Regularization in Deep Learning

Lecture 6 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

This lecture is comprehensive — Part 1 covers classical + data-centric regularization; Part 2 covers dropout and normalization. Intended to span two sessions.

Learning outcomes

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

1. Explain **double descent** and when it happens.
2. Pick among **L2 / early stopping / augmentation** by regime.
3. Apply **Mixup / CutMix / label smoothing** correctly.
4. Implement **dropout** (including inverted scaling) from scratch.
5. Choose among **BN / LN / GN / RMSNorm** by architecture.
6. Explain **pre-norm vs post-norm** trade-offs for depth.

Recap · where we are

- **Architecture** — ResNets, He init, ReLU (L2).
- **Optimizer** — AdamW with warmup + cosine (L4–L5).
- **Recipe** — debug ladder, error analysis (L3).

> REFERENCE
>
> Today maps to **UDL Ch 9** (Regularization) and the BatchNorm parts of **Ch 11** (Residual Networks). ES 654 covered ridge/LASSO — we skim those and focus on what's new.

Plan for the two sessions

Session 1 · classical & data-centric

1. Double descent — the modern bias-variance picture
2. L2, L1, early stopping (brisk — prereq covered)
3. Data augmentation
4. Mixup & CutMix
5. Label smoothing

Session 2 · architectural

6. Dropout
7. BatchNorm — mechanics, ICS debate
8. LayerNorm — why sequences differ
9. RMSNorm — the modern simplification
10. Pre-norm vs post-norm placement

Two students · the regularization story

KEY IDEA

Student A · memorizes the 100 practice problems. Aces the practice test. Fails the real exam (different problems).

Student B · learns the method. Doesn't memorize · understands. Does fine on both.

Regularization is how we force our model to be Student B.

Without it · a deep network has more than enough capacity to memorize the training set perfectly while learning nothing transferable. With it · the model is encouraged to find patterns that hold beyond the training data.

What's new in DL regularization vs classical ML

You already know (ES 654)

- L2 / ridge
- L1 / LASSO
- Cross-validation
- Bias-variance tradeoff

We **skim** these.

New for DL

- Double descent
- Data augmentation (images, text)
- Mixup / CutMix
- Label smoothing
- Early stopping as regularization
- Dropout, BatchNorm, LayerNorm, RMSNorm

We spend time here.

SESSION 1 · PART 1

Double descent

Classical bias-variance, revisited

The classical textbook picture

From ES 654 you know the U-curve:

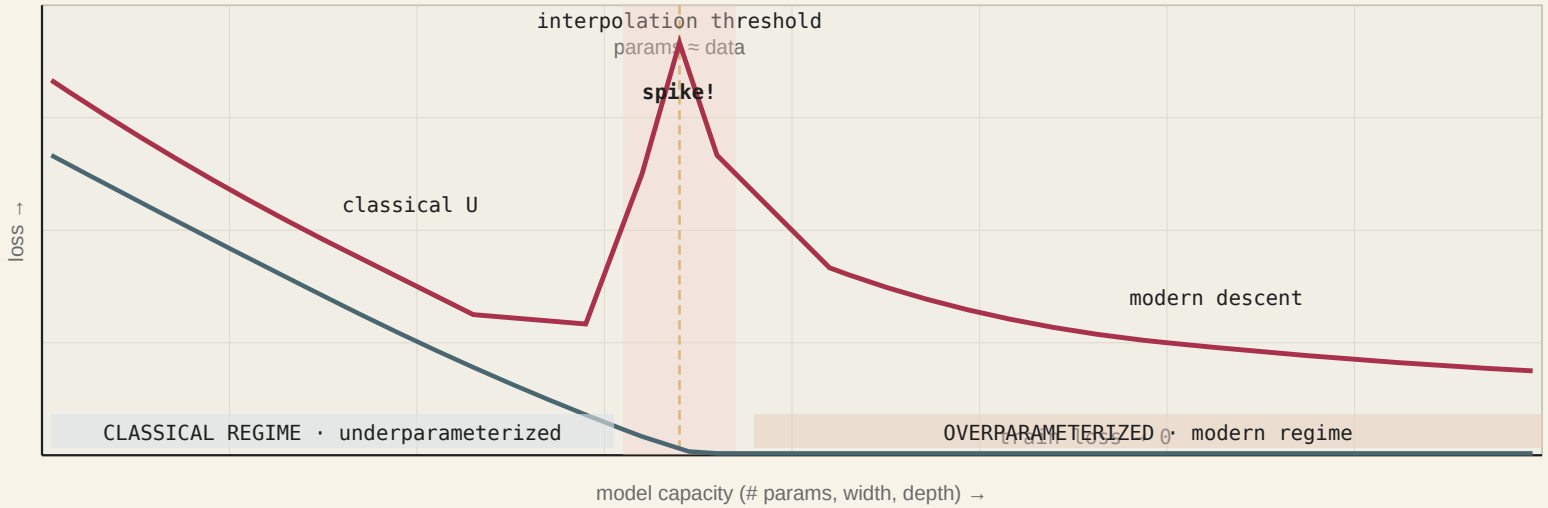
- **Too simple** → underfit (high bias)
- **Too complex** → overfit (high variance)
- **Sweet spot** in the middle

