = torch.rand(1000, 1000)
    x_gpu = x_cpu.cuda()  # or x_cpu.to("cuda")

    # Operations happen on GPU
    y_gpu = x_gpu @ x_gpu  # Matrix multiplication

    # Move back to CPU when needed
    y_cpu = y_gpu.cpu()
```

2. Autograd System

PyTorch's automatic differentiation capabilities:

```
# Create tensors with gradient tracking
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()

# Compute gradients
out.backward()

# Access gradients
print(f"x.grad: {x.grad}")

# Detach from computation graph
y_detached = y.detach()  # No gradient tracking

# Control gradient tracking
with torch.no_grad():
    # No operations here will track gradients
    w = x * 2
print(f"w.requires_grad: {w.requires_grad}")
```

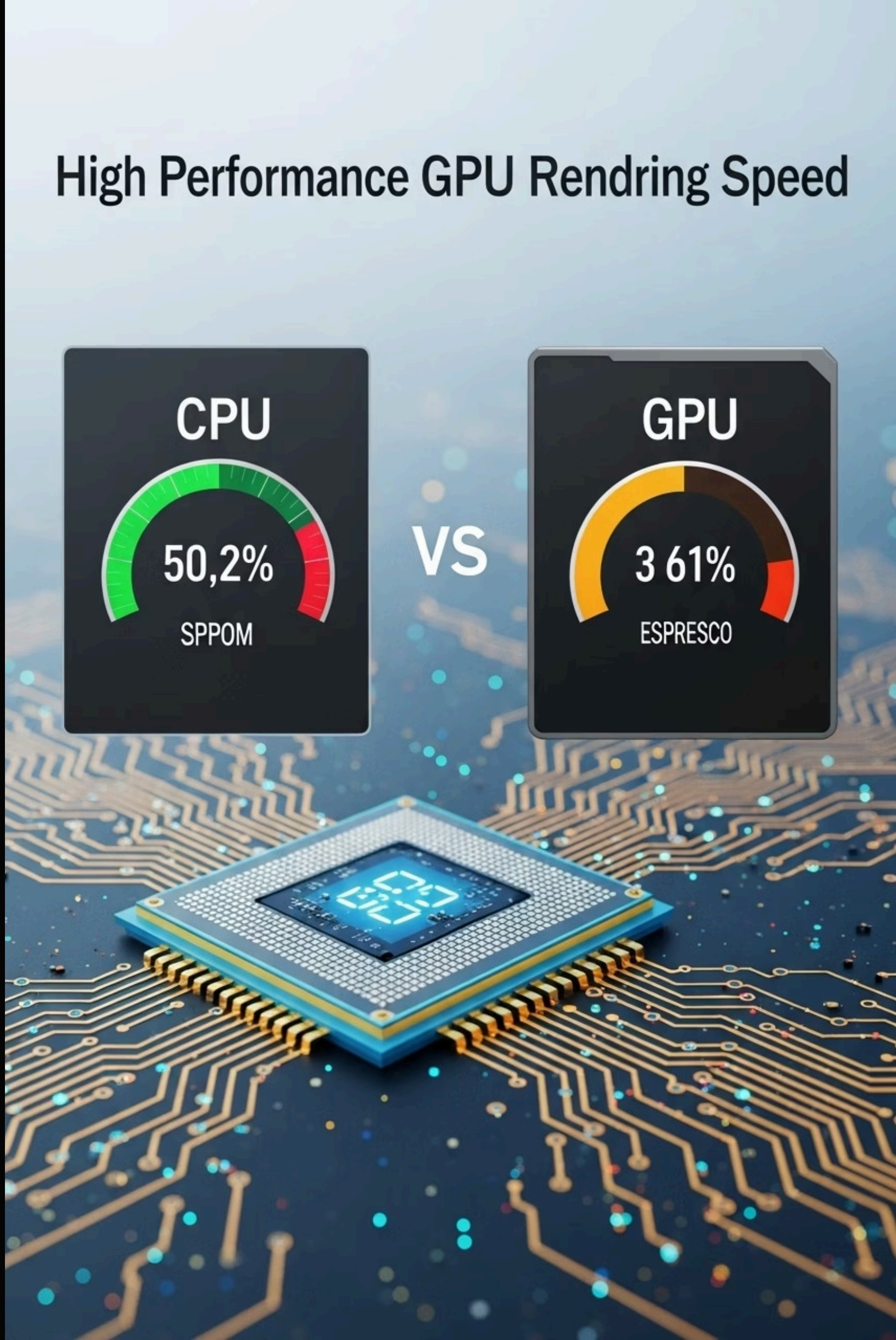
3. Neural Network Operations

Specialized functions for deep learning:

