IMEN574/IDSC574 Programming for Data Science

16. **PyTorch**

Spring 2026

Instructor: **Sangwoo Mo**

Lecture Outline

01

Autograd & Framework Evolution

How automatic differentiation works, static vs dynamic graphs

02

PyTorch Core

Tensors, autograd API, nn.Module, loss, optimizer, data loading

03

The Training Pipeline

Full training loop, GPU, checkpoints, debugging

01

Autograd & Framework Evolution

The engine behind every deep learning framework

The Core Problem

We need a robot that does calculus for us

Gradient descent needs $\partial L/\partial \theta$ for every parameter θ.

Tiny MLP (784→256→10): ~203K parameters → 203K partial derivatives per step

GPT-3: 175 billion parameters → 175B partial derivatives per step

Writing these by hand? Absolutely impossible. We need automatic differentiation.

✗ Numerical

Finite differences:
$\partial f/\partial x \approx (f(x+\varepsilon) - f(x-\varepsilon)) / 2\varepsilon$
Approximate, slow, numerically unstable. Only for testing.

△ Symbolic

Like Mathematica: manipulate the formula symbolically.
Exact, but expressions explode in size. Doesn't scale.

✓ Automatic (AD)

Break computation into simple steps, apply chain rule at each.
Exact, efficient, scales to billions of params. All DL uses this.

AD is not numerical and not symbolic. It computes exact derivatives by recording a trace of operations and applying the chain rule.

The Key Idea: Computation Graphs

Every complex function is just simple steps chained together

Think of it like a recipe

No matter how complex a neural network is, every computation is a sequence of **simple operations**: add, multiply, power, exp, etc.

A computation graph records each step:

- each node = one simple operation
- edges = data flow between operations

Example: loss = (w·x + b)²

```
x=2, w=3, b=1

Step 1:  z = w × x = 3 × 2 = 6
Step 2:  y = z + b = 6 + 1 = 7
Step 3:  loss = y² = 7² = 49

3 simple ops. Each has a known
derivative. That's all we need!
```

Chain Rule Refresher

If loss = f(g(h(x))), then ∂loss/∂x = f'(g(h(x))) × g'(h(x)) × h'(x)

Each factor is the derivative of one simple step — just multiply them all together.

Framework Evolution

How the idea of "automatic gradients" evolved into modern DL frameworks

TensorFlow 1.x

2015–2019

Define computation graph symbolically, compile it, run in tf.Session. Powerful but painful — debugging meant deciphering compiled graph errors. Control flow required special ops (tf.cond, tf.while_loop).

Static graph

PyTorch

2017–now

Define-by-run. Write normal Python, get gradients automatically. Use if/for/print naturally. Debugging with pdb just works. Became the research standard practically overnight.

Dynamic graph

JAX

2018–now

Google's functional rethink: pure functions + composable transforms (grad, jit, vmap). NumPy-compatible API compiled to XLA for GPU/TPU. Popular in research labs (DeepMind).

Functional

Static vs Dynamic Graphs

The key design decision that separates frameworks

Static Graph (TF 1.x, Theano)

```
# 1. Define graph (no computation yet!)
x = tf.placeholder(tf.float32)
y = tf.matmul(W, x) + b
loss = tf.reduce_mean((y - label)**2)
train = tf.train.AdamOptimizer().minimize(loss)

# 2. Run in session (now it computes)
with tf.Session() as sess:
    sess.run(train, feed_dict={x: data})
```

Dynamic Graph (PyTorch)

```
# Just write Python -- graph builds on the fly
y = W @ x + b
loss = ((y - label)**2).mean()
loss.backward()       # gradients computed now
optimizer.step()      # weights updated now

# Use if/for/print normally!
if loss.item() < threshold:
    print('Converged!')
```

Debugging	Hard (errors in compiled graph)	Easy (normal Python debugger)
Control flow	tf.cond, tf.while_loop	Native if/for/while
Performance	Optimized at compile time	Eager, but torch.compile closes gap
Use case	Production, mobile deploy	Research, prototyping, most new work

