Training Deep Networks in Practice

Lecture 3 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

Learning outcomes

By the end of this lecture you will be able to:

1. Write a **PyTorch training loop** by hand.
2. Apply the **seven-rung debug ladder** · data → overfit → scale up.
3. Use **LR finder** to pick a learning rate in 100 steps.
4. Set up **mixed-precision** training with autocast.
5. Diagnose **training curves** · underfit, sweet spot, overfit.
6. Save and load **checkpoints** correctly (state_dict, not pickle).
7. Detect **validation leakage** and train/test distribution shift.
8. Run disciplined **ablations** instead of changing many knobs at once.

Recap · what we have so far

- **Depth works** with ResNets + He init + ReLU (L2).
- **PyTorch recipe** — forward, loss, zero_grad, backward, step (L1).

> REFERENCE
>
> Today maps to **UDL Ch 6** (early — fitting models) and **Ch 8** (measuring performance).

Theory is done. Today is entirely **practical**.

Four questions:

1. How does PyTorch actually wire this together?
2. How do you build a data pipeline that doesn't bottleneck the GPU?
3. What is the **debugging procedure** for when training fails?
4. How do you do **error analysis** (Ng-style) after training?

Pop quiz · "my model isn't learning"

You wrote a model, hit `train()`. Loss stays flat for 100 steps.

POP QUIZ

(a) Lower the learning rate by 10×.
(b) Switch from SGD to AdamW.
(c) Try to **overfit a single batch of 4 examples**.
(d) Add a deeper architecture.

Stop and pick *one* before the next slide. Why that one?

The right answer (revealed in PART 4) is the one that **isolates the bug fastest** — and it's not the one most students reach for first.

PART 1

The PyTorch stack

What happens when you type `model(x)`

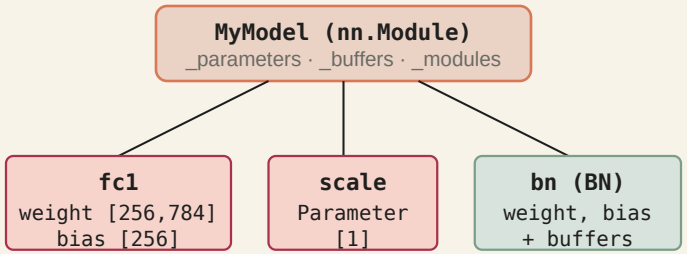
`nn.Module` · the container

Inside `nn.Module` — what `parameters()`, `.to()`, `state_dict()` actually track

YOUR CODE

```
class MyModel(nn.Module):
def __init__(self):
super().__init__()
self.fc1 = nn.Linear(784, 256) # ← Parameter
self.fc2 = nn.Linear(256, 10) # ← Parameter
self.scale = nn.Parameter(torch.ones(1))
self.register_buffer('mean',
torch.zeros(784)) # ← Buffer
self.bn = nn.BatchNorm1d(256) # ← submodule
def forward(self, x):
```

INTERNAL REGISTRY · WHAT PYTORCH STORES



WHAT NN.MODULE GIVES YOU

- **`model.parameters()`** → iterable of all learnable tensors, for optimizer
- **`model.state_dict()`** → serializable ordered-dict of all params + buffers (for saving)
- **`model.to('cuda')`** → move params + buffers (not buffers stored by you)
- **`model.train() / .eval()`** → toggle BatchNorm, Dropout behaviour

Parameter registration · a common footgun

WATCH OUT

If you store a tensor as `self.foo = torch.tensor(...)`, PyTorch will **not** track it as a parameter. No gradients, won't get moved to GPU, won't be saved.

- Learnable: use `nn.Parameter(torch.tensor(...))`.
- Non-learnable but device-bound: use `self.register_buffer('foo', ...)` (running stats, position codes).

```
self.temperature = nn.Parameter(torch.ones(1))   # ← learnable
self.register_buffer('running_mean',
                     torch.zeros(dim))            # ← moves with .to() but no gradient
```

How does PyTorch know the derivatives?

It watches you compute.

INTUITION

Analogy · Baking a Cake and Asking "Why is it so sweet?"

A friend watches you bake. They write down every step: *"1 cup flour, 2 cups sugar, 1 egg..."*. When you taste the cake and say *"too sweet,"* your friend looks at the notes (the **tape**) and says: *"The sugar had the biggest impact — use less and the sweetness drops the most."*

Autograd is that friend. It records every operation in a **computation graph**. When you call `.backward()`, it walks back through the graph to see how much each parameter contributed to the loss.

Autograd · let's build a graph by hand

Trace a simple computation: $L = a \cdot x + b$ with $a = 2,\ x = 3,\ b = 1$. Parameters: a, b. Input: x.

Forward pass

- $c = a \cdot x = 2 \cdot 3 = 6$
- $L = c + b = 6 + 1 = 7$

PyTorch builds the graph **as we compute**.

Backward pass (chain rule)

- $\partial L / \partial L = 1$
- $L = c + b \Rightarrow \partial L / \partial c = 1,\ \partial L / \partial b = 1$ ✓
- $c = a \cdot x \Rightarrow \partial L / \partial a = \partial L / \partial c \cdot x = 1 \cdot 3 = 3$ ✓

`loss.backward()` does this for every parameter, no matter how deep the graph.