```
import torch.nn.functional as F

# Deep learning specific operations
x = torch.randn(3, 5)
w = torch.randn(5, 10)
b = torch.randn(10)

# Linear layer
out = F.linear(x, w, b)
# Activations
relu_out = F.relu(out)
sigmoid_out = F.sigmoid(out)
# Pooling
maxpool = F.max_pool1d(x.unsqueeze(0), kernel_size=2)
```

GPU Acceleration in PyTorch

GPU acceleration is one of PyTorch's most powerful features, enabling significant speedups for deep learning computations. Let's look at how to effectively use GPUs with PyTorch.



Moving Tensors Between CPU and GPU

```
import torch

# Check for GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Method 1: Create tensor directly on target device
x = torch.rand(5, 3, device=device)
print(f"Tensor x is on: {x.device}")

# Method 2: Move existing tensor to device using .to()
y = torch.rand(5, 3)  # Created on CPU by default
y = y.to(device)      # Move to available device
print(f"Tensor y is on: {y.device}")

# Method 3: Using specific CUDA functions (when you know GPU is available)
if device.type == "cuda":
    z_cpu = torch.rand(5, 3)
    z_gpu = z_cpu.cuda()   # Equivalent to .to("cuda")
    print(f"Tensor z is on: {z_gpu.device}")

    # Moving back to CPU
    z_back = z_gpu.cpu()
    print(f"Tensor z_back is on: {z_back.device}")

# Checking device for operations
if device.type == "cuda":
    a = torch.rand(1000, 1000, device=device)
    b = torch.rand(1000, 1000, device=device)

    # Operation happens on GPU
    c = torch.matmul(a, b)
    print(f"Result tensor c is on: {c.device}")
```

Moving tensors between devices is a common operation in PyTorch workflows. The `.to(device)` method is the most flexible approach as it works regardless of whether the target device is CPU or GPU. Always ensure that tensors participating in the same operation are on the same device, or you'll encounter errors.

Rules for GPU Tensor Operations

Device Consistency

Tensors involved in an operation must be on the same device

```
# This will cause an error
x_cpu = torch.rand(2, 3)
y_gpu = torch.rand(2, 3, device="cuda")
# z = x_cpu + y_gpu  # Error!

# Correct approach
z = x_cpu.cuda() + y_gpu  # Move to same device
# Or
z = x_cpu + y_gpu.cpu()   # Move to same device
```

Model and Data Alignment

Models and their inputs must be on the same device

```
import torch.nn as nn

# Create a model and move it to GPU
model = nn.Linear(10, 5)
model = model.to(device)

# Input must be on same device as model
input_data = torch.rand(3, 10)
input_data = input_data.to(device)  # Move to same device

# Now we can do the forward pass
output = model(input_data)
```

Memory Management

GPU memory is limited and must be managed carefully

```
# Free up memory when tensors are no longer needed
large_tensor = torch.rand(10000, 10000, device="cuda")
result = large_tensor.sum()
del large_tensor  # Free up GPU memory

# Clear cache if needed
torch.cuda.empty_cache()

# Check memory usage
if torch.cuda.is_available():
    print(f"Allocated: {torch.cuda.memory_allocated()/1e9:.2f}
GB")
    print(f"Cached: {torch.cuda.memory_reserved()/1e9:.2f}
GB")
```

Synchronization

GPU operations are asynchronous by default