Autograd: Recording the Forward Pass

As you compute, PyTorch secretly builds a graph behind the scenes

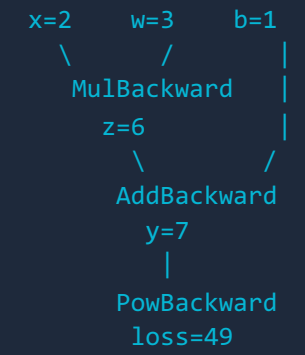
Python — every line secretly records itself

```python
import torch

x = torch.tensor(2.0, requires_grad=True)
w = torch.tensor(3.0, requires_grad=True)
b = torch.tensor(1.0, requires_grad=True)

# Each op creates a node in the graph
z = w * x       # z=6  (records MulBackward)
y = z + b       # y=7  (records AddBackward)
loss = y ** 2   # loss=49 (records PowBackward)
```

Graph built automatically:



requires_grad=True Tells PyTorch: "please track everything that happens to this tensor."

grad_fn Every result tensor has a .grad_fn pointing back to the operation that created it.

Autograd: Walking Backward

One call to .backward() → all gradients, via chain rule

One line does all the work

```
loss.backward()  # triggers reverse-mode AD

# Gradients now stored on leaf tensors:
print(x.grad)  # tensor(42.)  <-- dloss/dx
print(w.grad)  # tensor(28.)  <-- dloss/dw
print(b.grad)  # tensor(14.)  <-- dloss/db
```

Let's verify by hand:

loss = (wx+b)² = (3·2+1)² = 49

∂loss/∂x = 2(wx+b)·w = 2·7·3 = 42 ✓

∂loss/∂w = 2(wx+b)·x = 2·7·2 = 28 ✓

∂loss/∂b = 2(wx+b)·1 = 2·7·1 = 14 ✓

How backward() walks the graph — step by step

Start at loss	loss = 49	**∂loss/∂loss = 1**	*Always start with 1*
PowBackward	loss = y²	**∂loss/∂y = 2y = 2×7 = 14**	*d(x²)/dx = 2x*
AddBackward	y = z + b	**∂loss/∂z = 14, ∂loss/∂b = 14**	*d(a+b) passes grad through*
MulBackward	z = w×x	**∂loss/∂w = 28, ∂loss/∂x = 42**	*d(ab): swap and multiply*

Forward Mode AD

Carry the derivative alongside the value, from inputs to outputs

Idea: Pick one input variable (say a). At every step, compute both the value and its derivative w.r.t. a simultaneously. The derivative "rides along" with the value through each operation.

Example: $f(a, b) = a \cdot b + a^2$ at $a=2, b=3$. Compute $\partial f/\partial a$

Step	Value (forward)	Derivative w.r.t. a (carried forward)
Seed inputs	$a = 2,\ \ b = 3$	$da = 1$ (seed for a), $db = 0$ (not tracking b)
$v_1 = a \times b$	$v_1 = 2 \times 3 = 6$	$dv_1 = da \times b + a \times db = 1 \times 3 + 2 \times 0 = 3$
$v_2 = a^2$	$v_2 = 2^2 = 4$	$dv_2 = 2a \times da = 2 \times 2 \times 1 = 4$
$f = v_1 + v_2$	$f = 6 + 4 = 10$	$df = dv_1 + dv_2 = 3 + 4 = 7$

Result: $\partial f/\partial a = 7$ (verify: $b + 2a = 3+4 = 7$ ✓)

Limitation: one pass = one input's gradient.
Want $\partial f/\partial b$ too? Run the whole thing again.

Reverse Mode AD (Backpropagation)

Compute ALL gradients in one backward pass

Idea: First do the forward pass normally. Then walk backward from the output, asking at each node: "how much did you affect the output?" Apply the chain rule to propagate the gradient backward to all inputs.