  For 50 years ML chose "the middle" via cross-validation. Done.

  Q. But modern nets have 10^6–10^{12} parameters for datasets of 10^6 examples. We should be in catastrophic-overfit territory. Why aren't we?

Double descent · the 2019 surprise

Double descent — classical U, then a second descent past the interpolation threshold



Belkin et al. 2019. *More params past the threshold does not hurt — it helps. This is why 10^9-param models generalize.*

What's actually happening

Three regimes:

1. **Classical underparameterized** (params ≪ data): U-curve as expected.
2. **Interpolation threshold** (params ≈ data): test error **spikes**.
3. **Modern overparameterized** (params ≫ data): test error **drops again**.

KEY IDEA

More parameters past the threshold *does not hurt*. Implicit regularization from SGD + overparameterization finds flat, generalizing minima.

This is one of the big open questions in DL theory. Prince Ch 20 — *"Why does deep learning work?"*

Practical implication

You don't need to shrink a model to generalize. You can just **go bigger** and rely on:

- **Implicit regularization** from SGD
- **Explicit regularization** (today's topic)
- **Massive data** (when available)

  ResNet-50 has 25M params; modern LLMs have 10^{11}. Both generalize fine *because* they are past the interpolation threshold.

SESSION 1 · PART 2

L2, L1, early stopping

Brisk — you know these from ES 654

L2 · the weight-leash analogy

INTUITION

Analogy. The data loss is your **dog**, trying to run toward an interesting smell (the optimum on the training data).

The L2 penalty is a **leash** pulling the dog back toward you (toward zero). The actual update is a **compromise** — the dog moves toward the smell, but the leash keeps it from running too far.

L2 · derive the gradient

Total loss = data loss + L2 penalty:

$$L_{\text{total}} = L_{\text{data}} + \frac{\lambda}{2} w^2$$

Differentiate term by term:

$$\frac{dL_{\text{total}}}{dw} = \frac{dL_{\text{data}}}{dw} + \underbrace{\frac{d}{dw}\left(\frac{\lambda}{2} w^2\right)}_{= \lambda w}$$

Plug into the SGD update rule $w_{\text{new}} = w - \eta\,(\text{grad})$:

$$w_{\text{new}} = w - \eta \left(\frac{dL_{\text{data}}}{dw} + \lambda w \right)$$

The extra term $-\eta\lambda w$ is the **leash pull** toward zero.

Worked numeric · L2 single-weight update

$w = 2.0$, $dL_{\text{data}}/dw = 1.5$, $\eta = 0.1$, $\lambda = 0.01$.