```
# Operations on GPU run asynchronously
start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)

start.record()
# GPU operation
a = torch.randn(10000, 10000, device="cuda")
b = a * a
end.record()

# Wait for the operation to finish
torch.cuda.synchronize()

# Get elapsed time
print(f"Time: {start.elapsed_time(end)} ms")
```

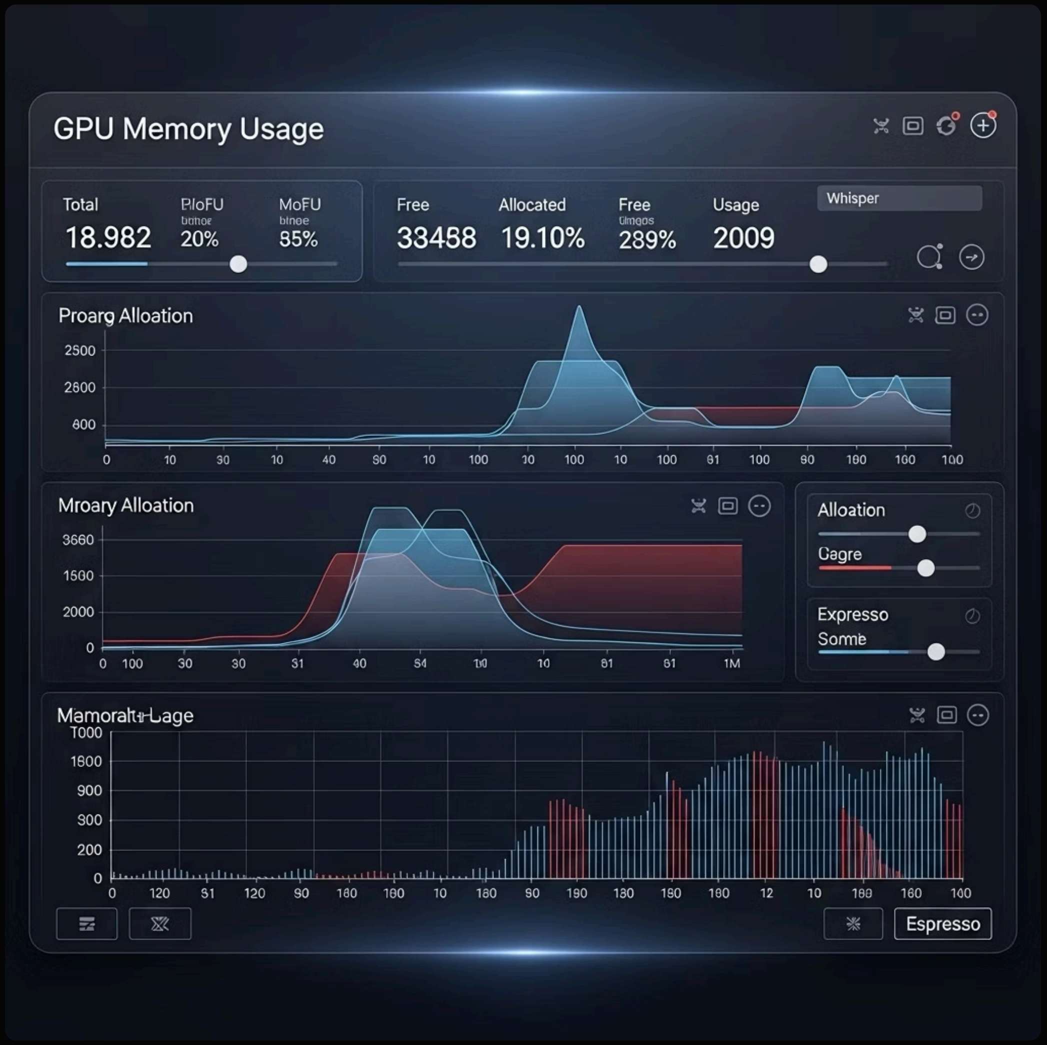
GPU Memory Management and Diagnostics

Monitoring GPU Memory Usage:

```
# Get current memory allocation
allocated = torch.cuda.memory_allocated()
print(f"Allocated: {allocated/1e9:.2f} GB")

# Get maximum memory allocated since program start
max_allocated = torch.cuda.max_memory_allocated()
print(f"Max allocated: {max_allocated/1e9:.2f} GB")

# Get current memory reserved by cache
reserved = torch.cuda.memory_reserved()
print(f"Reserved: {reserved/1e9:.2f} GB")

# Get maximum memory reserved
max_reserved = torch.cuda.max_memory_reserved()
print(f"Max reserved: {max_reserved/1e9:.2f} GB")

# Reset peak stats
torch.cuda.reset_peak_memory_stats()
```

Memory Management Techniques:

```
# 1. Explicit deletion of tensors
large_tensor = torch.rand(5000, 5000, device="cuda")
del large_tensor
torch.cuda.empty_cache()

# 2. Use context manager for controlled memory usage
with torch.no_grad():  # Disable gradient tracking
    # Computation without gradient tracking
    x = torch.rand(1000, 1000, device="cuda")
    y = x * x

# 3. Move tensors to CPU when not in use
x_gpu = torch.rand(5000, 5000, device="cuda")
x_cpu = x_gpu.cpu()  # Move to CPU
del x_gpu  # Free GPU memory
torch.cuda.empty_cache()
# Later...
x_gpu = x_cpu.cuda()  # Move back when needed

# 4. Use gradient checkpointing for large models
# (saves memory by recomputing intermediate
# activations during backward pass)
```



Automatic Differentiation with Autograd

Automatic differentiation is the engine that powers neural network training in PyTorch. It allows the framework to automatically compute gradients of tensors, which are essential for optimizing model parameters using gradient-based methods like stochastic gradient descent.

Introduction to Automatic Differentiation

Automatic differentiation (autograd) is a technique for calculating derivatives of functions through computational graphs. Unlike numerical differentiation (which approximates derivatives) or symbolic differentiation (which derives analytical expressions), autograd computes exact derivatives efficiently by applying the chain rule through a computational graph.

Key Concepts:

- **Forward Pass:** Computes the function's output while building a computational graph
- **Backward Pass:** Propagates gradients from outputs to inputs using the chain rule
- **Computational Graph:** A directed acyclic graph representing operations and their dependencies
- **Chain Rule:** Mathematical rule for computing derivatives of composite functions

Consider a simple example: $f(x) = x^2 + 2x$. The derivative is $f'(x) = 2x + 2$.

```
# Manual calculation
x_value = 3
derivative = 2 * x_value + 2  # = 8

# With autograd
import torch
x = torch.tensor([3.0], requires_grad=True)
f = x**2 + 2*x
f.backward()
print(f"df/dx at x=3: {x.grad}")  # Should be 8
```

PyTorch's autograd system builds a computational graph during the forward pass and automatically applies the chain rule during the backward pass to compute gradients with respect to any tensor marked with `requires_grad=True`.

How Autograd Works

Enable Gradient Tracking

Create tensors with `requires_grad=True` to tell PyTorch to track operations on these tensors.

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

Forward Pass

Perform operations on the tensors. PyTorch records these operations in a computational graph.

```
y = x**2
z = y.sum()  # z = x[0]^2 + x[1]^2 = 13
```

Backward Pass

Call `.backward()` on the output to compute gradients of all tensors in the graph that require gradients.

```
z.backward()  # Compute dz/dx
```

Access Gradients

The gradients are stored in the `.grad` attribute of the input tensors.