Same example: $f(a, b) = a \cdot b + a^2$ at $a=2, b=3$. Now get BOTH $\partial f/\partial a$ AND $\partial f/\partial b$ in one pass

Forward pass (just compute values):

$v_1 = a \times b = 6 \quad v_2 = a^2 = 4 \quad f = v_1 + v_2 = 10$

Backward pass (propagate gradients):

$\partial f/\partial f = 1$

$\partial f/\partial v_1 = 1, \quad \partial f/\partial v_2 = 1$

$\partial f/\partial a = 1 \times b + 1 \times 2a = 3 + 4 = 7 \quad \partial f/\partial b = 1 \times a = 2$

Forward Mode

- Picks ONE input, gets its gradient
- n inputs → n passes needed
- Cost scales with # of inputs
- Good for: few inputs, many outputs

Reverse Mode

- Gets ALL input gradients at once
- Always 1 backward pass
- Cost scales with # of outputs
- Good for: many inputs, few outputs

Why Deep Learning Uses Reverse Mode

The shape of the problem makes reverse mode the only practical choice

A neural network loss function has this shape:

$f : \mathbb{R}^n \to \mathbb{R}^1$ where n = millions to billions of parameters, output = one scalar loss

Forward Mode

Need n passes for n parameters

GPT-3: 175,000,000,000 passes

Completely infeasible

Reverse Mode (Backprop)

Need 1 pass regardless of n

GPT-3: 1 backward pass

All 175B gradients at once

Backpropagation IS reverse-mode AD. The name "backprop" predates the general AD framework, but they're the same algorithm.

JAX: A Different Philosophy

What if we rethought DL frameworks from scratch using functional programming?

PyTorch's limitation: gradients are coupled to tensors (requires_grad, .backward()). JAX's answer: make grad, jit, and vmap composable function transformations.

```python
import jax
import jax.numpy as jnp

# Pure function (no classes, no state)
def loss_fn(params, x, y):
    pred = jnp.dot(x, params['w']) + params['b']
    return jnp.mean((pred - y) ** 2)

# Composable transforms!
grad_fn = jax.grad(loss_fn)        # gradient
fast_grad = jax.jit(grad_fn)       # + JIT compile
batch_grad = jax.vmap(grad_fn)     # + auto-vectorize

grads = fast_grad(params, x, y)    # fast!
```

`jax.grad(f)`

Reverse-mode AD. Returns a new function that computes gradients.

`jax.jit(f)`

XLA-compile for GPU/TPU speed. Like torch.compile but first-class.

`jax.vmap(f)`

Auto-batching. Write for one sample, vmap makes it work on batches.

PyTorch: best for prototyping, debugging, and most research. JAX: best for large-scale training, TPUs, and composable transforms.

02

PyTorch Core

Tensors, autograd, nn.Module, and everything you need to train a model

Tensors: The Fundamental Data Type

Multi-dimensional arrays with GPU support and automatic differentiation

```python
import torch

# Creation
a = torch.zeros(3, 4)                # 3x4 zeros
b = torch.randn(3, 4)                # normal distribution
c = torch.tensor([1.0, 2.0, 3.0])    # from Python list
d = torch.arange(0, 10, 2)           # [0, 2, 4, 6, 8]
e = torch.linspace(0, 1, 5)          # [0, 0.25, 0.5, 0.75, 1]

# Key properties
print(b.shape)    # torch.Size([3, 4])
print(b.dtype)    # torch.float32
print(b.device)   # cpu
```

dtype

float32 (default), float16, bfloat16,
int64, bool

device

cpu or cuda:0, cuda:1, ...
All operations must be on same device

NumPy interop

torch.from_numpy(arr)
tensor.numpy()

Tensors are like NumPy arrays but with two superpowers: GPU acceleration and automatic gradient tracking.

