PyTorch Core Notes

Summary & Learning Map

These notes provide a **concept-first, system-level introduction to PyTorch**.
They are designed to teach *how PyTorch works*, not just how to write code that runs.

If you complete this entire sequence, you understand PyTorch at a level where: - You can write models from scratch - You can debug training failures - You are not dependent on high-level abstractions

---

Learning Philosophy

These notes follow three principles:

- **Explicit over implicit**
  You see every step of training. Nothing is hidden.
- **Concepts before APIs**
  Understanding comes before memorization.
- **Systems thinking**
  PyTorch is treated as a pipeline, not a bag of functions.

Learning Map (Concept Flow)

The material is structured to follow the actual lifecycle of deep learning in PyTorch.

1. Introduction

- What PyTorch is
- Why dynamic computation graphs matter
- How PyTorch differs from other frameworks

2. Fundamentals

- Tensors as the core data structure
- Tensor operations and shape reasoning
- Devices and data types

3. Autograd

- Automatic differentiation

- Computation graphs
- Gradient flow and accumulation

4. Models

- `nn.Module`
- Layers and parameters
- What actually learns in a model

5. Training

- The training process
- Loss functions
- Optimizers
- The full training loop

6. Data Pipeline

- Dataset abstraction
- DataLoader batching and iteration

7. Evaluation and Inference

- Training vs evaluation mode
- Disabling gradients
- Correct inference practices

8. Serialization

- Saving and loading models
- `state_dict`
- Reproducibility and persistence

What You Can Do After Completing This

After finishing these notes, you should be able to:

- Build PyTorch models from first principles
- Write correct training and evaluation loops
- Debug shape, gradient, and data issues
- Save, load, and reuse trained models safely
- Read and understand real-world PyTorch codebases

These skills transfer directly to: - Research code - Production systems - Higher-level frameworks built on PyTorch

---

What Is Intentionally Not Covered

These notes do **not** focus on: - High-level trainers or automation frameworks - Framework-specific shortcuts - Task-specific architectures

Those belong **after** fundamentals.

Offline Access

PDF versions of these notes are provided for offline use.
The Markdown files remain the **source of truth**.

Where to Go Next

Recommended next steps: - Vision, NLP, or multimodal PyTorch projects - Distributed and mixed-precision training - Model optimization and deployment - Higher-level abstractions (used intentionally)

Final Note

If PyTorch feels simple after this, that is not because it *is* simple.

It is because you understand it.

Authored by: Sai Harsha Kondaveeti | LinkedIn # What is PyTorch?

PyTorch is a **Python-first deep learning framework** used to build, train, and deploy neural networks. It is designed to feel natural to Python programmers while providing the performance and flexibility required for modern deep learning.

At its core, PyTorch combines:

- A powerful tensor computation library
- Automatic differentiation (autograd)
- Explicit, transparent training loops
- Native GPU acceleration

PyTorch is widely used in research, startups, and production systems because it prioritizes **clarity and control over abstraction**.

Why PyTorch Exists

Before PyTorch, many deep learning frameworks relied on **static computation graphs**. In those systems: - You first defined the entire computation graph - Then compiled it - Then executed it

This made experimentation, debugging, and teaching unnecessarily difficult.

PyTorch was created to solve this problem by using **dynamic computation graphs**.

In PyTorch: - The graph is built **as the code runs** - Normal Python control flow (if, loops, functions) works naturally - Debugging feels like debugging regular Python code

---

Dynamic Computation Graphs

A computation graph represents how data flows through operations.

In PyTorch: - The graph is created during the **forward pass** - Each operation on tensors is recorded - The graph is discarded after the backward pass

This design allows: - Flexible model architectures - Conditional logic inside models - Easy debugging with print statements and breakpoints

Important

- PyTorch does **not** build graphs ahead of time
- The graph is **re-created every forward pass**
- This is why PyTorch feels intuitive and Pythonic

PyTorch vs TensorFlow (Conceptual Comparison)

This is not about which framework is "better". It is about **design philosophy**.

PyTorch Philosophy

- Code-first
- Explicit training loops
- Python control flow
- Transparency over abstraction

TensorFlow Philosophy (Traditional)

- Graph-first
- Higher abstraction
- More framework-managed behavior

Important

- PyTorch makes learning **how deep learning works** easier
- TensorFlow emphasizes large-scale deployment and tooling
- Understanding PyTorch makes it easier to learn any other framework later

---

What PyTorch Is Not

It is important to be precise.

PyTorch is **not**: - An automatic model trainer - A no-code or low-code tool - A replacement for understanding math

PyTorch gives you **control**, not answers.

Important

- PyTorch will not stop you from writing incorrect logic
- It assumes the user understands what they are doing
- This is a feature, not a flaw

How to Think About PyTorch

The correct mental model is:

> PyTorch = Tensors + Gradients + Python

Everything else is built on top of this idea.