```
print(x.grad)  # tensor([4., 6.])
# dz/dx[0] = d(x[0]^2)/dx[0] = 2*x[0] = 2*2 = 4
# dz/dx[1] = d(x[1]^2)/dx[1] = 2*x[1] = 2*3 = 6
```

The autograd system is particularly powerful because it can handle arbitrary computational graphs and automatically compute gradients through complex operations. This is what enables neural networks to learn from data.

Using Autograd in PyTorch

```
import torch

# Create tensors with gradient tracking
x = torch.tensor([[1.0, 2.0], [3.0, 4.0]], requires_grad=True)
print(f"x: {x}")

# Perform operations (building the computational graph)
y = x**2 + 2*x
print(f"y: {y}")
print(f"y's grad_fn: {y.grad_fn}")  # Shows the operation that created y

# Further operations
z = y.mean()  # z = mean(x^2 + 2x)
print(f"z: {z}")
print(f"z's grad_fn: {z.grad_fn}")

# Backward pass (compute gradients)
z.backward()  # Equivalent to z.backward(torch.tensor(1.0))

# Access gradients
print(f"x.grad: {x.grad}")  # dz/dx

# Explanation of the gradient calculation:
# z = mean(x^2 + 2x) = sum(x^2 + 2x) / 4
# dz/dx = d(sum(x^2 + 2x))/dx / 4 = (2x + 2) / 4 = x/2 + 1/2

# Verification:
# For x[0,0] = 1: 1/2 + 1/2 = 1
# For x[0,1] = 2: 2/2 + 1/2 = 1.5
# For x[1,0] = 3: 3/2 + 1/2 = 2
# For x[1,1] = 4: 4/2 + 1/2 = 2.5

# Reset gradients (gradients accumulate by default)
x.grad.zero_()

# Another computation with the same tensor
y2 = x.sum()
y2.backward()
print(f"x.grad after y2.backward(): {x.grad}")  # Should be all 1s
```

Autograd works by building a computational graph during the forward pass and then applying the chain rule during the backward pass. The `grad_fn` attribute shows what operation created a tensor, forming nodes in this graph. When `.backward()` is called, gradients are computed and stored in the `.grad` attribute of tensors with `requires_grad=True`.

Controlling Gradient Computation

Detaching Tensors:

```
# Create a tensor with gradient tracking
x = torch.tensor([2.0, 3.0], requires_grad=True)
y = x**2

# Detach y from the computation graph
y_detached = y.detach()
z = y_detached * 2  # No gradient tracking for this op

# This would fail because z doesn't have grad_fn
# z.backward()
```

No Gradient Mode:

```
# Context manager to temporarily disable gradient tracking
with torch.no_grad():
    a = x * 2  # a doesn't require gradients
    print(f"a.requires_grad: {a.requires_grad}")

# Outside the context, gradient tracking is enabled again
b = x * 2
print(f"b.requires_grad: {b.requires_grad}")
```

Setting requires_grad:

```
# Create tensor without gradient tracking
a = torch.rand(3, 3)
print(f"a.requires_grad: {a.requires_grad}")

# Enable gradient tracking
a.requires_grad_(True)  # Note the underscore
print(f"a.requires_grad: {a.requires_grad}")

# Disable gradient tracking
a.requires_grad_(False)
print(f"a.requires_grad: {a.requires_grad}")
```

Gradient Accumulation:

```
# Gradients accumulate by default
x = torch.tensor([1.0], requires_grad=True)
y1 = x * 2
y1.backward()
print(f"x.grad after first backward: {x.grad}")

y2 = x * 3
y2.backward()
print(f"x.grad after second backward: {x.grad}")  # 2+3=5

# Reset gradients when needed
x.grad.zero_()
print(f"x.grad after zero_(): {x.grad}")
```

Custom Autograd Functions

```
import torch

# Define a custom autograd function by subclassing torch.autograd.Function
class CustomReLU(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input_tensor):
        """
        Forward pass computation.

        Args:
            ctx: A context object to store information for backward
            input_tensor: Input tensor

        Returns:
            Output tensor
        """
        ctx.save_for_backward(input_tensor)  # Save input for backward pass
        return input_tensor.clamp(min=0)  # ReLU operation: max(0, x)

    @staticmethod
    def backward(ctx, grad_output):
        """
        Backward pass computation.