Tensor Operations & Broadcasting

Familiar NumPy-like API with automatic shape broadcasting

```
# Arithmetic (element-wise)
c = a + b           # or torch.add(a, b)
c = a * b           # element-wise multiply
c = a @ b           # matrix multiply

# Reshaping
x = torch.randn(2, 3, 4)
y = x.view(2, 12)         # reshape (contiguous)
y = x.reshape(6, 4)       # always works
y = x.permute(0, 2, 1)    # transpose dims
y = x.unsqueeze(0)        # add dim


# Broadcasting (same rules as NumPy)
a = torch.randn(3, 4)     # (3, 4)
b = torch.randn(4)        # (4,) -> broadcast to (3, 4)
c = a + b                 # works!
```

view vs reshape

view requires contiguous memory.
reshape copies if needed.

In-place operations

a.add_(b) modifies a in-place.
Can break autograd — avoid in training.

@ vs * vs mm

@ = matmul (broadcasts)
* = element-wise
mm = strict 2D, bmm = batched

.item()

Get Python scalar from 0-dim tensor
for logging.

Autograd in Practice

The gradient computation API you'll use every day

Gradient Basics

```
x = torch.randn(3, requires_grad=True)
# Forward: build graph
y = x ** 2
loss = y.sum()
# Backward: compute gradients
loss.backward()
print(x.grad)   # tensor([...])  = 2*x
# IMPORTANT: gradients accumulate!
loss.backward()  # x.grad is now 4*x!
x.grad.zero_()   # must zero before next
```

Controlling Gradients

```
# Stopping gradient tracking
with torch.no_grad():        # no graph
    val_loss = model(x_val)
with torch.inference_mode(): # even faster
    val_loss = model(x_val)  # for eval
# Detach from graph
z = y.detach()  # shares data, no grad
# Logging safely
loss_value = loss.item()  # no grad
```

Critical Rules

1. Always zero gradients before backward
2. Use inference_mode() for eval (fastest)
3. Use .item() for logging (avoids mem leak)
4. Graph freed after backward (retain_graph=True if needed)

The #1 PyTorch bug: forgetting to zero gradients. They accumulate by design, but you must manage it.

nn.Module: Building Models

The base class for all neural network components in PyTorch

```python
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, in_dim, hidden, out_dim):
        super().__init__()
        self.fc1 = nn.Linear(in_dim, hidden)
        self.fc2 = nn.Linear(hidden, out_dim)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.1)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        return self.fc2(x)

model = MLP(784, 256, 10)
```

`.parameters()`
Iterator over all learnable params

`.to(device)`
Move all params to GPU/CPU

`.train() / .eval()`
Switches dropout/BN behavior

`.state_dict()`
Serialize weights for saving

Auto registration
Assign nn.Module as attribute → auto-registered as submodule

Rule: define layers in __init__, compute in forward. Never call forward() directly — use model(x) which calls hooks + forward.

Essential nn Layers

The building blocks you'll use most often

`nn.Linear(in, out)`	Fully connected: y = Wx + b
`nn.Conv2d(in_ch, out_ch, k)`	2D convolution with kernel size k
`nn.ReLU() / nn.GELU()`	Activation functions
`nn.BatchNorm2d(ch)`	Batch normalization (for CNNs)
`nn.LayerNorm(dim)`	Layer normalization (for Transformers)
`nn.Dropout(p)`	Randomly zero elements with prob p
`nn.Embedding(vocab, dim)`	Lookup table: integer → vector
`nn.MultiheadAttention(d, h)`	Multi-head self-attention
`nn.TransformerEncoderLayer(d, h)`	Full Transformer encoder block

Tip: nn.Sequential(nn.Linear(784, 256), nn.ReLU(), nn.Linear(256, 10)) — quick models without a custom class

torch.nn.functional: The Stateless API

Same operations as nn.Module layers, but as plain functions

```
import torch.nn.functional as F

class MyModel(nn.Module):
    def __init__(self, in_dim, hidden, out_dim):
        super().__init__()
        self.fc1 = nn.Linear(in_dim, hidden)
        self.fc2 = nn.Linear(hidden, out_dim)
        self.bn = nn.BatchNorm1d(hidden)

    def forward(self, x):
        x = F.relu(self.fc1(x))       # no params
        x = self.bn(x)                # has state
        x = F.dropout(x, p=0.1,
                training=self.training)
        return self.fc2(x)
```