Autograd · the dynamic tape

PyTorch autograd — every tensor op silently records how to undo itself

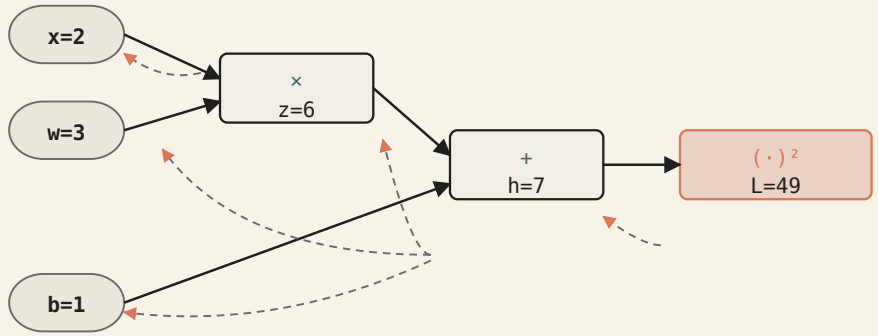
PYTHON

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

z = w * x # MulBackward
h = z + b # AddBackward
L = h ** 2 # PowBackward

L.backward()
# → w.grad = 28, x.grad = 42, b.grad = 14
```

AUTOGRAD TAPE (DAG)



*The tape is built **at runtime** (dynamic graph)*

Each node stores grad_fn pointing to the backw

WATCH

Wrap eval / inference in with torch.no_grad(): —

Worked numeric · a 2-layer backward step

Compute $\partial L/\partial w_1$ for $L = (\text{relu}(w_2 h) - y)^2$ where $h = \text{relu}(w_1 x)$.
Let $x = 2,\ y = 1,\ w_1 = w_2 = 0.5$.

Forward

- $a_1 = w_1 x = 0.5 \cdot 2 = 1.0$
- $h = \text{relu}(a_1) = 1.0$
- $a_2 = w_2 h = 0.5$
- $\hat{y} = \text{relu}(a_2) = 0.5$
- $L = (\hat{y} - y)^2 = 0.25$

Backward (chain rule)

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a_2} \cdot \frac{\partial a_2}{\partial h} \cdot \frac{\partial h}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$

- $\partial L/\partial \hat{y} = 2(\hat{y} - y) = -1.0$
- ReLU' = 1 (both gates open)
- $\partial a_2/\partial h = w_2 = 0.5$
- $\partial a_1/\partial w_1 = x = 2$

$$\partial L/\partial w_1 = (-1.0)(1)(0.5)(1)(2) = \boxed{-1.0}$$

Two safety habits

`torch.no_grad()` for evaluation

```
model.eval()
with torch.no_grad():
    for x, y in val_loader:
        pred = model(x)
        ...
```

Skips tape construction · faster · less memory.

`.detach()` to stop gradient flow

```
target = model_old(x).detach()
loss = mse(model_new(x), target)
```

Treats the tensor as a constant in the graph.

PART 2

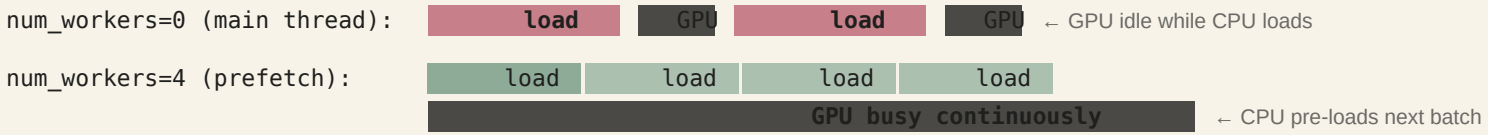
The data pipeline

CPU loads while the GPU computes

Dataset → DataLoader → device

The PyTorch data pipeline — CPU prep while the GPU computes

WHAT NUM_WORKERS DOES



Rule of thumb · num_workers ≈ 4 × num_GPUs · pin_memory=True with GPU · persistent_workers=True for long epochs.

The batch contract

Before the model sees a batch, the batch should satisfy a contract.

ITEM	CLASSIFICATION EXPECTATION	COMMON BUG
`x.shape`	`[B, C, H, W]` for images or `[B, d]` for vectors	missing batch dimension
`x.dtype`	`float32` / `bfloat16` after transforms	raw `uint8` pixels
`x.range`	normalized, often roughly centered	values still in `[0, 255]`
`y.shape`	`[B]` for class indices	one-hot when loss expects indices
`y.dtype`	`torch.long` for `CrossEntropyLoss`	float labels

```
assert x.ndim == 4
assert x.dtype in (torch.float32, torch.bfloat16)
assert y.ndim == 1 and y.dtype == torch.long
```

KEY IDEA

Many "model bugs" are actually batch-contract bugs. Verify the batch before touching the architecture.

Writing a custom `Dataset`

```python
from torch.utils.data import Dataset

class TinyImageDataset(Dataset):
    def __init__(self, paths, labels, tfm=None):
        self.paths, self.labels, self.tfm = paths, labels, tfm

    def __len__(self):
        return len(self.paths)

    def __getitem__(self, i):
        img = PIL.Image.open(self.paths[i]).convert('RGB')
        if self.tfm: img = self.tfm(img)
        return img, self.labels[i]
```

Two methods. That is the entire contract.

DataLoader flags that matter

```
loader = DataLoader(dataset,
    batch_size=64,
    shuffle=True,
    num_workers=4,
    pin_memory=True,
    persistent_workers=True,
    drop_last=True)
```

INTUITION

Rule of thumb · `num_workers ≈ 4 × num_GPUs`. `pin_memory=True` when using CUDA. `persistent_workers=True` when epochs are short.

Is the GPU waiting?

Data loading is successful when the GPU almost never waits for the CPU.

SYMPTOM	LIKELY CAUSE	FIRST FIX
GPU utilization sawtooths	CPU/disk cannot feed batches	increase `num_workers`
first batch slow every epoch	worker restart overhead	`persistent_workers=True`
transfer to CUDA slow	pageable host memory	`pin_memory=True`
random crop dominates time	heavy CPU transforms	cache, simplify, or move to GPU