If you understand: - How tensors store data - How gradients flow through operations - How Python code defines computation

Then PyTorch becomes predictable and debuggable.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain what PyTorch is in simple terms - Describe what a dynamic computation graph means - Understand why PyTorch is widely used for learning and research - Know what PyTorch does *and does not* provide

The next step is to understand **tensors**, the single most important object in PyTorch.

Tensor Operations: How Data Actually Moves

Tensor operations define how data flows through a model.
Every layer, every loss, every gradient is built on these operations.

Understanding tensor operations is understanding **how computation happens**.

Element-wise Operations

Element-wise operations apply **independently** to each element.

Examples: - Addition - Subtraction - Multiplication - Division

These operations require tensors to: - Have the same shape, or - Be broadcastable

Important

- Element-wise operations do **not** perform matrix math
- Confusing element-wise ops with matrix ops is a classic beginner mistake

---

Matrix Multiplication

Matrix multiplication follows linear algebra rules.

Common operations: - Vector × Matrix - Matrix × Matrix - Batch matrix multiplication

PyTorch provides explicit APIs for this.

Important

- Matrix multiplication is **not** element-wise multiplication
- Shape alignment matters
- One wrong dimension and the operation fails immediately (good)

Broadcasting

Broadcasting allows PyTorch to perform operations on tensors of different shapes by **implicitly expanding dimensions**.

Example scenarios: - Adding a vector to each row of a matrix - Applying bias terms

Broadcasting follows strict rules: - Dimensions must be equal or - One of them must be 1

Important

- Broadcasting is powerful but dangerous
- It can silently produce incorrect results
- Always verify resulting shapes

Reshaping Tensors

Reshaping changes how data is viewed **without changing the data itself**.

Common operations: - `view()` - `reshape()`

Typical use cases: - Flattening tensors before linear layers - Changing batch structure

Important

- Total number of elements must remain the same
- `view()` requires contiguous memory
- Shape mistakes here propagate everywhere

Transposing and Permuting

These operations reorder dimensions.

Use cases: - Switching between channel-first and channel-last formats - Preparing data for specific layers

Important

- Transpose changes how data is interpreted, not stored
- Incorrect dimension ordering breaks models silently
- Always document expected tensor layouts

Reduction Operations

Reduction operations collapse dimensions.

Examples: - Sum - Mean - Max

Used heavily in: - Loss functions - Metrics

Important

- Reductions change tensor rank
- Losing dimensions accidentally can break downstream code
- Always know what shape comes out

In-place vs Out-of-place Operations

In-place operations modify the tensor directly.

Out-of-place operations create new tensors.

Important

- In-place ops can interfere with autograd
- They can corrupt the computation graph
- Avoid them unless memory constraints force your hand

Debugging Tensor Operations

Professional habits: - Print tensor shapes - Check intermediate outputs - Verify assumptions early

Important

- Shape debugging is not a beginner skill, it's a survival skill
- Most deep learning bugs are **not** mathematical, they're structural

---

How to Think About Tensor Operations

The correct mental model:

> Tensor operations define **data flow**, not just computation.

If data flows incorrectly: - Gradients are meaningless - Training becomes unstable - Models appear to "not learn"

What You Should Understand After This Section

By the end of this section, you should be able to: - Distinguish element-wise and matrix operations - Reason about broadcasting safely - Reshape tensors confidently - Debug shape-related issues

The next step is understanding **automatic differentiation (autograd)**.

Tensors: The Core Data Structure in PyTorch

A **tensor** is the fundamental building block of PyTorch.
Every model, every gradient, every operation ultimately works on tensors.

If you understand tensors well, PyTorch stops being confusing.

What is a Tensor?

Conceptually, a tensor is a **generalization of arrays** to higher dimensions.

- Scalar → 0-D tensor
- Vector → 1-D tensor
- Matrix → 2-D tensor
- Images, batches → 3-D / 4-D tensors

Tensors are similar to NumPy arrays but with two critical additions: - They can track gradients - They can live on GPUs

Creating Tensors

Common ways to create tensors:

- `torch.tensor(data)`
- `torch.zeros(shape)`
- `torch.ones(shape)`
- `torch.rand(shape)`

Tensors can be created from: - Python lists - NumPy arrays - Existing tensors

Important

- Tensor creation defines **shape**, **dtype**, and **device**
- Be explicit when teaching beginners. Defaults hide problems.

Tensor Shape

The `shape` of a tensor defines its dimensions.

Examples: - `(10,)` → vector of length 10 - `(3, 4)` → matrix - `(32, 3, 224, 224)` → batch of images

Important

- Shape mismatches are the **most common PyTorch error**
- Always inspect shapes during debugging
- Printing shapes is not amateur behavior. It's professional survival.

Tensor Data Types (dtype)

Common dtypes: - `float32` (default for most ML work) - `float64` - `int64`

Important

- Most neural networks expect **floating-point tensors**
- Loss functions and layers can fail silently with wrong dtypes
- Classification labels are usually `int64`

Tensor Devices (CPU vs GPU)

Tensors live on a **device**: - CPU - GPU (CUDA)

A tensor and a model **must be on the same device** to interact.