Use nn.Module when:

- Layer has learnable parameters (Linear, Conv2d, Embedding)
- Layer has state that changes (BatchNorm, Dropout w/ training)

Use F.* when:

- Just a math operation (relu, gelu)
- Need explicit control over args
- One-off ops in forward()

```
Most used: F.relu  F.gelu  F.softmax(x, dim=-1)  F.cross_entropy(logits, labels)  F.mse_loss  F.dropout
F.normalize  F.pad
```

F.cross_entropy = nn.CrossEntropyLoss as a function. Both take raw logits. The functional form is handy for custom training steps.

Weight Initialization & Reproducibility

Get training started right, and make it reproducible

```
# Custom initialization
def init_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.kaiming_normal_(
            m.weight, nonlinearity='relu')
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(
            m.weight, mode='fan_out')

model = MLP(784, 256, 10)
model.apply(init_weights)
```

kaiming_normal_
Default for ReLU networks. Var = 2/fan_in.

xavier_uniform_
For tanh/sigmoid. Var = 2/(fan_in+fan_out).

normal_ / uniform_
Manual distributions.

zeros_ / ones_
For biases. Never for weights!

Reproducibility

```
torch.manual_seed(42)
torch.cuda.manual_seed_all(42)        # all GPUs
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Or with Lightning:
import lightning as L
L.seed_everything(42, workers=True)
```

Loss Functions & Optimizers

Connecting the model to the training objective

```
# Classification
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(logits, labels)  # logits: (B,C), labels: (B,)

# Regression
loss_fn = nn.MSELoss()
loss = loss_fn(predictions, targets)

# Binary classification
loss_fn = nn.BCEWithLogitsLoss()  # sigmoid + BCE


# Optimizer setup
optimizer = torch.optim.AdamW(
    model.parameters(), lr=3e-4, weight_decay=0.01)

# One training step
optimizer.zero_grad()
loss = loss_fn(model(x), y)
loss.backward()
optimizer.step()
```

CrossEntropyLoss

Takes raw logits (no softmax!)
Internally applies log_softmax + NLL

Adam vs AdamW

torch.optim.Adam with weight_decay
≠ true AdamW. Use torch.optim.AdamW.

Learning Rate

3e-4 is a good default for Adam.
1e-3 to 1e-1 for SGD.

The Holy Loop

zero_grad → forward → loss →
backward → step. Always this order.

Advanced Optimizer Patterns

Gradient clipping, per-layer learning rates, and hooks

```
# Gradient clipping (prevents exploding gradients)
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()

# Parameter groups: different LR for different parts
optimizer = torch.optim.AdamW([
    {'params': model.backbone.parameters(), 'lr': 1e-5},   # pretrained: low LR
    {'params': model.head.parameters(),     'lr': 3e-4},   # new layers: high LR
], weight_decay=0.01)
```

Hooks for Debugging

```
def print_grad(module, grad_in, grad_out):
    print(f'{module.__class__.__name__}: '
          f'grad_out norm = {grad_out[0].norm():.4f}')
model.fc1.register_backward_hook(print_grad)
```

When to clip?

- RNNs/LSTMs: almost always
- Transformers: yes (max_norm=1.0)
- CNNs: usually unnecessary
- If loss goes to NaN: try clipping

Parameter groups are essential for fine-tuning: freeze/slow-learn pretrained layers, fast-learn new layers.

Dataset, DataLoader & Transforms

Efficient data loading with batching, shuffling, and augmentation

```python
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

# Transforms: augmentation + normalization
train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225]),
])
val_transform = transforms.Compose([    # NO augmentation!
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225]),
])
```

Batching

Groups samples into mini-batches

Shuffling

Randomize order each epoch (critical for SGD convergence)

Parallelism

num_workers > 0 loads data in background while GPU trains

pin_memory=True

Faster GPU transfer via pre-allocated memory

Augmentation = free regularization. Always different transforms for train vs val. Normalize to ImageNet stats when using pretrained models.