```
import time
t0 = time.perf_counter()
for i, batch in enumerate(loader):
    if i == 100: break
print("batches/sec", 100 / (time.perf_counter() - t0))
```

KEY IDEA

Benchmark the loader alone. If data throughput is low, a bigger model or better optimizer will not fix the bottleneck.

Why not just use full precision (FP32)?

Modern GPUs are highways for numbers. To go faster: build a bigger highway (new GPU) **or make the cars smaller**.

- **FP32** (single precision) — the standard car. 32 bits per number.
- **FP16** (half precision) — a motorcycle. 16 bits.
  - **Pro:** 2× as many fit on the highway → ≈2× faster training, half memory.
  - **Con:** small range. Big numbers don't fit; gradients **overflow** to infinity → run dies.

We want the speed of 16 bits **without** losing FP32's range.

Precision vs. range · the trade-off

INTUITION

Analogy · measuring with rulers

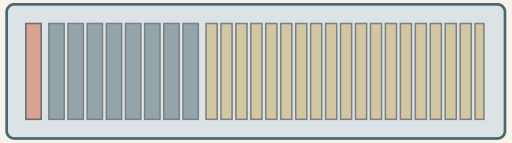
- **FP32** → 1-metre stick with millimetre marks. Big range, fine precision. Default.
- **FP16** → 15 cm ruler with millimetre marks. **Very precise**, but **range is tiny**. Try to measure a person's height — ruler runs out instantly.
- **BF16** → 1-metre stick with **centimetre** marks. **Same range as FP32**, less precision. Sacrifices a little precision for huge range.

For deep learning, **range matters far more than ultra-fine precision**. BF16 almost never overflows.

Mixed precision · BF16 is the 2026 default

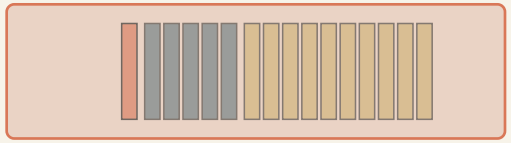
Mixed precision — trading bits for speed

FP32 (DEFAULT) · 32 BITS



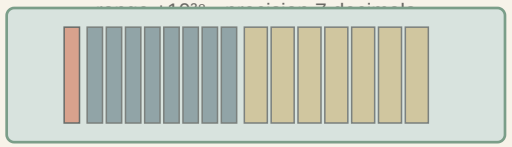
sign · 8 exp · 23 mantissa

FP16 · 16 BITS (IEEE HALF)



sign · 5 exp · 10 mantissa

BF16 · 16 BITS (BRAIN FLOAT, AMPERE+ DEFAULT)



sign · 8 exp · 7 mantissa

FP32 range · FP16 memory · sweet spot

PYTORCH · THE ACTUAL RECIPE

```
scaler = torch.amp.GradScaler() # only needed for fp16

with torch.amp.autocast('cuda',
dtype=torch.bfloat16):
logits = model(x) # runs in bf16
loss = criterion(logits, y)

loss.backward(); opt.step(); opt.zero_grad()
```

*Default in 2026 · use **BF16 autocast** on Ampere+ / TPU. 2× speedup, half the memory, no GradScaler needed.*

Mixed precision · why BF16 > FP16

FP16

- Range · ±65,504
- Precision · 3-4 decimal digits
- Easy to **overflow** during loss computation
- Needs **loss scaling** to keep gradients in range

BF16

- Range · $\pm 10^{38}$ (same as FP32)
- Precision · 2-3 decimal digits
- Never overflows
- **No loss scaling needed**

KEY IDEA

BF16 · trades precision for range. Same memory as FP16. Available on NVIDIA A100+, AMD MI200+, TPU v3+. Default on any modern LLM training.

```
with torch.autocast(device_type='cuda', dtype=torch.bfloat16):
    output = model(input)   # most ops in BF16
    loss = criterion(output, target)
loss.backward()             # grads in BF16 too
```

What if the batch doesn't fit in your GPU?

Your GPU handles batches of 64, but the model trains better with batch 256.

INTUITION

Analogy · polling a small town. You want the average opinion of 256 people, but only 64 fit in the room.

1. Bring in 64. Tally — but **do not declare a result**.
2. Bring in the next 64. Add to the tally.
3. Repeat for groups 3 and 4.
4. After all 256 → compute the average → make a decision.

Gradient accumulation does exactly this with batches: it processes several **micro-batches**, sums their gradients, and updates the weights **once** at the end.

Simulating a big batch · gradient accumulation

The key identity: **gradient of a sum = sum of gradients.**

$$\nabla\left(\sum_i L_i\right) = \sum_i \nabla L_i$$

Effective batch 256, GPU fits 64 → $K = 4$ accumulation steps.

```
for i, (x, y) in enumerate(loader):
    loss = criterion(model(x), y) / K   # divide so we average over K
    loss.backward()                     # ADDS to existing grads in .grad
    if (i + 1) % K == 0:
        opt.step()                      # update once per K micro-batches
        opt.zero_grad()                 # then clear for next big batch
```

Two crucial details: divide loss by K (so we get the **mean**, matching a real batch of 256), and only call `step()` / `zero_grad()` after K micro-batches.

Worked numeric · gradient accumulation

Update weight w for $L = (wx - y)^2$. LR $\alpha = 0.1$. Effective batch 2, micro-batch 1, $K = 2$, start $w = 0$.

Micro-batch 1 · $x = 2,\ y = 1$

- $L_1 = (0 \cdot 2 - 1)^2/2 = 0.5$
- $\nabla L_1 = \frac{x}{K} \cdot 2(wx - y) = \frac{2}{2} \cdot 2(-1) = -2$
- `loss.backward()` → `w.grad = -2`
- *No `step()`, no `zero_grad()`.*

Micro-batch 2 · $x = 3,\ y = 2$

- $L_2 = (0 \cdot 3 - 2)^2/2 = 2.0$
- $\nabla L_2 = \frac{3}{2} \cdot 2(-2) = -6$
- `loss.backward()` → `w.grad = -2 + (-6) = -8`

Update step. $w_{\text{new}} = 0 - 0.1 \cdot (-8) = 0.8$. Then `zero_grad()` resets `w.grad = 0`.

This is the **exact** update we'd get from one big batch of $\{(2, 1), (3, 2)\}$.

Why we need gradient clipping

Some batches — especially in RNNs and Transformers — produce **ridiculously large** gradients. This is an **exploding gradient**.

> INTUITION
>
> **Analogy · learning to drive.** Your instructor gives small corrections: *"turn 5°," "forward 10 cm."* Then suddenly screams **"TURN 10,000° LEFT!"**. Following that literally → smashed wall. Weights destroyed, restart training.

Gradient clipping is a safety rule:

> *"No matter what gradient is computed, never let the step be larger than* `max_norm` *."*

It prevents one bad batch from wrecking the entire run.

How gradient clipping works

A gradient is a vector — direction × magnitude.

1. After `loss.backward()`, gather full gradient vector g.
2. Compute its **L2-norm** $\|g\|$.
3. Set threshold `max_norm` (e.g. 1.0).
4. If $\|g\| > \text{max\_norm}$, **rescale**:

$$g_{\text{clipped}} = g \cdot \frac{\text{max\_norm}}{\|g\|}$$

5. Otherwise, leave g alone.

Direction is preserved — only step size is capped.