Important

- Device mismatches cause runtime errors
- GPU acceleration is explicit, not automatic
- PyTorch will not guess what you want

Tensor vs NumPy Array

Similarities: - Both store numerical data - Both support vectorized operations

Differences: - PyTorch tensors support **autograd** - PyTorch tensors can run on **GPU** - NumPy arrays cannot

Important

- PyTorch tensors can convert to NumPy and back
- Once gradients matter, NumPy is no longer enough

Tensor Mutability and In-place Operations

Some operations modify tensors **in-place**.

Example: - `add_()` modifies the tensor directly

Others create new tensors.

Important

- In-place operations can **break gradient computation**
- Avoid in-place ops unless you know exactly why you need them
- Correctness > micro-optimizations

How to Think About Tensors

The correct mental model:

> A tensor is **data + metadata**

Metadata includes: - Shape - dtype - device - gradient history (if enabled)

Tensors are not "just arrays". They are active participants in computation.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain what a tensor is - Reason about tensor shapes - Understand dtype and device implications - Know why tensors are central to PyTorch

The next step is understanding **how tensors interact through operations**.

Automatic Differentiation (Autograd)

Autograd is the mechanism that allows PyTorch to **automatically compute gradients**.

Without autograd: - Neural networks do not learn - Optimizers have nothing to update - Training is impossible

Autograd is not optional. It is the engine behind learning.

What is Automatic Differentiation?

Automatic differentiation is a technique to compute derivatives **exactly** using the chain rule.

PyTorch does this by: - Tracking operations performed on tensors - Building a computation graph dynamically - Applying the chain rule during backpropagation

This is **not** numerical differentiation and **not** symbolic math.

The Computation Graph

A computation graph is a directed graph where: - Nodes represent operations - Edges represent tensors flowing between operations

In PyTorch: - The graph is built **during the forward pass** - Only tensors with `requires_grad=True` are tracked - The graph exists only temporarily

Important

- PyTorch uses **dynamic computation graphs**
- The graph is rebuilt on every forward pass
- This allows conditional logic and loops inside models

`requires_grad`

Gradients are tracked only for tensors that explicitly request it.

- `requires_grad=True` tells PyTorch to track operations
- Parameters inside `nn.Module` default to this

Important

- Not all tensors need gradients
- Inputs often do not require gradients
- Model parameters almost always do

Forward Pass vs Backward Pass

Forward Pass

- Computes outputs from inputs
- Builds the computation graph
- Stores operation history

Backward Pass

- Triggered by calling `.backward()`
- Traverses the graph in reverse
- Computes gradients using the chain rule

Important

- Forward builds the graph
- Backward **consumes and frees** the graph
- A new forward pass builds a new graph

Calling `.backward()`

The backward pass starts from a **scalar value**, usually the loss.

Why scalar? - Gradients must propagate from a single objective

Calling `.backward()`: - Computes gradients for all leaf tensors - Stores gradients in `.grad`

Important

- Only scalar tensors can call `.backward()` directly
- Gradients are accumulated, not overwritten

Gradient Accumulation

By default, PyTorch **accumulates gradients**.

This means: - Each `.backward()` adds to existing gradients - Gradients must be cleared manually

This design supports: - Gradient accumulation across batches - Advanced training strategies

Important

- Forgetting to clear gradients is a common bug
- Optimizers expect fresh gradients each step

Accessing Gradients

After backpropagation: - Gradients are stored in `tensor.grad`

Only **leaf tensors** store gradients: - Model parameters - User-created tensors with `requires_grad=True`

Important

- Intermediate tensors do not retain gradients by default
- This saves memory and improves performance

Disabling Gradient Tracking

Not all computation requires gradients.

PyTorch provides mechanisms to disable tracking: - During evaluation - During inference - During parameter-free computation

Important

- Disabling gradients improves speed and memory usage
- Forgetting to disable gradients during inference is wasteful

---

Detaching Tensors

Sometimes you want to stop gradient flow.

Detaching: - Creates a new tensor - Shares the same data - Removes gradient history

Important

- Detaching breaks the computation graph intentionally
- This is often used in advanced training loops

Common Autograd Mistakes

Typical errors include: - Using in-place operations on tracked tensors - Forgetting to clear gradients - Expecting gradients where none exist - Mixing NumPy and PyTorch incorrectly

Important

- Autograd is deterministic, not magical
- Most issues come from incorrect assumptions

How to Think About Autograd

The correct mental model:

> Autograd records **operations**, not equations.

If an operation happens: - It is tracked (if gradients are enabled) - It becomes part of the graph

If it does not happen: - No gradient exists - No learning occurs

---

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain how PyTorch computes gradients - Describe what a computation graph is - Understand why gradients accumulate - Debug basic autograd-related issues

The next step is understanding **how models are structured using `nn.Module`**.

Layers and Parameters: What Actually Learns

In PyTorch, **layers are objects** and **parameters are data**.
Understanding the difference is critical to understanding how models learn.

If this section is clear, optimizers and training loops become obvious.