03

The Training Pipeline

Putting it all together: from data to trained model

The Training Flow Template

Every deep learning project follows the same pattern

1. Prepare	Data → Dataset → DataLoader	*Batching, shuffling, transforms*
2. Build	Define model (nn.Module)	*Layers in __init__, logic in forward*
3. Configure	Loss function + Optimizer	*What to minimize + how to update*
4. Train	The inner loop (see next slides)	*forward → loss → backward → step*
5. Validate	Evaluate on held-out data	*Detect overfitting, track metrics*
6. Save	Checkpoint best model	*So you can resume or deploy later*

Steps 4–6 repeat every epoch. The rest of Part 3 teaches each step in detail, ending with how Lightning organizes them all.

The 5-Line Core

This is the atomic unit of all deep learning training

1	`optimizer.zero_grad()`	**Clear old gradients**	Gradients accumulate by default. Without this, you get wrong updates.
2	`logits = model(x)`	**Forward pass**	Run input through the network. Computation graph is built here.
3	`loss = loss_fn(logits, y)`	**Compute loss**	Single scalar measuring how wrong the prediction is.
4	`loss.backward()`	**Backward pass**	Walk the graph in reverse. Compute gradient for every parameter.
5	`optimizer.step()`	**Update weights**	Apply gradients: $\theta \rightarrow \theta - lr \times \nabla\theta$. One step of gradient descent.

This 5-line pattern never changes. Everything else (GPU, scheduling, logging, checkpoints) wraps around it.

The Complete Training Loop

The 5-line core, wrapped in epochs and batches

```
# train.py — Canonical Training Loop (with gradient clipping)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = MLP(784, 256, 10).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
loss_fn = nn.CrossEntropyLoss()

for epoch in range(num_epochs):
    model.train()                        # enable dropout/BN
    for batch_x, batch_y in train_loader:
        batch_x = batch_x.to(device)     # move to GPU
        batch_y = batch_y.to(device)
        logits = model(batch_x)          # forward
        loss = loss_fn(logits, batch_y)  # compute loss
        optimizer.zero_grad()            # clear grads
        loss.backward()                  # backprop
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()                 # update weights
    print(f'Epoch {epoch}: loss={loss.item():.4f}')
```

This is the skeleton. Everything else (validation, scheduling, logging, checkpoints) is built around this loop.

Validation & Evaluation

Always evaluate on held-out data to detect overfitting

```python
# evaluate.py — Validation
@torch.no_grad()                        # disable autograd entirely
def evaluate(model, loader, loss_fn, device):
    model.eval()                        # disable dropout/BN
    total_loss, correct, total = 0, 0, 0
    for batch_x, batch_y in loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)
        logits = model(batch_x)
        total_loss += loss_fn(logits, batch_y).item() * len(batch_y)
        correct += (logits.argmax(1) == batch_y).sum().item()
        total += len(batch_y)
    return total_loss / total, correct / total

# Call after each epoch
val_loss, val_acc = evaluate(model, val_loader, loss_fn, device)
print(f'Val loss={val_loss:.4f}, acc={val_acc:.2%}')
```

Key: model.eval() switches BatchNorm/Dropout to inference mode. @torch.no_grad() saves memory by not building the graph.

GPU & Mixed Precision

Move to GPU, then train faster with automatic mixed precision

```
# GPU setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)

# Mixed precision training (AMP) -- 1.5-2x faster, half the memory
scaler = torch.amp.GradScaler()

for batch_x, batch_y in train_loader:
    batch_x, batch_y = batch_x.to(device), batch_y.to(device)

    with torch.amp.autocast(device_type='cuda'):    # auto float16
        logits = model(batch_x)
        loss = loss_fn(logits, batch_y)

    scaler.scale(loss).backward()      # scale loss to prevent underflow
    scaler.step(optimizer)             # unscale & step
    scaler.update()                    # adjust scale factor
    optimizer.zero_grad()
```

autocast

Runs matmuls in float16 for speed, keeps accumulations in float32 for accuracy. Best of both.