```
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
opt.step()
```

Worked numeric · gradient clipping

Two weights w_1, w_2. After `.backward()`: $g = [3.0,\ 4.0]$. Set `max_norm = 1.0`.

1. **Compute the norm.**
   $\|g\| = \sqrt{3^2 + 4^2} = \sqrt{25} = 5.0$
2. **Compare.** $5.0 > 1.0$ → must clip.
3. **Scaling factor.** $s = \text{max\_norm} / \|g\| = 1.0 / 5.0 = 0.2$
4. **Rescale.** $g_{\text{clipped}} = [3.0 \cdot 0.2,\ 4.0 \cdot 0.2] = [0.6,\ 0.8]$
5. **Verify.** $\|g_{\text{clipped}}\| = \sqrt{0.36 + 0.64} = 1.0$ ✓

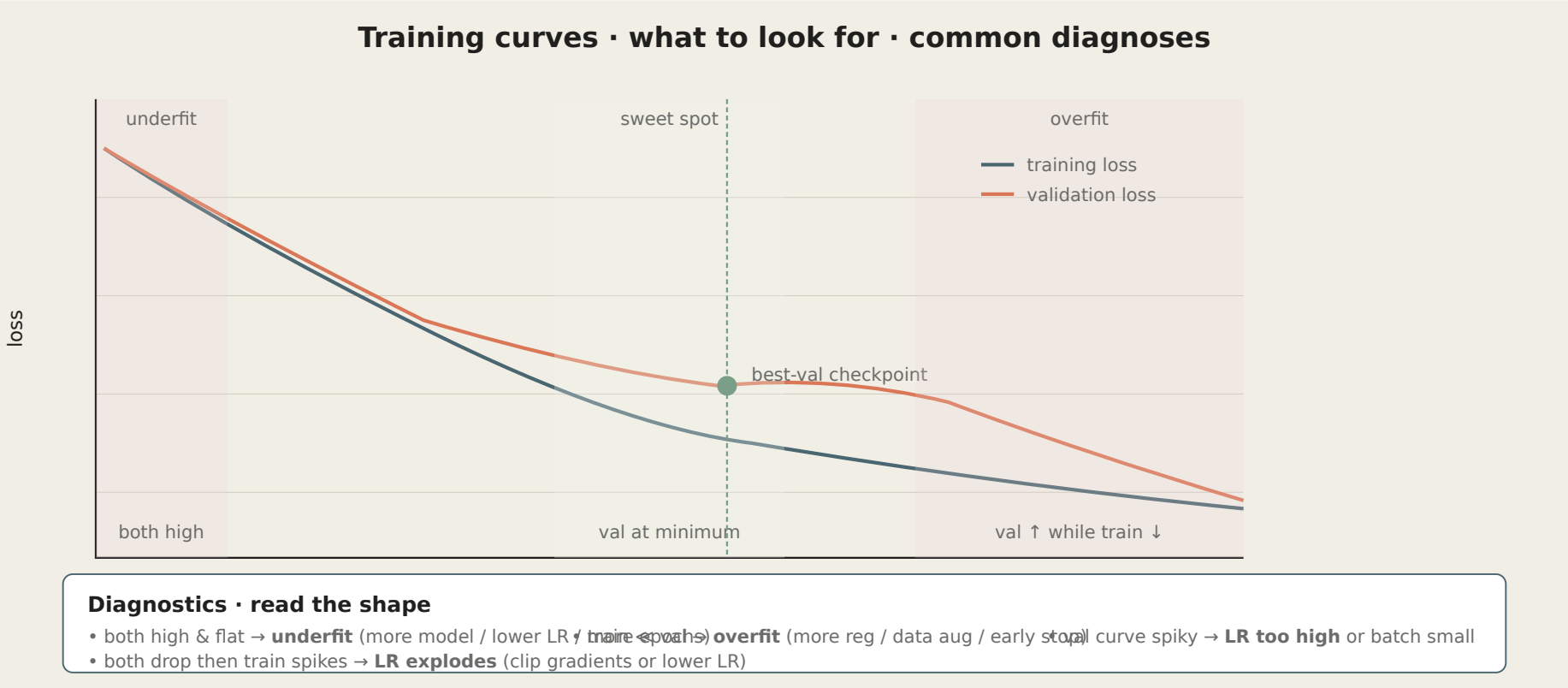
   The optimizer now uses $[0.6, 0.8]$ — same direction as $[3, 4]$ but a much safer step.

PART 3

The full training recipe

From zero to a trained model

Training curves · the diagnostic language



Loss and metric are not the same

Training optimizes a **loss**. Reporting uses a **metric**. They answer different questions.

SETTING	OPTIMIZED LOSS	REPORTED METRIC	FAILURE IF CONFUSED
balanced classification	cross-entropy	accuracy	hides calibration
imbalanced classification	weighted CE / focal	F1, AUROC, AUPRC	high accuracy by predicting majority
regression	MSE / MAE / NLL	RMSE, MAE, R^2	loss punishes errors differently
ranking / retrieval	contrastive / pairwise	recall@k, MRR	good loss, poor top-k behavior

WATCH OUT

Choose the metric before training. Otherwise you will optimize the convenient loss and later discover it does not match the real objective.

The recipe · one function

```
def train(model, loader_tr, loader_val, opt, loss_fn, n_epochs, device):
    best_val = float('inf')
    for ep in range(n_epochs):
        model.train()
        for x, y in loader_tr:
            x, y = x.to(device), y.to(device)
            loss = loss_fn(model(x), y)
            opt.zero_grad(); loss.backward(); opt.step()

        val = evaluate(model, loader_val, loss_fn, device)
        print(f'epoch {ep:3d}  val_loss={val:.4f}')
        if val < best_val:
            best_val = val
            torch.save(model.state_dict(), 'best.pt')
```

Every real script is a variation on this. Keep it boring.

Save / load correctly

Save

```
torch.save({
    'model':  model.state_dict(),
    'optim':  opt.state_dict(),
    'epoch':  ep,
    'config': cfg,
}, 'ckpt.pt')
```

Load