What is a Layer?

A layer is a reusable computation unit.

Examples: - Linear layers - Convolution layers - Activation functions - Normalization layers

In PyTorch, **layers are instances of `nn.Module`**.

Important

- Layers define *how* data is transformed
- Layers may or may not have parameters
- Not all layers learn

Parameters

Parameters are **trainable tensors**.

They represent: - Weights - Biases - Any value updated by gradient descent

Characteristics: - Stored as `nn.Parameter` - Have `requires_grad=True` - Updated by optimizers

Important

- Only parameters are optimized
- If something is not a parameter, it will not learn

- Parameters are discovered automatically by the optimizer

How Parameters Are Registered

Parameters are registered when: - They are assigned to `self` inside `__init__` - They are instances of `nn.Parameter` or inside a layer

Example conceptually: `- self.weight = nn.Parameter(...)` → registered - Local variables → ignored

Important

- Registration is based on object ownership
- Scope matters more than names

Layers With and Without Parameters

Layers With Parameters

Examples: - Linear - Convolution - Embedding

These layers: - Contain trainable tensors - Change during training

Layers Without Parameters

Examples: - ReLU - Sigmoid - Dropout

These layers: - Perform fixed operations - Do not learn values

Important

- Not all layers contribute parameters
- All layers contribute computation

Sharing Parameters

The same layer instance can be reused multiple times.

This results in: - Shared parameters - Coupled learning behavior

Important

- Parameter sharing is intentional and powerful
- Used in recurrent and tied-weight architectures

Inspecting Parameters

PyTorch allows inspection of: - Parameter names - Shapes - Gradient status

This transparency is deliberate.

Important

- You should always know what is being trained
- Silent parameters are a red flag

Freezing Parameters

Parameters can be frozen by disabling gradients.

Use cases: - Transfer learning - Fine-tuning - Feature extraction

Important

- Frozen parameters still participate in forward passes
- They simply stop learning

Common Mistakes with Layers and Parameters

Typical errors: - Creating layers inside `forward` - Expecting activations to learn - Forgetting that optimizers only see parameters - Accidentally freezing everything

Important

- Learning happens only where gradients flow
- Structure mistakes lead to silent failures

How to Think About Layers and Parameters

The correct mental model:

> Layers define computation.
> Parameters define what can change.

If parameters are wrong: - Learning fails - Optimization is meaningless

---

What You Should Understand After This Section

By the end of this section, you should be able to: - Distinguish layers from parameters - Know what actually learns in a model - Inspect and reason about model parameters - Avoid silent training bugs

The next step is understanding **loss functions**, which turn predictions into learning signals.

Models as Code: Understanding `nn.Module`

In PyTorch, a neural network model is **just a Python class**.

There is no separate model definition language.
There is no hidden graph builder.
There is only Python code.

This design choice is one of PyTorch's biggest strengths.

What is `nn.Module`?

`nn.Module` is the **base class** for all neural network models in PyTorch.

When you define a model, you: - Subclass `nn.Module` - Define layers in `__init__` - Define computation in `forward`

Everything else is handled automatically.

Basic Structure of a Model

A PyTorch model has two key parts:

1. `__init__()` – defines the structure

2. `forward()` – defines the computation

Conceptually: - `__init__` answers: *What components does the model have?* - `forward` answers: *How does data flow through them?*

Important

- `forward()` defines **math**, not training
- No gradients are computed here explicitly
- No optimizer logic belongs here

Layers as Objects

Layers such as: - Linear - Convolution - Activation functions

are themselves instances of `nn.Module`.

When you assign them as attributes:

```
self.layer = nn.Linear(...)
```

PyTorch automatically: - Registers parameters - Tracks them for optimization - Includes them in `model.parameters()`

Important

- Only attributes assigned to `self` are registered
- Temporary tensors inside `forward` are not parameters

Parameters and Buffers

Parameters

- Trainable values
- Automatically updated by optimizers
- Have `requires_grad=True`

Buffers

- Non-trainable tensors
- Still part of the model's state
- Used for things like running statistics

Important

- Parameters learn
- Buffers support learning
- Both are saved with the model

The `forward()` Method

`forward()` defines how input tensors are transformed into outputs.

Key points: - Accepts tensors as input - Returns tensors as output - Can include any valid Python control flow

Important

- You never call `forward()` directly
- Calling the model object triggers `forward()` internally
- This allows PyTorch to manage hooks and autograd correctly

Sequential vs Custom Models

Sequential Models

Useful for simple, linear architectures. - Easy to read - Limited flexibility

Custom Models

Required for: - Multiple inputs - Conditional logic - Complex architectures

Important

- `nn.Sequential` is a convenience, not a replacement
- Serious models almost always subclass `nn.Module`

Model Introspection

PyTorch allows you to inspect models easily: - Parameters - Layer structure - State

This transparency is intentional.