Without L2

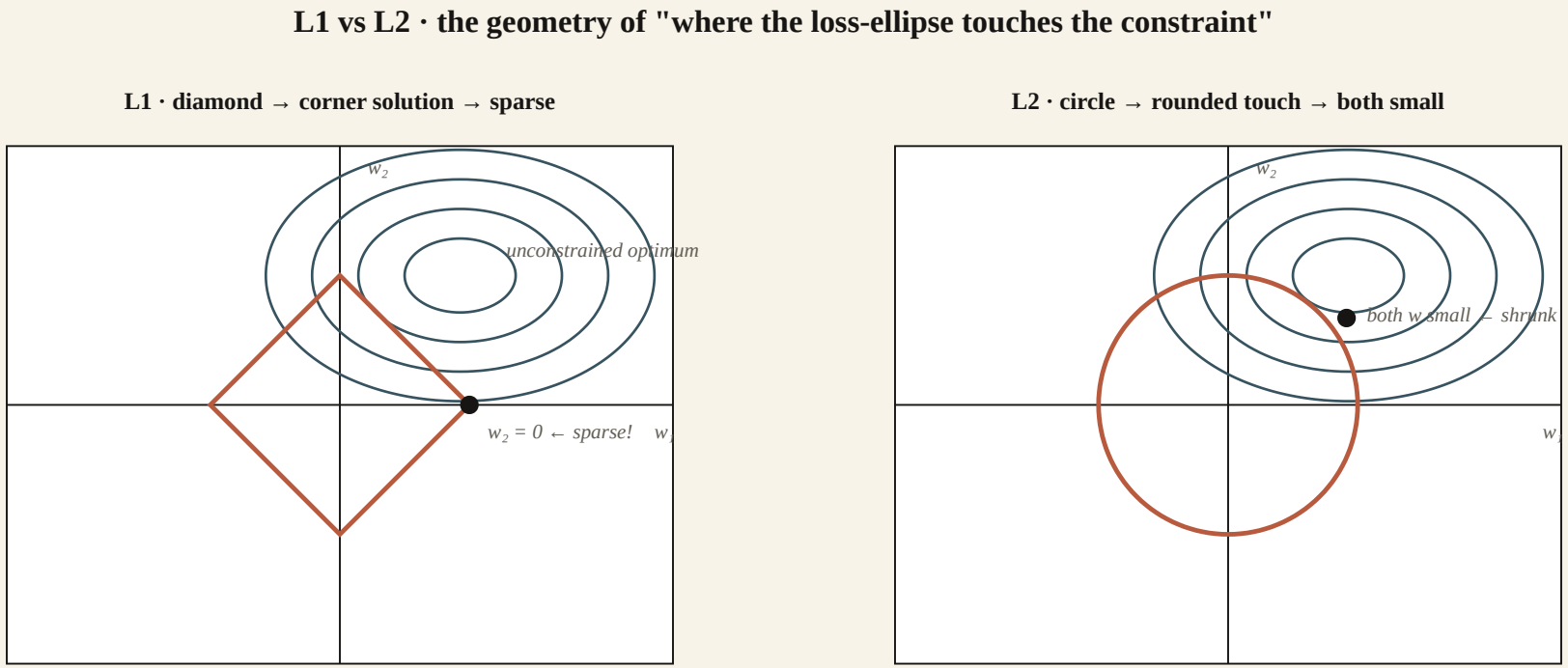
$w_{\text{new}} = 2.0 - 0.1 \cdot 1.5 = \mathbf{1.85}$

With L2

- L_2 contribution: $\lambda w = 0.01 \cdot 2.0 = 0.02$
- Full gradient: $1.5 + 0.02 = 1.52$
- $w_{\text{new}} = 2.0 - 0.1 \cdot 1.52 = \mathbf{1.848}$

The weight ends up **slightly smaller** — decayed toward zero. Repeat over thousands of steps · big weights shrink unless the data loss really wants them.

L1 vs L2 · the geometry



L1's diamond corners hit the axes → coordinates become exactly 0. L2's circle has no corners → coords just shrink.

L2 = weight decay · regrouping the update

Take the SGD-with-L2 step:

$$w_{\text{new}} = w - \eta \frac{dL_{\text{data}}}{dw} - \eta\lambda w$$

Group the two w terms:

$$w_{\text{new}} = (1 - \eta\lambda)\, w \; - \; \eta \frac{dL_{\text{data}}}{dw}$$

The factor $(1 - \eta\lambda)$ is **slightly less than 1**. So at every step, the weight is first **shrunk a little** (decayed), then updated by the data gradient.

That's why "L2 regularization" is the same thing as "**weight decay**".

IN PRACTICE

In PyTorch · `AdamW(..., weight_decay=0.1)` is the one line you need. (For why decoupling matters in adaptive optimizers, see L5.)

L1 · the sparsity-inducing sibling

Add $\lambda \|\theta\|_1$ → encourages many weights to be exactly 0.

	L1	L2
Prior	Laplace	Gaussian
Solution	sparse (many zeros)	small (everything shrinks)
Use in DL	rare	ubiquitous

DL rarely uses L1 — features are *distributed* across many weights, not localized in a few. L1's sparsity breaks distributed representations.

Early stopping as implicit regularization

Q. If val loss starts rising at epoch 30, why stop training?

Because continuing means:

- More optimization steps → more capacity effectively used
- Same function class, but walking further into the parameter landscape
- Eventually you'll memorize training noise

> KEY IDEA
>
> Early stopping = an implicit form of capacity control. It's **free** and almost always helps. Every serious training script checkpoints on val loss.

Covered in Lecture 1 — the curve with the "best val" marker.