        Args:
            ctx: Context object with saved tensors
            grad_output: Gradient of the loss w.r.t. the output

        Returns:
            Gradient of the loss w.r.t. the input
        """
        input_tensor, = ctx.saved_tensors  # Retrieve saved input
        grad_input = grad_output.clone()
        # ReLU gradient: 1 if input > 0, else 0
        grad_input[input_tensor < 0] = 0
        return grad_input

# Use the custom function
input = torch.randn(3, 3, requires_grad=True)
output = CustomReLU.apply(input)
output.sum().backward()

print(f"Input:\n{input}")
print(f"Output (custom ReLU):\n{output}")
print(f"Input gradient:\n{input.grad}")

# Compare with built-in ReLU
input_builtin = input.detach().requires_grad_(True)
output_builtin = torch.nn.functional.relu(input_builtin)
output_builtin.sum().backward()

print(f"\nBuilt-in ReLU gradient:\n{input_builtin.grad}")
print(f"Gradients match: {torch.allclose(input.grad, input_builtin.grad)}")
```

Custom autograd functions allow you to define operations with custom forward and backward behavior. This is useful when implementing new layers or operations not available in PyTorch, or when optimizing specific operations for performance. The ctx object is used to pass information from the forward pass to the backward pass.

Hands-On: Tensor Operations

Exercise 1: Basic Tensor Manipulation

```
# Your task: Create a 3x3 tensor of random values,
# then extract a 2x2 sub-tensor from the top-right
# corner and compute its sum.

import torch

# Create 3x3 tensor
x = torch.rand(3, 3)
print(f"Original tensor:\n{x}")

# Extract 2x2 sub-tensor from top-right
# Your code here

# Compute sum
# Your code here

# Expected output:
# Sub-tensor:
# tensor([[x[0,1], x[0,2]],
#         [x[1,1], x[1,2]]])
# Sum: (sum of the 4 values)
```

Exercise 2: Broadcasting

```
# Your task: Create a batch of 10 vectors,
# each with 5 elements. Then normalize each
# vector by its maximum value.

import torch

# Create a batch of vectors (10x5 tensor)
vectors = torch.rand(10, 5)
print(f"Original vectors shape: {vectors.shape}")

# Find maximum value for each vector
# Your code here

# Normalize each vector by its maximum value
# Your code here

# Verify that the maximum value in each row is 1.0
# Your code here

# Expected output:
# Max values shape: torch.Size([10, 1])
# Normalized vectors max: tensor([1., 1., ...])
```

Exercise 3: Reshaping

```
# Your task: Create a 12-element tensor and
# reshape it in three different ways:
# 1. As a 3x4 matrix
# 2. As a 4x3 matrix
# 3. As a 2x2x3 tensor

import torch

# Create a 12-element tensor
x = torch.arange(12)
print(f"Original tensor: {x}")

# Reshape as 3x4 matrix
# Your code here

# Reshape as 4x3 matrix
# Your code here

# Reshape as 2x2x3 tensor
# Your code here

# Expected output:
# 3x4 matrix:
# tensor([[ 0, 1, 2, 3],
# [ 4, 5, 6, 7],
# [ 8, 9, 10, 11]])
# (and similar for the other shapes)
```

Tensor Operations Exercise Solutions

Solution 1: Basic Tensor Manipulation

```
import torch

# Create 3x3 tensor
x = torch.rand(3, 3)
print(f"Original tensor:\n{x}")

# Extract 2x2 sub-tensor from top-right
sub_tensor = x[0:2, 1:3]
print(f"Sub-tensor:\n{sub_tensor}")

# Compute sum
tensor_sum = sub_tensor.sum()
print(f"Sum: {tensor_sum}")

# Alternative approaches:
# Using indexing directly:
# sub_tensor = x[:2, 1:]

# Using advanced indexing:
# row_indices = torch.tensor([0, 1])
# col_indices = torch.tensor([1, 2])
# sub_tensor = x[row_indices[:, None], col_indices]
```

Solution 2: Broadcasting