Important

- You can see exactly what your model contains
- There is no hidden state or opaque behavior

Common Mistakes with `nn.Module`

Typical errors: - Defining layers inside `forward` - Forgetting to call `super().__init__()` - Mixing training logic into the model

Important

- Models should be pure computation
- Training logic belongs outside the model

How to Think About Models in PyTorch

The correct mental model:

> A PyTorch model is a callable object that transforms tensors.

Nothing more. Nothing less.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain what `nn.Module` is - Define a custom model cleanly - Understand how parameters are registered - Separate model logic from training logic

The next step is understanding layers and parameters in more detail.

Loss Functions: Turning Error into a Learning Signal

A loss function converts model predictions into a **single numerical value** that represents how wrong the model is.

Without a loss function: - There is no objective - There are no gradients - Learning cannot happen

Loss functions are not optional. They define *what learning means*.

What is a Loss Function?

A loss function measures the difference between: - Model predictions - Ground-truth targets

It outputs a **scalar value** that: - Is minimized during training - Drives gradient computation - Guides parameter updates

---

Why Loss Must Be a Scalar

Gradients must flow from a **single objective**.

Important

- Backpropagation requires a scalar starting point
- Vector or matrix losses must be reduced
- Reduction is part of loss design, not an afterthought

Common Types of Loss Functions

Loss functions are chosen based on the problem type.

Regression Losses

Used when predicting continuous values.

Examples: - Mean Squared Error (MSE) - Mean Absolute Error (MAE)

Characteristics: - Penalize distance between prediction and target - Sensitive to scale and outliers

---

Classification Losses

Used when predicting class labels.

Examples: - Cross Entropy Loss

Characteristics: - Compare predicted class probabilities with true labels - Strongly penalize confident wrong predictions

---

Cross Entropy Loss (Critical Concept)

Cross Entropy Loss is commonly misunderstood.

Important

- `CrossEntropyLoss` **includes Softmax internally**
- Model outputs should be **raw scores (logits)**
- Applying Softmax manually before this loss is incorrect

This design improves numerical stability.

Reduction Modes

Loss functions reduce multiple errors into a single value.

Common reductions: - `mean` (default) - `sum`

Important

- Reduction affects gradient magnitude
- Changing reduction can change training dynamics
- Be consistent across experiments

Loss and Gradients

The loss function: - Connects predictions to targets - Defines how errors propagate backward - Shapes the gradient landscape

Important

- Different losses produce different gradients
- Same model + different loss = different learning behavior

Choosing the Right Loss Function

Choosing a loss function is a **design decision**, not a syntax choice.

Consider: - Output type - Target format - Sensitivity to errors - Stability during training

Important

- Wrong loss = model that never converges
- Loss choice matters as much as model architecture

Common Mistakes with Loss Functions

Typical errors: - Using classification loss for regression - Applying Softmax before Cross Entropy - Mismatched target shapes or types - Ignoring reduction effects

Important

- Loss bugs often look like "model not learning"
- Always validate loss behavior early

How to Think About Loss Functions

The correct mental model:

> A loss function defines **what the model is trying to optimize**.

If the loss is wrong: - Gradients are wrong - Updates are wrong - Learning fails silently

---

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain what a loss function does - Choose appropriate losses for tasks - Understand why loss must be scalar - Avoid common loss-related mistakes

The next step is understanding **optimizers**, which turn gradients into parameter updates.

Optimizers: Turning Gradients into Learning

Optimizers are responsible for **updating model parameters** using gradients computed during backpropagation.

Without optimizers: - Gradients exist but do nothing - Parameters never change - Training does not occur

Optimizers define *how* learning happens.

What is an Optimizer?

An optimizer is an algorithm that: - Reads gradients from parameters - Updates parameter values - Controls learning speed and stability

Optimizers operate **only on parameters**, never on tensors or outputs.

The Optimizer Workflow

Every optimizer step follows the same logical sequence:

1. Gradients are computed via backpropagation
2. Optimizer reads gradients from parameters
3. Parameters are updated
4. Gradients are cleared

Important

- Optimizers do not compute gradients
- They only consume gradients
- If gradients are wrong, updates are wrong

Learning Rate

The learning rate controls **how large each update step is**.

Important

- Too large → training becomes unstable
- Too small → training becomes slow or stuck
- Learning rate matters more than optimizer choice

Common Optimizers

Stochastic Gradient Descent (SGD)

SGD updates parameters using the raw gradient.

Characteristics: - Simple - Predictable - Requires careful learning rate tuning

Often used with momentum to improve convergence.

Adam Optimizer

Adam adapts learning rates per parameter.

Characteristics: - Faster convergence in many cases - Less sensitive to learning rate choice - Common default in modern deep learning

Important

- Adam is convenient, not magical
- Poor data or loss choices still fail

Gradient Clearing (`zero_grad`)

Gradients in PyTorch **accumulate by default**.

Before each update: - Gradients must be reset

Important

- Forgetting to clear gradients is a common bug
- Accumulated gradients change training behavior
- Clearing gradients is an explicit step by design

Optimizers and Parameters

Optimizers are initialized with: - Model parameters - Hyperparameters (learning rate, momentum, etc.)