SESSION 1 · PART 3

Data augmentation

The single highest-value regularizer for vision

Why augmentation is powerful

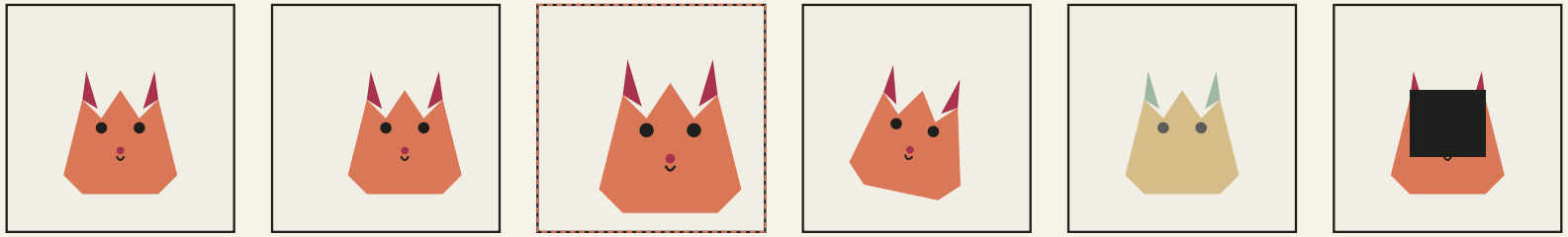
INTUITION

Classical regularization constrains the *model*. Data augmentation constrains the *data* — by telling the model what invariances it should respect.

A flipped cat is still a cat. A color-jittered cat is still a cat. By training on the augmented versions, the model **learns** that these transformations should not change the prediction.

Standard vision augmentations

Data augmentation — same label, more training examples, built for free



original	horizontal flip	random crop	rotate ±15°	color jitter	cutout
label = "cat"	label unchanged	scale + translate	orientation-invariant	hue · brightness · saturation	force reliance on partial info

PYTORCH · TORCHVISION.TRANSFORMS

```
train_tfm = transforms.Compose([
transforms.RandomResizedCrop(224), transforms.RandomHorizontalFlip(),
transforms.ColorJitter(0.2, 0.2, 0.2), transforms.ToTensor(),
transforms.RandomErasing(p=0.25), transforms.Normalize(mean, std)])
```

*Key idea · every augmentation should **preserve the label**. A flipped cat is still a cat; a flipped "6" is a "9" (so don't flip MNIST).*

Which augmentations · which problem

DOMAIN	USEFUL	AVOID
Natural images	flip · crop · color jitter · rotate	vertical flip (changes sky/ground)
Medical imaging	small rotations · mild intensity	flips (mirrors anatomy)
MNIST / digits	small rotations · elastic	flip (6 ↔ 9)
Satellite imagery	all rotations (no "up") · flip	color jitter (semantic)
Text	synonym · back-translation · mask	random char shuffle

Rule · augmentation must **preserve the label**. If not, it's noise, not signal.

Advanced · RandAugment, AutoAugment

Instead of hand-picking augmentations:

- **AutoAugment** (2018) — learn the augmentation policy from data.
- **RandAugment** (2020) — pick N augmentations uniformly; pick magnitude M uniformly. Two knobs.

In `torchvision`:

```
transforms.RandAugment(num_ops=2, magnitude=9)
```

IN PRACTICE

RandAugment is the 2026 default for vision pre-training. Two-line change, consistent +1–3% accuracy.

SESSION 1 · PART 4

Mixup & CutMix

Augment the *label*, not just the input

The idea

Standard augmentation: one image → one transformed image, same label.

Mixup and CutMix go further: *combine two images* and *interpolate their labels* correspondingly.

Mixup and CutMix in one picture

Mixup and CutMix — interpolate inputs and labels

MIXUP · PIXEL-WISE BLEND

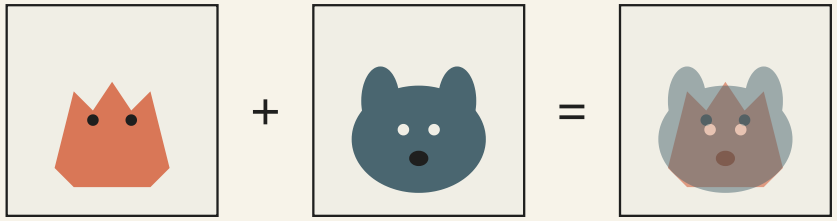


image A
"cat"

image B
"dog"

mixed image
label = (0.7, 0.3)

$$\tilde{x} = \lambda \cdot x_A + (1-\lambda) \cdot x_B$$
$$\tilde{y} = \lambda \cdot y_A + (1-\lambda) \cdot y_B$$
$\lambda \sim \text{Beta}(\alpha, \alpha)$, $\alpha = 0.2$