GradScaler

float16 has tiny range → grads can underflow to 0. Scaler multiplies loss to prevent this.

bfloat16

Same range as float32 but half precision. No scaler needed. A100/H100+ only.

AMP is nearly free performance. In Lightning: Trainer(precision='16-mixed'). One flag, no code changes.

Learning Rate Scheduling

Warm up, then decay — the modern default

```
from torch.optim.lr_scheduler import (
    CosineAnnealingLR, LinearLR, SequentialLR
)

optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)

# Warmup: 0 -> 3e-4 over 500 steps
warmup = LinearLR(optimizer, start_factor=0.01, total_iters=500)
# Cosine decay: 3e-4 -> 0 over remaining steps
cosine = CosineAnnealingLR(optimizer, T_max=total_steps - 500)
# Combine them
scheduler = SequentialLR(optimizer, [warmup, cosine], milestones=[500])

# In training loop:
for batch in train_loader:
    ...                     # forward, backward, step
    scheduler.step()        # update LR after each step
```

Call scheduler.step() after optimizer.step(). In Lightning: return {'optimizer': opt, 'lr_scheduler': sched} from configure_optimizers().

Checkpoints & Early Stopping

Save the best model, stop when it stops improving

```python
# Save checkpoint
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'val_loss': val_loss,
}, 'checkpoint.pt')
# Load checkpoint
ckpt = torch.load('checkpoint.pt', map_location=device)
model.load_state_dict(ckpt['model_state_dict'])
```

```python
# Early stopping (vanilla pattern)
best_loss = float('inf')
patience, counter = 5, 0
for epoch in range(max_epochs):
    val_loss = evaluate(model, val_loader, ...)
    if val_loss < best_loss:
        best_loss = val_loss; counter = 0
        torch.save(model.state_dict(), 'best.pt')
    else:
        counter += 1
        if counter >= patience: break
```

Lightning: 3 lines

```python
from lightning.pytorch.callbacks import (
    ModelCheckpoint, EarlyStopping)
trainer = L.Trainer(callbacks=[
    ModelCheckpoint(monitor='val_loss',
        mode='min', save_top_k=3),
    EarlyStopping(monitor='val_loss',
        patience=5),
])
```

Also useful callbacks:

LearningRateMonitor
RichProgressBar
GradientAccumulationScheduler
StochasticWeightAveraging

Organizing It All: Lightning

You now know every piece. Lightning is just a standard way to organize them.

```python
import lightning as L

class LitModel(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.model = MLP(784, 256, 10)
        self.loss_fn = nn.CrossEntropyLoss()

    def training_step(self, batch, batch_idx):
        x, y = batch
        loss = self.loss_fn(self.model(x), y)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        loss = self.loss_fn(self.model(x), y)
        self.log('val_loss', loss)

    def configure_optimizers(self):
        return torch.optim.AdamW(
            self.parameters(), lr=3e-4)
```

Model (nn.Module)	→	`__init__ + forward`
Forward + loss	→	`training_step()`
Validation loop	→	`validation_step()`
Optimizer + LR	→	`configure_optimizers()`
GPU + Mixed prec.	→	`Trainer(accelerator, precision)`
Gradient clipping	→	`Trainer(gradient_clip_val)`
Checkpoints	→	`Callbacks list`

Lightning Trainer: The Full Picture

Every flag = something you'd manually code in vanilla PyTorch

```python
import lightning as L

trainer = L.Trainer(
    # Hardware
    accelerator='cuda',             # GPU training
    devices=2,                      # 2 GPUs (auto DDP)

    # Training
    max_epochs=100,
    gradient_clip_val=1.0,          # clip_grad_norm_
    accumulate_grad_batches=4,      # effective batch = 4x
    precision='16-mixed',           # AMP

    # Callbacks
    callbacks=[ModelCheckpoint(...), EarlyStopping(...)],

    # Logging
    log_every_n_steps=10,
    enable_progress_bar=True,
)

trainer.fit(model, train_loader, val_loader)
```