Important

- Optimizers only update parameters they are given
- Frozen parameters will not change
- New parameters require optimizer re-initialization

When Optimizers Run

Optimizers run: - After the backward pass - Once per batch or accumulation step

Important

- Calling optimizer step without gradients does nothing
- Calling backward without optimizer step computes gradients only

---

Common Optimizer Mistakes

Typical errors: - Forgetting `zero_grad` - Updating before backpropagation - Expecting optimizers to fix poor losses or data - Reusing optimizers incorrectly after model changes

Important

- Optimizers do not make models smart
- They only apply math consistently

How to Think About Optimizers

The correct mental model:

Optimizers are **parameter update rules**, not learning logic.

They do not decide *what* to learn.
They only decide *how* to update.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain what optimizers do - Understand the role of learning rate - Distinguish between SGD and Adam - Debug common optimizer-related issues

The next step is understanding **the full training loop**, where everything comes together.

The Training Loop: Where Everything Comes Together

The training loop is where **models actually learn**.

PyTorch does not hide this loop.
You write it yourself.
This is a feature, not a limitation.

If you understand the training loop, you understand PyTorch.

What is a Training Loop?

A training loop is the repeated process of: - Making predictions - Measuring error - Computing gradients - Updating parameters

This loop runs: - Over batches - Over epochs - Until learning stabilizes or stops

---

The Canonical Training Steps

Every correct training loop follows this exact order:

1. Zero existing gradients
2. Forward pass through the model
3. Compute the loss
4. Backward pass (compute gradients)
5. Optimizer step (update parameters)

Important

- The order is not optional
- Changing the order breaks training
- PyTorch will not warn you if you get it wrong

Zeroing Gradients

Before each iteration: - Existing gradients must be cleared

Important

- Gradients accumulate by default
- Forgetting this causes exploding updates
- This is an intentional PyTorch design choice

Forward Pass

During the forward pass: - Data flows through the model - Predictions are produced - The computation graph is built

Important

- No parameters are updated here
- Forward pass only computes values

Loss Computation

The loss: - Compares predictions with targets - Produces a scalar value - Anchors the backward pass

Important

- Loss must be computed every iteration
- Stale loss = stale gradients

Backward Pass

The backward pass: - Starts from the loss - Computes gradients using the chain rule - Stores gradients in parameters

Important

- Backward consumes the computation graph
- A new forward pass builds a new graph

Optimizer Step

The optimizer step: - Reads gradients - Updates parameters - Applies the learning rule

Important

- No gradients = no update
- Optimizer step does not compute gradients

Iterations and Epochs

- **Iteration**: one batch update
- **Epoch**: one full pass over the dataset

Important

- Learning happens at the iteration level
- Epochs are a bookkeeping concept

Training vs Evaluation Mode

Models behave differently during training and evaluation.

Important

- Training mode enables learning behavior
- Evaluation mode disables it
- Forgetting to switch modes causes incorrect results

Common Training Loop Mistakes

Typical errors: - Wrong operation order - Forgetting gradient reset - Updating in evaluation mode - Ignoring batch size effects

Important

- Most "model not learning" issues live here
- Debug the loop before debugging the model

How to Think About the Training Loop

The correct mental model:

> The training loop is a **controlled gradient feedback system**.

Break the loop, and learning stops.

What You Should Understand After This Section

By the end of this section, you should be able to: - Write a training loop from scratch - Explain each step's purpose - Debug common training failures - Understand why PyTorch exposes this logic

The next step is understanding **data pipelines**, which feed data into this loop.

The Training Process: How Models Learn

Training is the process of **adjusting model parameters** so that predictions improve over time.

In PyTorch, training is **explicit**. Nothing is hidden.
This is intentional and essential for understanding.

What Does "Training" Mean?

Training a model means: - Making a prediction - Measuring how wrong it is - Computing gradients - Updating parameters - Repeating this process

This cycle is called **optimization**.

The Core Training Components

Training always involves four components:

1. **Model** – produces predictions

2. **Loss Function** – measures error

3. **Optimizer** – updates parameters

4. **Data** – provides learning signal

If any one of these is wrong, training fails.

High-Level Training Flow

The training process follows a strict sequence:

1. Input data is passed through the model (forward pass)
2. Loss is computed from predictions and targets
3. Gradients are computed via backpropagation
4. Optimizer updates model parameters
5. Gradients are cleared
6. Process repeats for many iterations

This loop is the **heart of deep learning**.

Forward Pass

The forward pass: - Moves data through layers - Produces predictions - Builds the computation graph

Important

- No learning happens here
- Forward pass only computes values

Backward Pass

The backward pass: - Starts from the loss - Computes gradients using the chain rule - Stores gradients in parameters

Important

- Gradients tell parameters *how to change*
- No optimizer update happens yet

Parameter Update

The optimizer: - Reads gradients - Updates parameters according to an algorithm - Controls learning speed via learning rate

Important

- Optimizers only update parameters
- If something does not have gradients, it will not change

Iterative Nature of Training

Training is **iterative**, not instant.