CUTMIX · SPATIAL PATCH REPLACEMENT

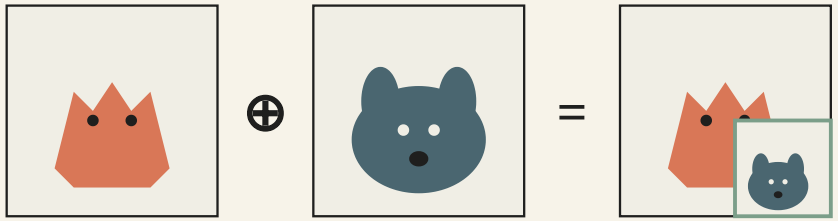


image A
"cat"

image B
"dog"

pasted image
label = (0.8, 0.2)

WHY THIS HELPS

- ✓ smooths the decision boundary
- ✓ forces model to consider whole image
- ✓ regularizer + improves calibration
- ✓ no extra data needed

Why Mixup and CutMix work

Three observations:

1. **Decision boundary smoothing.** The model sees "half-cat half-dog" examples with mixed labels → boundary becomes smoother, not piecewise-constant.
2. **Implicit regularizer.** Forces calibration — the output distribution has to reflect the interpolation.
3. **Free data.** No manual annotation; the mixing happens inside the training loop.

Empirically: Mixup/CutMix adds ~1–2% CIFAR-10 accuracy. Essentially a free win for vision.

Mixup · the smoothie analogy

INTUITION

Analogy. Standard augmentation = slightly reshape a "cat" fruit. **Mixup** = put 70% cat + 30% dog into a blender → a smoothie that is *neither fully cat nor fully dog*.

Crucially, **the label is also a smoothie**: "70% cat, 30% dog". This forces the model to learn that predictions can live **between classes** → smoother decision boundary.

Mixup · the math, step by step

Mix two examples (x_1, y_1) and (x_2, y_2):

1. **Pick a mixing ratio** $\lambda \sim \mathrm{Beta}(\alpha, \alpha)$, $\lambda \in (0, 1)$. Say $\lambda = 0.7$.
2. **Mix inputs.** $x_{\text{mix}} = \lambda\, x_1 + (1 - \lambda)\, x_2 = 0.7\, x_1 + 0.3\, x_2$.
3. **Mix labels.** $y_{\text{mix}} = \lambda\, y_1 + (1 - \lambda)\, y_2$.
4. **Mix the loss** equivalently:

$$\text{loss} = \lambda\, \mathrm{CE}(\text{logits}, y_1) + (1 - \lambda)\, \mathrm{CE}(\text{logits}, y_2)$$

Worked numeric · Mixup label

Cats = class 0, dogs = class 1. One-hot:

- $y_1 = [1, 0]$ (cat)
- $y_2 = [0, 1]$ (dog)
- $\lambda = 0.7$

$$y_{\text{mix}} = 0.7 \cdot [1, 0] + 0.3 \cdot [0, 1] = [0.7, 0.3]$$

The model now sees a faded cat overlaid with 30%-opacity dog, and must produce $[0.7, 0.3]$ to minimize the loss. **Calibrated outputs** for free.

Mixup in PyTorch · 10 lines

```
def mixup_batch(x, y, alpha=0.2):
    lam = np.random.beta(alpha, alpha)
    idx = torch.randperm(x.size(0))
    x_mix = lam * x + (1 - lam) * x[idx]
    y_a, y_b = y, y[idx]
    return x_mix, y_a, y_b, lam

# in training step
x_mix, y_a, y_b, lam = mixup_batch(x, y)
logits = model(x_mix)
loss = lam * criterion(logits, y_a) + (1 - lam) * criterion(logits, y_b)
```