```
import torch

# Create a batch of vectors (10x5 tensor)
vectors = torch.rand(10, 5)
print(f"Original vectors shape: {vectors.shape}")

# Find maximum value for each vector
max_values, _ = torch.max(vectors, dim=1, keepdim=True)
print(f"Max values shape: {max_values.shape}")

# Normalize each vector by its maximum value
normalized_vectors = vectors / max_values
print(f"Normalized vectors shape: {normalized_vectors.shape}")

# Verify that the maximum value in each row is 1.0
normalized_max, _ = torch.max(normalized_vectors, dim=1)
print(f"Normalized vectors max: {normalized_max}")
print(f"All vectors normalized correctly: {torch.allclose(normalized_max, torch.ones(10))}")

# Note how broadcasting made this operation simple:
# max_values has shape [10, 1] but is automatically
# broadcast to [10, 5] for the division
```

Solution 3: Reshaping

```
import torch

# Create a 12-element tensor
x = torch.arange(12)
print(f"Original tensor: {x}")

# Reshape as 3x4 matrix
matrix_3x4 = x.reshape(3, 4)
print(f"3x4 matrix:\n{matrix_3x4}")

# Reshape as 4x3 matrix
matrix_4x3 = x.reshape(4, 3)
print(f"4x3 matrix:\n{matrix_4x3}")

# Reshape as 2x2x3 tensor
tensor_2x2x3 = x.reshape(2, 2, 3)
print(f"2x2x3 tensor:\n{tensor_2x2x3}")

# Alternative methods:
# Using view() (shares memory with original):
# matrix_3x4 = x.view(3, 4)

# Using -1 to automatically infer one dimension:
# matrix_4x3 = x.reshape(4, -1)
```

Hands-On: Custom Autograd Functions

```
# Exercise: Implement a custom exponential linear unit (ELU) function
# ELU(x) = x if x > 0, alpha * (exp(x) - 1) if x <= 0

import torch
import matplotlib.pyplot as plt

class CustomELU(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input_tensor, alpha=1.0):
        """
        Forward pass for ELU: f(x) = x if x > 0, alpha * (exp(x) - 1) if x <= 0

        Args:
            ctx: Context object for backward
            input_tensor: Input tensor
            alpha: Hyperparameter controlling negative saturation

        Returns:
            ELU output
        """
        # TODO: Implement the forward pass
        # 1. Save any values needed for backward pass using ctx.save_for_backward
        # 2. Calculate and return the ELU output

        # Your implementation here
        pass

    @staticmethod
    def backward(ctx, grad_output):
        """
        Backward pass for ELU

        Args:
            ctx: Context with saved tensors
            grad_output: Gradient of loss w.r.t. output

        Returns:
            Gradient of loss w.r.t. input and None for alpha (no gradient for alpha)
        """
        # TODO: Implement the backward pass
        # 1. Retrieve saved tensors from ctx
        # 2. Calculate the gradient of the ELU function
        # 3. Return the gradient multiplied by grad_output

        # Your implementation here
        pass

# Function to test and plot your implementation
def test_elu():
    x = torch.linspace(-3, 3, 1000, requires_grad=True)

    # Your custom ELU
    # y_custom = CustomELU.apply(x, 1.0)

    # PyTorch's built-in ELU
    y_builtin = torch.nn.functional.elu(x, alpha=1.0)

    # Compare results
    # max_diff = (y_custom - y_builtin).abs().max().item()
    # print(f"Maximum difference: {max_diff}")

    # Test backpropagation
    # y_custom.sum().backward()
    # x_grad_custom = x.grad.clone()

    # x.grad.zero_()
    # y_builtin.sum().backward()
    # x_grad_builtin = x.grad.clone()

    # max_grad_diff = (x_grad_custom - x_grad_builtin).abs().max().item()
    # print(f"Maximum gradient difference: {max_grad_diff}")