- One pass is not enough
- Learning emerges over many updates
- Noise and instability are normal early on

Important

- Training dynamics matter more than single outputs
- Patience and monitoring are required

How to Think About Training

The correct mental model:

> Training is a controlled feedback loop.

Predictions → Error → Gradients → Updates → Better predictions

Break any link, and learning stops.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain what training actually means - Describe the full training cycle - Understand where loss functions fit in - See why PyTorch exposes the training loop

The next step is understanding **loss functions**, which convert errors into learning signals.

DataLoader: Feeding Data to the Model

A **DataLoader** is responsible for delivering data from a Dataset to the training loop **efficiently and safely**.

If Dataset defines *what* the data is,
DataLoader defines *how* the data is consumed.

What is a DataLoader?

A DataLoader: - Wraps a Dataset - Iterates over it - Produces batches of data

It handles: - Batching - Shuffling - Parallel data loading

Dataset vs DataLoader (Hard Boundary)

- **Dataset** → defines individual samples

- **DataLoader** → controls iteration and batching

Important

- Dataset returns **one sample**
- DataLoader returns **many samples at once**
- Do not mix these responsibilities

Batching

Batching groups multiple samples together.

Why batching matters: - Improves computational efficiency - Stabilizes gradient updates - Matches how GPUs work

Important

- Batch size affects memory usage
- Larger batches are not always better
- Batch size changes training dynamics

Shuffling

Shuffling randomizes the order of samples.

Why it matters: - Prevents learning order-specific patterns - Improves generalization - Reduces bias from data ordering

Important

- Shuffling should be enabled during training
- Shuffling is usually disabled during evaluation

Iteration Over Data

A DataLoader: - Is iterable - Yields batches one at a time - Stops automatically at the end of the dataset

Important

- One full pass over the DataLoader equals one epoch
- Iteration order is controlled by the DataLoader, not the Dataset

---

Parallel Data Loading

DataLoader can load data using multiple workers.

Benefits: - Faster data throughput - Reduced training bottlenecks

Important

- More workers   always faster
- Incorrect worker usage can cause bugs
- Debug with a single worker first

What a DataLoader Returns

Each iteration typically returns: - A batch of input tensors - A batch of target tensors

The exact structure depends on the Dataset output.

Important

- Training loops must match DataLoader output format
- Mismatches cause runtime errors

Common DataLoader Mistakes

Typical errors: - Using batch size too large for memory - Forgetting to shuffle training data - Overusing workers without testing - Assuming DataLoader modifies data

Important

- DataLoader controls flow, not content
- Data correctness is still the Dataset's job

How to Think About DataLoaders

The correct mental model:

> A DataLoader is a **batch-producing iterator**.

It does not care what the data means.
It only cares about delivering it efficiently.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain the role of a DataLoader - Distinguish it clearly from a Dataset - Choose appropriate batch sizes - Debug data feeding issues

This completes the **core PyTorch data pipeline**.

Dataset: Defining Where Data Comes From

A **Dataset** represents the source of data used during training and evaluation.

In PyTorch, data handling is **explicit and modular**.
You define *what* the data is.
PyTorch decides *when* and *how* to fetch it.

What is a Dataset?

A Dataset is an object that: - Knows how many data samples exist - Knows how to fetch a single data sample

It does **not** handle batching, shuffling, or parallelism.

Those responsibilities belong elsewhere.

The Dataset Interface

A custom Dataset must implement two methods:

- `__len__()` → returns dataset size

- `__getitem__(index)` → returns one data sample

This design allows PyTorch to: - Index data lazily - Load only what is needed - Scale to large datasets

Important

- Data is loaded **on demand**
- Nothing is loaded all at once unless you force it

What a Dataset Returns

Each call to `__getitem__` typically returns: - Input data (features) - Target data (labels)

These are usually returned as tensors.

Important

- Dataset output format defines what the training loop receives
- Inconsistent returns cause downstream failures
- Be strict and predictable

Lazy Loading

Datasets are **lazy** by default.

This means: - Data is not loaded into memory upfront - Samples are fetched only when requested - Large datasets become manageable

Important

- Lazy loading is critical for real-world datasets
- Memory errors usually indicate poor dataset design

Dataset vs DataLoader (Clear Separation)

- **Dataset** → defines the data
- **DataLoader** → feeds the data

A Dataset: - Knows nothing about batches - Knows nothing about shuffling - Knows nothing about parallel workers

Important

- Mixing responsibilities leads to rigid designs
- Clean separation improves scalability

Common Dataset Use Cases

Datasets are used to: - Load files from disk - Read structured data (CSV, JSON) - Apply preprocessing - Map raw data to tensors

Important

- Datasets are the right place for data-specific logic
- Training logic does not belong here

Common Dataset Mistakes

Typical errors: - Loading all data into memory unnecessarily - Performing batching inside the Dataset - Returning inconsistent shapes or types - Mixing preprocessing with training logic

Important

- Dataset bugs propagate silently
- Always validate dataset outputs early

How to Think About Datasets

The correct mental model:

> A Dataset is a **dictionary-like interface** to your data.