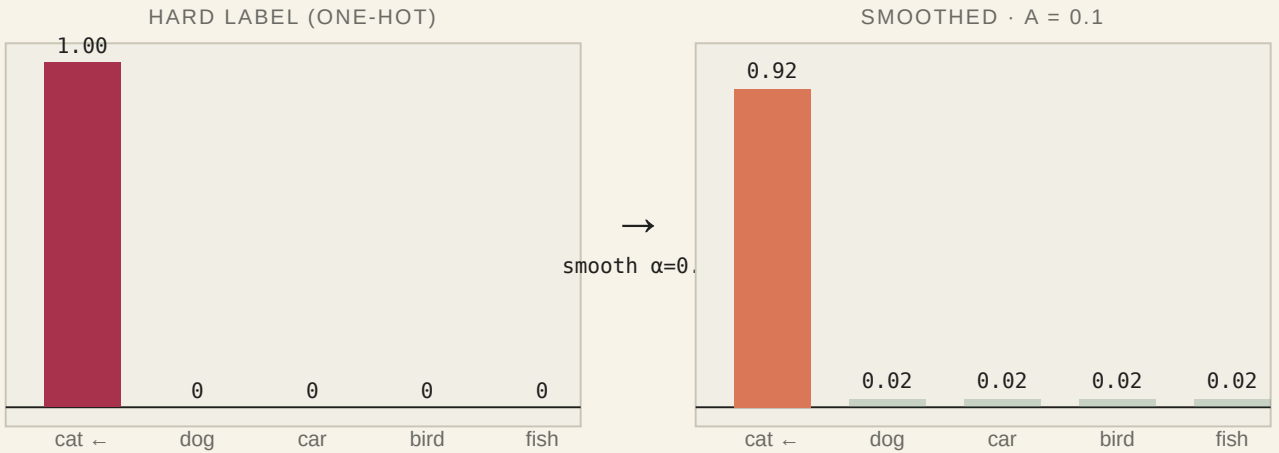
SESSION 1 · PART 5

Label smoothing

Because "1.0 for the right class" is a lie

Hard vs soft targets

Label smoothing — soft targets instead of one-hot



FORMULA

$$y_{smooth} = (1-\alpha)\cdot y_{hard} + \alpha/K$$

WHY

- prevents overconfidence
- better calibration
- slight test-acc boost
- free · 1 line of code

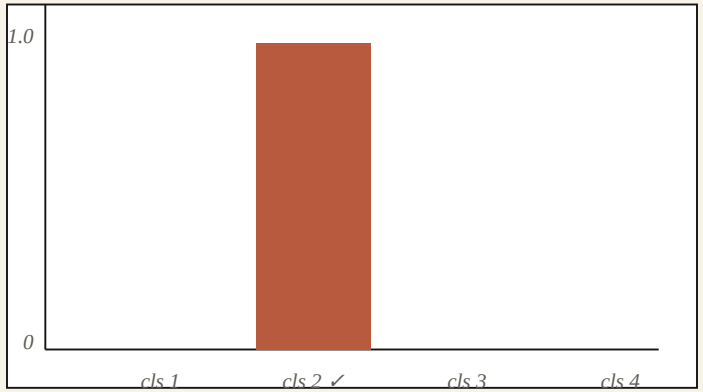
PYTORCH (SINCE 1.10)

```
criterion = nn.CrossEntropyLoss(label_smoothing=0.1) # that's the entire API change
```

Hard target vs soft target · in bars

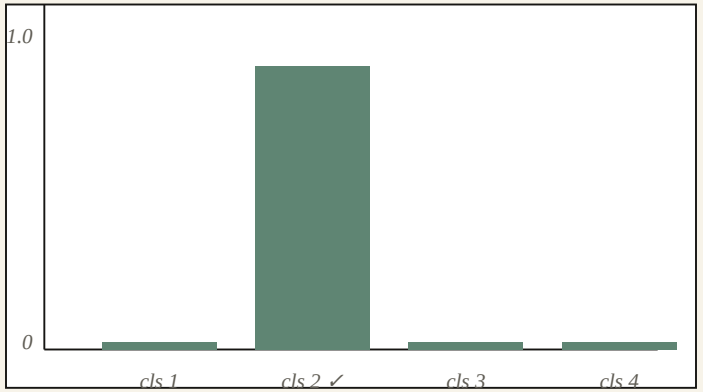
Label smoothing · soften the one-hot target so the model can't get overconfident

Hard target · one-hot



`y = [0, 1, 0, 0]`

Soft target · α = 0.1



`y = [0.025, 0.925, 0.025, 0.025]`

In the loss, soft target penalizes "very low" logits for wrong classes too · prevents model from being too sharp

model can drive cls 2 logit → ∞ · cls 2 logit can't go to ∞ · target is bounded

Why soften the labels?