`devices=2`	torch.nn.parallel.DDP setup
`precision='16-mixed'`	autocast + GradScaler
`gradient_clip_val`	clip_grad_norm_ call
`accumulate_grad_batches`	Manual grad accumulation
`callbacks=[...]`	Manual if/save/break logic
`log_every_n_steps`	Manual print/logging
`trainer.fit()`	The entire double for-loop

Lightning = vanilla PyTorch, organized. Same ops under the hood. You can always drop down to raw code.

Debugging Checklist

When your model isn't training, check these first

①	**Overfit one batch first**	Can the model memorize 1 batch? If not, bug in model or loss.
②	**Check shapes**	Print shapes at each step. Most bugs are shape mismatches.
③	**Verify gradients exist**	for p in model.parameters(): assert p.grad is not None
④	**Learning rate**	Too high → loss explodes. Too low → barely moves. Start 3e-4.
⑤	**Data normalization**	Normalize to mean=0, std=1. Unnormalized → unstable.
⑥	**Check labels**	0-indexed? Correct class count? Print (input, label) pairs.
⑦	**train() vs eval()**	Forgot model.train()? Dropout/BN in wrong mode.

"If your model isn't learning, it's almost always a bug, not a hyperparameter problem." — Andrej Karpathy

Modern PyTorch: torch.compile

Get static-graph speed with dynamic-graph ergonomics

```
# One line to speed up your model (PyTorch 2.0+)
model = torch.compile(model)

# That's it. Training loop stays exactly the same.
# First iteration is slow (compilation), then 1.5-2x faster.

# Options:
model = torch.compile(model, mode='reduce-overhead')  # less memory
model = torch.compile(model, mode='max-autotune')     # most optimized
model = torch.compile(model, backend='inductor')       # default backend
```

How It Works

Traces your eager Python code into an optimized graph (via TorchDynamo), then compiles it (via TorchInductor). Best of both worlds: write dynamic code, run at compiled speed.

When to Use

Production / long training runs: yes. Quick experiments: maybe not (compile time overhead). Dynamic shapes (varying seq lengths) can cause recompilation. Always works as a drop-in optimization.

The PyTorch Ecosystem

Libraries you'll use alongside core PyTorch

torchvision	Image datasets, transforms, pretrained models (ResNet, ViT, ...)
torchaudio	Audio processing, datasets, pretrained models
HuggingFace	Pretrained LLMs, tokenizers, Trainer. The NLP/LLM standard.
Lightning	Reduce boilerplate: handles distributed, logging, checkpoints
Weights & Biases	Experiment tracking: loss curves, hyperparams, model versioning
Hydra / OmegaConf	Configuration management for experiments. YAML-based configs.
torchinfo	Model summary (shapes, param counts). Like Keras model.summary().

Start with raw PyTorch to understand the fundamentals. Then adopt Lightning for training, HuggingFace for models, W&B for tracking.

Key Takeaways

1. Autograd traces operations on the forward pass, then applies the chain rule in reverse. One backward pass = all gradients.

2. PyTorch = dynamic graph. Write Python, get gradients. Debug normally.

3. The training loop: zero_grad → forward → loss → backward → step. Memorize this 5-line pattern.

4. nn.Module: define layers in __init__, compute in forward. Lightning adds structure on top.

5. Always validate on held-out data. Always checkpoint. Lightning automates both with Callbacks.

The Complete Picture

Your Code	model = MLP(...) → loss.backward() → optimizer.step()
PyTorch	Tensors + Autograd + nn.Module + DataLoader + Lightning
torch.compile	TorchDynamo traces → TorchInductor compiles → optimized kernels
CUDA / cuDNN	GPU-accelerated linear algebra, convolutions, attention
Hardware	NVIDIA GPUs (A100, H100) / TPUs / Apple Silicon