```
ckpt = torch.load('ckpt.pt',
                  map_location=device,
                  weights_only=True)
model.load_state_dict(ckpt['model'])
opt.load_state_dict(ckpt['optim'])
```

WATCH OUT

Don't `torch.save(model)` — it pickles the class, which breaks across refactors. Always save `state_dict`.

Before trusting validation

A validation score is useful only if the split matches the deployment question.

PROBLEM	WHAT GOES WRONG	BETTER SPLIT
same patient in train and val	memorizes patient-specific artifacts	split by patient
adjacent video frames	train and val nearly duplicates	split by video / scene
time-series random split	future leaks into training	chronological split
user logs	same user behavior in both sets	split by user or time
repeated documents	near-duplicate text leakage	split by document/source

WATCH OUT

Leakage makes the training recipe look correct while the model has learned the wrong thing. Check the split before celebrating a curve.

Distribution shift is the next test

Even without leakage, validation may differ from real deployment.

SHIFT	EXAMPLE	PRACTICAL RESPONSE
covariate shift	new camera, hospital, device	collect representative val/test
label shift	class priors change	report per-class metrics
concept shift	definition of target changes	refresh labels and monitor drift
temporal shift	user behavior changes over time	time-based test set

KEY IDEA

The question is not "did validation improve?" The question is "does validation measure the future cases we care about?"

PART 4

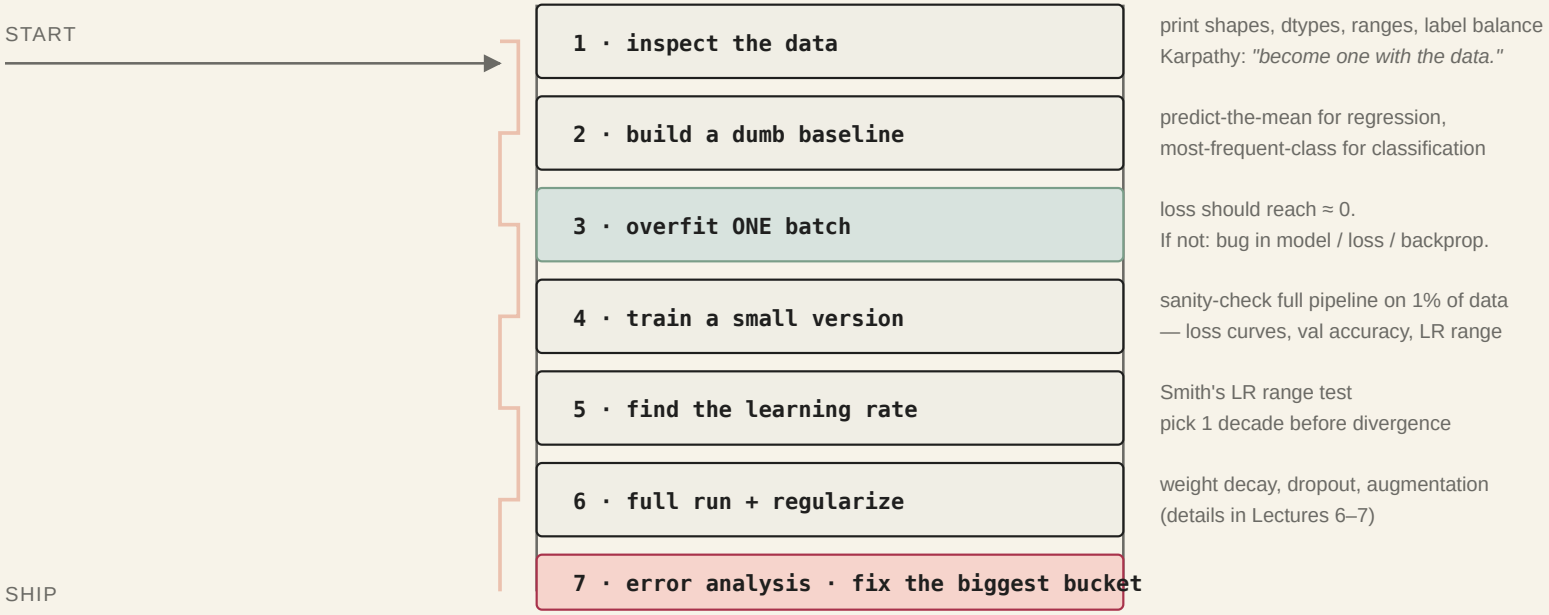
The debugging ladder

(Karpathy's recipe — don't skip a rung)

The ladder

Andrej Karpathy's recipe — climb the debugging ladder one rung at a time

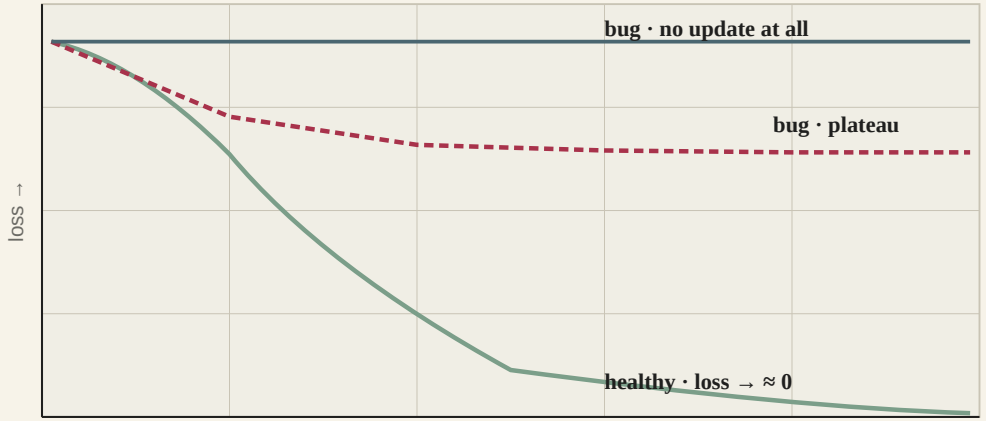
NEVER SKIP A RUNG. IF THE CURRENT RUNG FAILS, DEBUG HERE — DON'T TUNE ABOVE.



Rung 3 · overfit one batch

The first diagnostic: can your model overfit a single batch?

IF YES → ARCHITECTURE + LOSS ARE WIRED CORRECTLY · IF NO → THERE IS A BUG



IF IT PLATEAUS, CHECK:

- ☐ `lr too small / too large`
- ☐ `softmax + CE double-applied`
- ☐ `forgot optimizer.zero_grad()`
- ☐ `dead ReLUs (all activations 0)`
- ☐ `frozen params (.requires_grad)`
- ☐ `wrong label shape / dtype`
- ☐ `data normalization missing`

Fix these before scaling up.

Debug checklist · why won't loss go down? · part 1

If you can't even overfit a single batch, something is fundamentally broken.

- **LR too large** → loss explodes to `NaN`. **Fix:** divide LR by 10.
- **LR too small** → loss barely moves. **Fix:** multiply LR by 10.
- **Double softmax** → model ends with `nn.Softmax` AND you use `nn.CrossEntropyLoss`. The loss already includes softmax → applied twice → gradients muffled. **Fix:** remove `nn.Softmax` from the model; output raw **logits**.
- **Forgot** **`opt.zero_grad()`** → gradients from previous batches pile up; updates point in nonsense directions. **Fix:** add `opt.zero_grad()` at the start of each step.

Debug checklist · why won't loss go down? · part 2

- **Dead ReLUs** → input to a ReLU is always negative → output 0, gradient 0; the unit can never learn. **Fix:** LeakyReLU, lower LR, or He init.