INTUITION

Analogy · the humble professor. A bad professor: *"The answer is A. Memorize it."* → encourages overconfidence.
A good professor: *"It's very likely A — but reserve some confidence for being wrong."* → calibrated thinking.

Label smoothing is the good professor for your neural network.

Label smoothing · derive the formula

Goal · take a tiny fraction α of confidence away from the correct class and spread it evenly across all K classes.

1. **Hard one-hot label** for class 3 ($K = 10$):
   $y_{\text{hard}} = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$
2. **Uniform label** (complete uncertainty):
   $y_{\text{uniform}} = [0.1, 0.1, \ldots, 0.1]$ (each $= 1/K$)
3. **Mix them** with weight $\alpha = 0.1$:
   $$y_{\text{smooth}} = (1 - \alpha)\, y_{\text{hard}} + \alpha\, y_{\text{uniform}}$$
   $$= 0.9\, y_{\text{hard}} + 0.1\, y_{\text{uniform}}$$
4. **Compute** entry-by-entry:
   $y_{\text{smooth}} = [0.01, 0.01, \mathbf{0.91}, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]$
   Sum is still 1. The compact form:

$$y_{\text{smooth}} = (1 - \alpha)\, y_{\text{hard}} + \frac{\alpha}{K}$$

Why label smoothing helps

1. **Prevents overconfidence.** Hard labels push logits to $\pm\infty$ → miscalibrated model.
2. **Regularizes the output layer.** Softer target → smaller logit magnitudes.
3. **Label noise robustness.** Some labels are wrong anyway — smoothing acknowledges that.

> IN PRACTICE
>
> One flag in PyTorch: `CrossEntropyLoss(label_smoothing=0.1)`.

SESSION 2

Architectural regularization

Dropout + Normalization

Session 2 · what we will cover

Two architectural regularizers that ship inside the network:

1. **Dropout** — randomly silence neurons during training.
2. **Normalization** (BatchNorm / LayerNorm / RMSNorm) — re-center and re-scale activations.
3. **Placement** — pre-norm vs post-norm.

All 2026-relevant. All in every modern architecture.

SESSION 2 · PART 6

Dropout

An implicit ensemble, one line of code

The idea (Hinton 2012)

Every forward pass during training:

1. Sample a random binary mask $\mathbf{m}_i \sim \text{Bernoulli}(p)$ for each hidden unit i.
2. Multiply: $\mathbf{h}_{\text{drop}} = \mathbf{h} \odot \mathbf{m}$.
3. Scale surviving units by $1/p$ to keep expected activation the same.

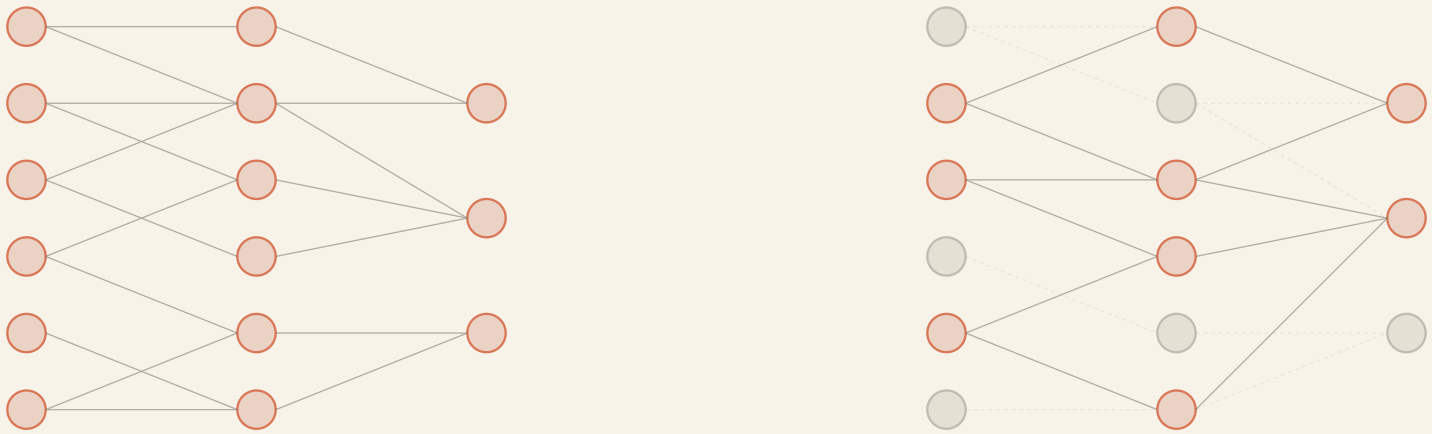
At eval time, **turn dropout off** — use all units.