    # Plot results
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(x.detach().numpy(), y_builtin.detach().numpy(), label='ELU')
    # plt.plot(x.detach().numpy(), y_custom.detach().numpy(), '--', label='Custom ELU')
    plt.title('ELU Function')
    plt.legend()
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(x.detach().numpy(), x_grad_builtin.numpy(), label='ELU Gradient')
    # plt.plot(x.detach().numpy(), x_grad_custom.numpy(), '--', label='Custom ELU Gradient')
    plt.title('ELU Gradient')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

# Call the test function
# test_elu()
```

Complete the implementation of the `CustomELU` autograd function, which implements the Exponential Linear Unit activation function. The forward and backward passes should match the mathematical definition of ELU. After implementing the function, uncomment the code in the test function to visualize and verify your implementation.

Custom Autograd Function Solution

```
import torch
import matplotlib.pyplot as plt

class CustomELU(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input_tensor, alpha=1.0):
        """
        Forward pass for ELU: f(x) = x if x > 0, alpha * (exp(x) - 1) if x <= 0

        Args:
            ctx: Context object for backward
            input_tensor: Input tensor
            alpha: Hyperparameter controlling negative saturation

        Returns:
            ELU output
        """
        # Save input and alpha for backward pass
        ctx.save_for_backward(input_tensor)
        ctx.alpha = alpha

        # Create output tensor
        output = input_tensor.clone()

        # Apply ELU formula
        mask = output <= 0
        output[mask] = alpha * (torch.exp(output[mask]) - 1)

        return output

    @staticmethod
    def backward(ctx, grad_output):
        """
        Backward pass for ELU

        Args:
            ctx: Context with saved tensors
            grad_output: Gradient of loss w.r.t. output

        Returns:
            Gradient of loss w.r.t. input and None for alpha (no gradient for alpha)
        """
        # Retrieve saved tensors and values
        input_tensor, = ctx.saved_tensors
        alpha = ctx.alpha

        # Initialize gradient as a copy of the output gradient
        grad_input = grad_output.clone()

        # For x <= 0, the gradient is: alpha * exp(x) * grad_output
        # For x > 0, the gradient is: 1 * grad_output
        mask = input_tensor <= 0
        grad_input[mask] = grad_input[mask] * alpha * torch.exp(input_tensor[mask])

        # Return gradient for input and None for alpha (no gradient for alpha)
        return grad_input, None

# Function to test and plot the implementation
def test_elu():
    x = torch.linspace(-3, 3, 1000, requires_grad=True)

    # Your custom ELU
    y_custom = CustomELU.apply(x, 1.0)

    # PyTorch's built-in ELU
    y_builtin = torch.nn.functional.elu(x, alpha=1.0)

    # Compare results
    max_diff = (y_custom - y_builtin).abs().max().item()
    print(f"Maximum difference: {max_diff}")

    # Test backpropagation
    y_custom.sum().backward()
    x_grad_custom = x.grad.clone()

    x.grad.zero_()
    y_builtin.sum().backward()
    x_grad_builtin = x.grad.clone()

    max_grad_diff = (x_grad_custom - x_grad_builtin).abs().max().item()
    print(f"Maximum gradient difference: {max_grad_diff}")

    # Plot results
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(x.detach().numpy(), y_builtin.detach().numpy(), label='ELU')
    plt.plot(x.detach().numpy(), y_custom.detach().numpy(), '--', label='Custom ELU')
    plt.title('ELU Function')
    plt.legend()
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(x.detach().numpy(), x_grad_builtin.numpy(), label='ELU Gradient')
    plt.plot(x.detach().numpy(), x_grad_custom.numpy(), '--', label='Custom ELU Gradient')
    plt.title('ELU Gradient')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

# Call the test function to verify the implementation
# test_elu()
```

This solution implements both the forward and backward passes for the ELU function. The forward pass applies the ELU formula: f(x) = x if x > 0, alpha * (exp(x) - 1) if x <= 0. The backward pass computes the gradient: f'(x) = 1 if x > 0, alpha * exp(x) if x <= 0. The test function compares our implementation with PyTorch's built-in ELU and visualizes both the function and its gradient.