- **Frozen parameters** → a layer has `requires_grad=False`. It will never update. **Fix:** assert `param.requires_grad` for each layer you intend to train.
- **Wrong label shape / dtype** → `CrossEntropyLoss` wants class **indices** (e.g. `[3, 0, 1]`, dtype `torch.long`). One-hot floats break it. **Fix:** check `.shape` and `.dtype` of the label tensor.
- **Data not normalized** → raw pixels in $[0, 255]$ → unstable training. **Fix:** apply `ToTensor` + `Normalize`.

> IN PRACTICE
>
> Karpathy: *"Become one with the data."* Before touching the model, print shapes, dtypes, ranges, label balance, and a few random examples.

How do we find a good starting LR?

The learning rate is **the most important** hyperparameter.

- Too high → training diverges (loss explodes).
- Too low → training crawls (or gets stuck).

We want the **highest LR that is still stable** — without dozens of trial-and-error runs.

> INTUITION
>
> **Analogy · pushing a cart up a hill.** Find the hardest you can push without tipping. Start with a tiny push and gradually increase. More force → more speed. At some point the cart wobbles. The best push is **just before** the wobble.

The LR finder does this with your learning rate.

The LR finder algorithm

A short fake training session (≈100 steps) sweeps over LRs.

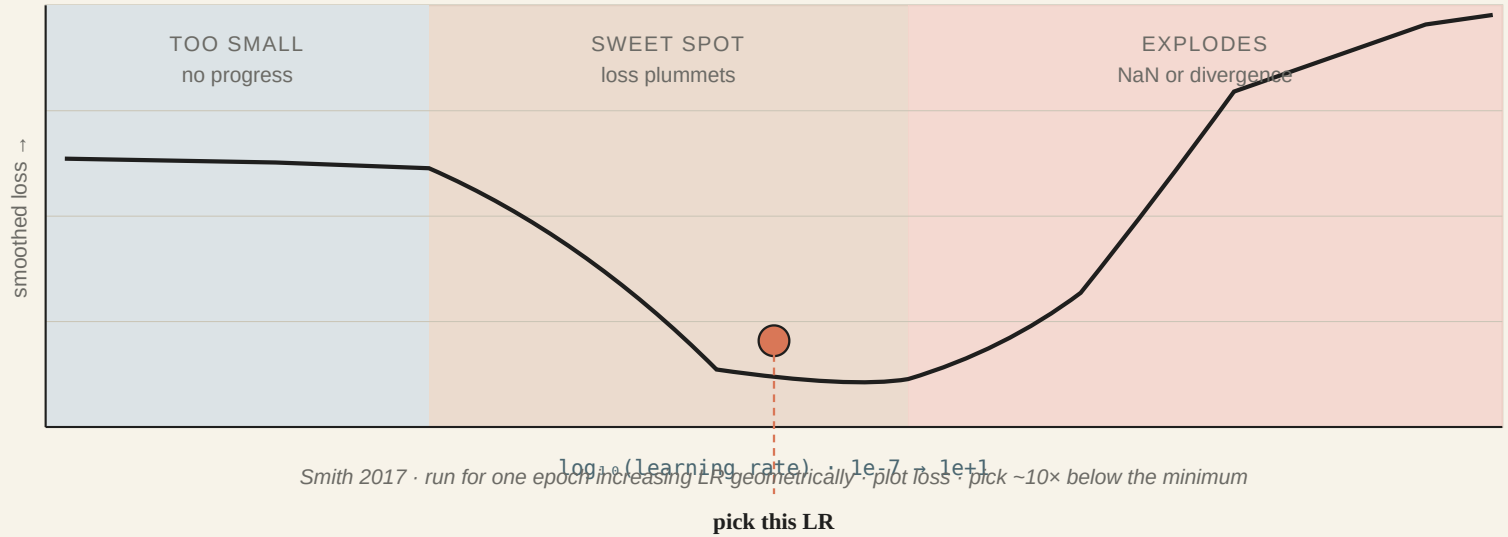
1. Start at LR = 10^{-7}.
2. Each mini-batch → forward, backward, step.
3. After each step, **multiply LR by ~1.05**.
4. Record loss at each step.
5. Plot loss vs. LR (log-x).

How to read the plot:

- Find the region where loss **drops fastest** (the steepest downward slope).
- Loss eventually shoots back up — that's where training is unstable.
- **Pick an LR one order of magnitude before the minimum**, in the steep-descent zone.

Rung 5 · the learning-rate finder

The LR finder — sweep from tiny to huge, watch where loss falls fastest



Smith 2017 · run for one epoch increasing LR geometrically · plot loss · pick ~10× below the minimum

Worked numeric · reading the LR plot

Suppose the LR finder gives:

LR	10^{-4}	10^{-3}	10^{-2}	10^{-1}	1.0
Loss	3.2	2.5	**1.1** (steepest drop)	0.9 (minimum)	2.8 (diverging)

Analysis. Minimum loss is at LR $= 0.1$ — **don't pick this**, it's already on the edge of diverging. The steepest descent is at LR $\approx 10^{-2}$. Rule of thumb: pick **one order of magnitude before the minimum**.

Good starting LR · 10^{-2}. Use this as `max_lr` for one-cycle training.

Full recipe · the 7 rungs

1. **Inspect the data.** Shapes, dtypes, label balance, a few random samples.
2. **Dumb baseline.** Predict-the-mean / most-frequent-class.
3. **Overfit one batch.** Loss $\to \approx 0$, or fix the bug.
4. **Small subset.** 1–5% of data; full pipeline sanity.
5. **LR finder.** Pick a sensible learning rate.
6. **Full run + regularize.** Weight decay, augmentation, dropout.
7. **Error analysis.** Which examples fail, and why?

Never skip a rung. If rung 3 fails, debug at rung 3 — don't go tune hyperparams at rung 6.

Ablation discipline