Given an index, it returns one clean, usable sample.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain what a Dataset is - Implement a custom Dataset correctly - Understand lazy data loading - Separate data definition from data feeding

The next step is understanding **DataLoaders**, which turn datasets into batches.

Evaluation and Inference: Using Models Correctly

Training a model is only half the job.
Using it **correctly** during evaluation and inference is just as important.

Many models appear to "fail" simply because evaluation is done wrong.

Training vs Evaluation

Models behave differently depending on their mode.

PyTorch provides two explicit modes: - Training mode - Evaluation mode

These modes control how certain layers behave.

Training Mode

Training mode is enabled by default.

In this mode: - Gradients are tracked - Layers behave in learning mode - Stochastic behavior is allowed

Important

- Training mode is required for learning
- This mode should be active only during training

Evaluation Mode

Evaluation mode is enabled explicitly.

In this mode: - Model parameters are frozen - Certain layers change behavior - Predictions become deterministic

Important

- Evaluation mode is mandatory for validation and testing
- Forgetting this causes incorrect metrics

Layers Affected by Mode Switching

Some layers behave differently depending on the mode.

Examples: - Dropout - Batch Normalization

Important

- These layers behave correctly **only** if mode is set properly
- This is a common source of silent bugs

Inference

Inference is the process of: - Passing new data through the model - Producing predictions - Not updating parameters

Inference usually happens: - After training - In production - During deployment

---

Disabling Gradient Computation

During evaluation and inference: - Gradients are not needed - Tracking them wastes memory and time

Important

- Gradient tracking should be disabled explicitly
- This improves speed and reduces memory usage

Evaluation Loop

An evaluation loop is similar to a training loop, but: - No backward pass - No optimizer step - No parameter updates

Important

- Evaluation should never modify model state
- Metrics should reflect true model performance

Common Evaluation Mistakes

Typical errors: - Forgetting to switch to evaluation mode - Tracking gradients unnecessarily - Mixing training and evaluation logic - Using training metrics as final results

Important

- Evaluation bugs invalidate results
- Always verify evaluation setup first

How to Think About Evaluation and Inference

The correct mental model:

> Training teaches the model.
> Evaluation measures the model.
> Inference uses the model.

Confusing these roles leads to incorrect conclusions.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain the difference between training and evaluation - Understand why mode switching matters - Run inference safely and efficiently - Avoid common evaluation mistakes

This completes the **PyTorch fundamentals lifecycle** from data to deployment-ready usage.

Saving and Loading Models: Preserving What Was Learned

Training a model without saving it is wasted effort.
Saving models correctly is what makes experiments reproducible and deployment possible.

PyTorch gives you **full control** here. With that control comes responsibility.

What Does It Mean to Save a Model?

Saving a model means preserving: - Learned parameters - Model structure (indirectly) - Training progress (optionally)

PyTorch separates **model definition** from **model state**.

This separation is intentional.

`state_dict`: The Core Concept

A `state_dict` is a Python dictionary that maps: - Parameter names → tensors - Buffer names → tensors

It contains: - Weights - Biases - Running statistics

Important

- `state_dict` does **not** store the model class
- It stores only the model's learned state

Saving a Model (Correct Way)

The recommended approach is to save: - The model's `state_dict`

Why? - Lightweight - Version-safe - Explicit

Important

- Always save the state, not the entire model object
- This avoids hidden dependencies and breakage

Loading a Model (Correct Way)

Loading involves two steps: 1. Recreate the model architecture 2. Load the saved `state_dict` into it

Important

- Model architecture must match the saved state
- PyTorch will tell you if parameters do not align

Why Not Save the Entire Model?

Saving the entire model object: - Ties the file to exact code structure - Breaks easily across environments - Is fragile for long-term use

Important

- Full-model saving is convenient, not robust
- Use it only for quick experiments, not production

Saving Optimizers and Training State

For resuming training, you may also save: - Optimizer state - Epoch number - Training metrics

Important

- Optimizer state is required to resume training exactly
- This matters for long-running or expensive jobs

CPU vs GPU When Loading

Models can be saved on one device and loaded on another.

Important

- Device mapping must be handled explicitly
- This is common when training on GPU and deploying on CPU

Common Saving and Loading Mistakes

Typical errors: - Saving full model objects blindly - Forgetting to save optimizer state - Loading mismatched architectures - Assuming device compatibility

Important

- Most loading bugs are design mistakes, not PyTorch bugs

How to Think About Model Persistence

The correct mental model:

> Model code defines *structure*.
> `state_dict` defines *knowledge*.

Keep them separate, and your workflow stays clean.

What You Should Understand After This Section

By the end of this section, you should be able to: - Explain what `state_dict` is - Save and load models safely - Resume training correctly - Avoid common persistence pitfalls

This completes the **end-to-end PyTorch fundamentals**, from data to deployment-ready models.