When improving a model, change one thing at a time.

RUN	CHANGE	VAL METRIC	INTERPRETATION
A	baseline	82.0	reference
B	stronger augmentation	84.1	likely useful
C	bigger model	82.4	small gain, more cost
D	augmentation + bigger model	84.0	bigger model added little

Rules:

- Keep the data split fixed.
- Log the config, seed, commit, and metric.
- Repeat important comparisons with 3 seeds if the gain is small.
- Compare against the simplest baseline that could work.

KEY IDEA

Without ablations, you do not know which change helped. You only know that the final run was different.

PART 5

Error analysis (Ng style)

After training — what next?

You have a model. Val accuracy is 82%.

POP QUIZ

Q. Which of these is most useful?

(a) Try a bigger model.
(b) Train for more epochs.
(c) Sample 100 val mistakes and categorize them.
(d) Tune the learning rate.

Answer · (c)

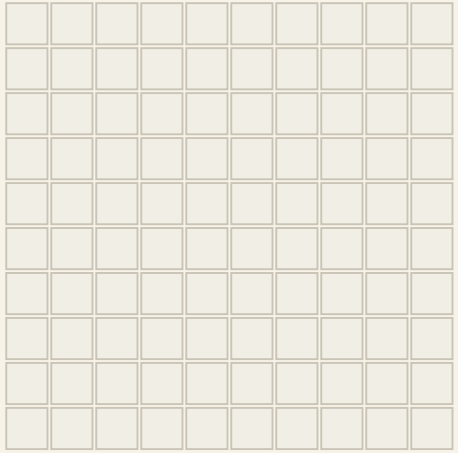
INTUITION

Ng's rule. Before adding complexity, *look at the errors*. Nearly always you will find a dominant failure category — fixing it moves val accuracy far more than architectural churn.

Error analysis · categorize, then prioritize

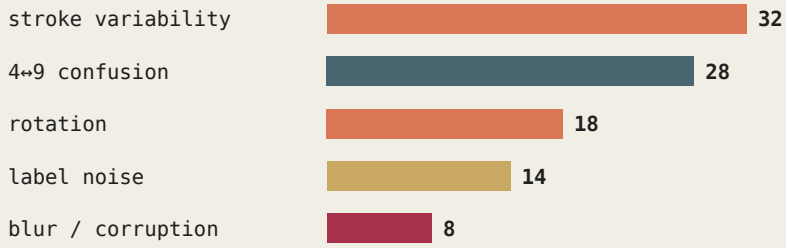
Error analysis — categorize 100 mistakes before you touch the model

100 MISCLASSIFIED VAL EXAMPLES



100 squares · 100 examples

CATEGORIZE → PRIORITIZE BY COUNT



DECISION

stroke + rotation = 50 of 100 → start with rotation / elastic augmentation

From error bucket to intervention

Do not stop at naming the error. Convert the bucket into an action.

ERROR BUCKET	EXAMPLE DIAGNOSIS	INTERVENTION
blurry images	model fails on motion blur	add blur augmentation / collect blur examples
rare class	few training examples	reweight loss / collect targeted data
label ambiguity	humans disagree	clean labels / merge classes / report uncertainty
background shortcut	model uses spurious context	crop, mask, augment backgrounds
threshold error	probabilities okay, decision bad	tune threshold on validation

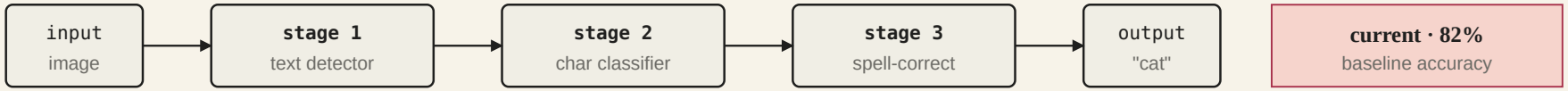
KEY IDEA

Error analysis is not a post-mortem. It is the fastest way to decide what experiment to run next.

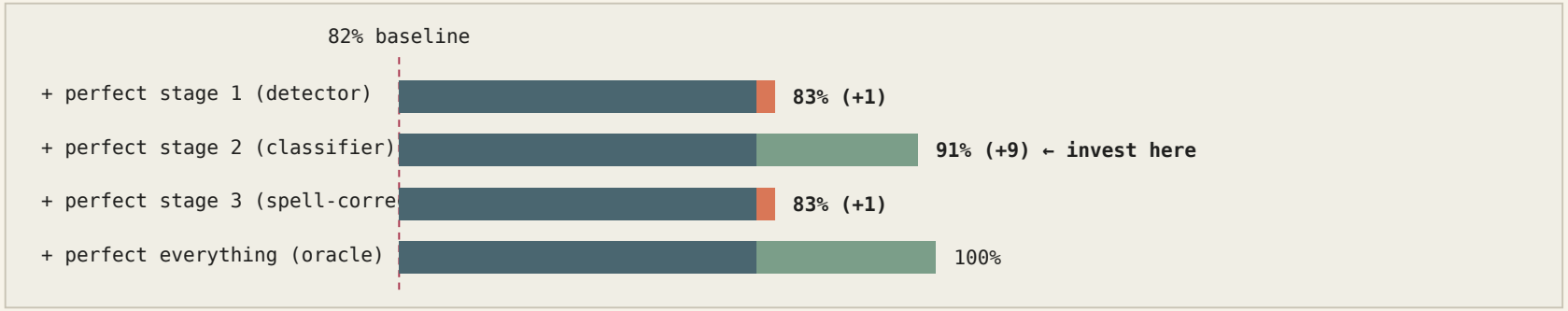
Ceiling analysis — for pipelines

Ceiling analysis — swap each stage for ground truth, measure what you'd gain

YOUR PIPELINE



REPLACE EACH STAGE WITH GROUND TRUTH · RE-MEASURE



*Decision · the **char classifier** is your biggest lever. Do not spend a month on the detector that costs you only +1%.*

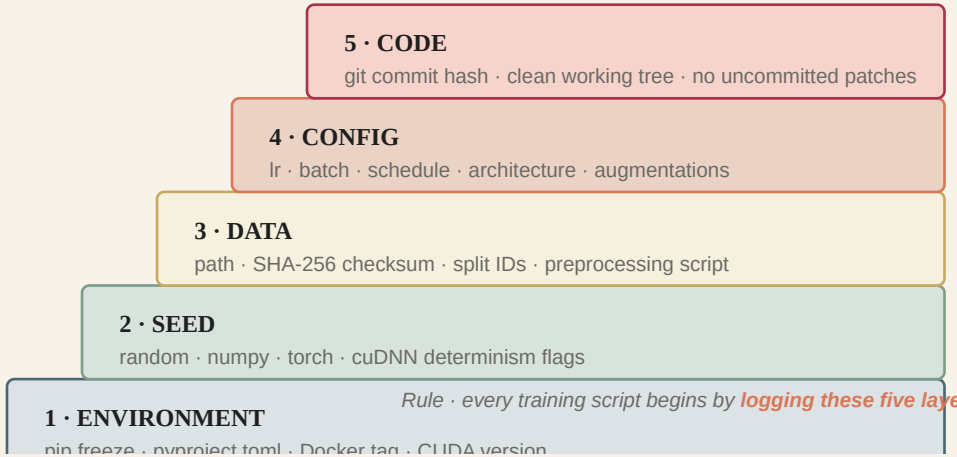
PART 6

Reproducibility

The small things that save you weeks later

Five layers of reproducibility

Five layers of reproducibility · what future-you needs to rerun an experiment



FLAG THESE IN EVERY RUN

- ☐ `git_hash = …`
- ☐ `python --version`
- ☐ `torch.__version__`
- ☐ `CUDA driver version`
- ☐ `seed = 42`
- ☐ `dataset_sha256 = …`
- ☐ `config.yaml → snapshot`
- ☐ `GPU model · driver`
- ☐ `run_id (uuid)`
- ☐ `wandb URL or CSV path`

*Rule · every training script begins by **logging these five layers** before step 1.*

Seeds and determinism · code

```python
import random, numpy as np, torch

def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

# Only if bit-exact reproduction matters (slower):
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark     = False
```

Seed set at the start; every experiment records its seed in the config.

Minimal experiment record

Every serious run should leave enough evidence to reproduce and compare it.

RECORD	WHY IT MATTERS
git commit	exact code
config file	hyperparameters and paths
data version / split id	same examples in train/val/test
random seed	reproducibility and variance estimate
hardware + precision	speed and numeric behavior
final checkpoint + best checkpoint	resume and deploy

```
run = {
    "commit": git_sha,
    "seed": seed,
    "config": cfg,
    "best_val": best_val,
}
```

If you cannot compare two runs later, the experiment did not really happen.

Putting it all together · the L03 master sentence

DERIVATION

Training a deep net is 10% picking an architecture and 90% running a disciplined loop. PyTorch is just an engine; the wins come from the **data pipeline**, the **debugging ladder**, and **error analysis** — none of which require new theory.

STAGE	THE DISCIPLINE	IF YOU SKIP IT
Build	`nn.Module` + `nn.Parameter` registration	silent missing parameters
Feed	`DataLoader` w/ workers + pin_memory	GPU sits idle
Train	forward · loss · zero_grad · backward · step	rotting gradients
Debug	overfit-1-batch → LR finder → ablation	weeks of red herrings
Analyze	bucket errors before scaling	scale the wrong thing

The single insight · **the bug almost never lives where students look first.** Hence the *ladder*.

Pop quiz · revisit

The flat-loss puzzle from the start? The right move is **(c) overfit one batch**.

KEY IDEA

If a model can't drive loss → 0 on 4 examples, the bug is **not** the LR, the optimizer, or the depth — it's the wiring. Until you climb that rung, every other tweak is guessing.

This is the single most important habit in this lecture.

Practice problems

DERIVATION

P1. You change `batch_size` from 32 to 256 but keep the LR fixed. Training diverges. Why? Name the standard rule for scaling LR with batch size.

P2. A `DataLoader` with `num_workers=0` and `pin_memory=False` is feeding a GPU running at 30% utilization. Name two changes that should help and the order in which you'd test them.

P3. Show that gradient accumulation over K micro-batches with per-step gradient g_k gives the **same** update as one large batch of size $K \cdot B$, *provided* you scale the loss by $1/K$.

P4. Your run has training loss 0.02, validation loss 0.6. Diagnose. Name three interventions in order of cost.

P5. Why does `model.eval()` differ from `torch.no_grad()`? Give a concrete example where you need both.

P6. A grad-clip of `max_norm=1.0` is applied to a 100M-parameter model. Show that this is **not** the same as clipping per-parameter. Which version do you want when sigmoid layers are blowing up only in early layers?

Summary · Lecture 3 — summary

- `nn.Module` auto-registers parameters; use `nn.Parameter` / `register_buffer` explicitly.
- **Autograd** is a dynamic tape built every forward; wrap eval in `torch.no_grad()`.
- **DataLoader tuning** — `num_workers`, `pin_memory`, `persistent_workers`.
- **Mixed precision (BF16)** — 2× speed, half memory. Default on Ampere+.
- **Loss ≠ metric** — pick the metric that matches the real objective.
- **Debug ladder** — never skip a rung. Overfit one batch first.
- **Validation discipline** — prevent leakage and check distribution shift.
- **Ablations** — change one thing at a time and log the comparison.
- **Error analysis (Ng)** — categorize failures *before* scaling up.
- **Reproducibility** — code + config + data + seed + env. Weeks saved.

Read before Lecture 4

Prince — Ch 6 (fitting models), Ch 8 (measuring performance). Free at udlbook.github.io.

Next lecture

Optimization properly — loss landscapes, momentum, Nesterov.

NOTEBOOK

Notebook 3a · `03a-training-recipe.ipynb` — full training loop with checkpointing, LR finder