Dropout in one picture

Dropout — randomly silence p × N neurons per batch, effectively training an ensemble

FULL NETWORK · ALL NEURONS ALIVE

WITH DROPOUT · P = 0.5 · RANDOM BATCH MASK



`training · h ← (h ⊙ mask) / p eval · h unchanged (dropout off)`

"Inverted dropout" keeps the expected activation constant between train and eval.

IN PRACTICE

▶

Interactive: slide `p` and watch a small network flicker; toggle train/eval mode — dropout-playground.

Two intuitions for why it helps

Ensemble view

Each mini-batch trains a *different* sub-network (a subset of units).

Over many batches you are implicitly training an **ensemble of 2^N thinned networks**, all sharing weights.

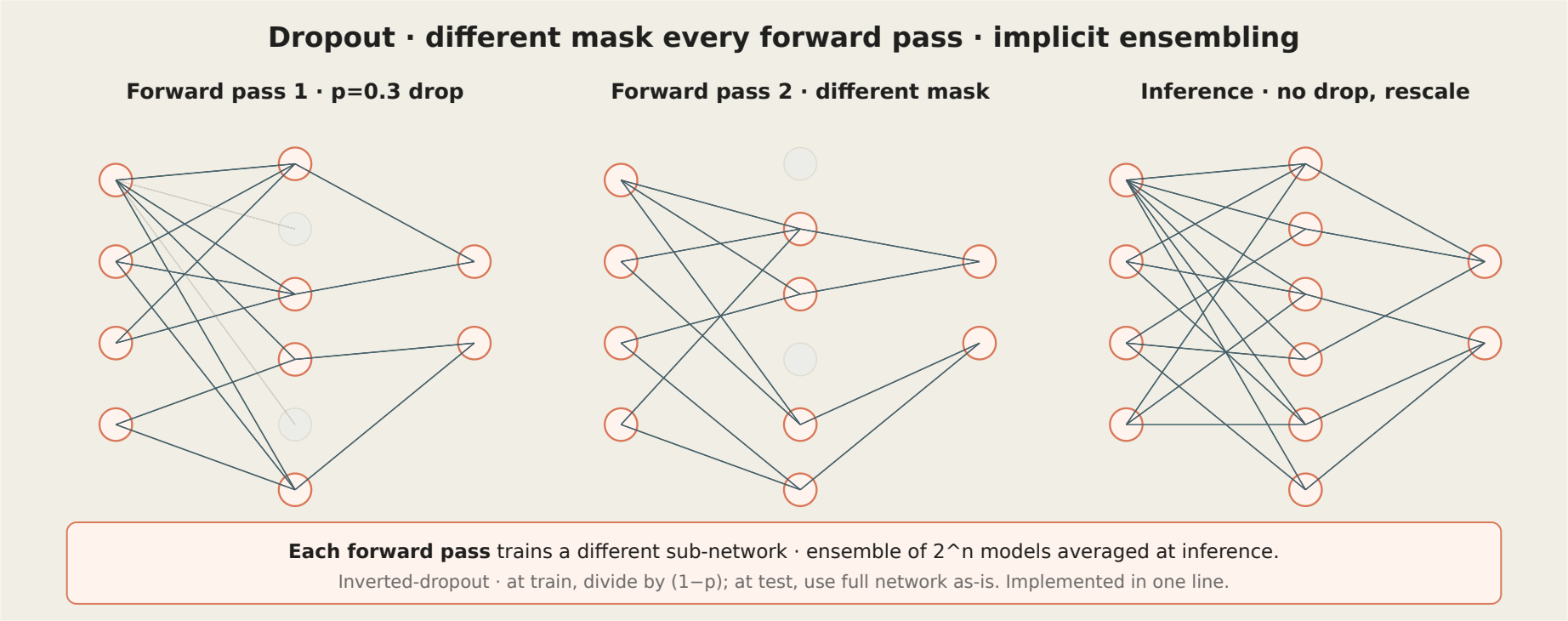
At test time, no mask → like averaging the ensemble.

Co-adaptation view

Without dropout, units **co-adapt** — neuron j relies on neuron k being alive to do its job.

Dropout forces every unit to be useful *on its own* → more distributed representation.

Dropout · different masks per pass



Inverted dropout · the part-time-team analogy

INTUITION

Analogy. A 4-person construction crew. Some days, randomly, only 2 show up.

- **Bad:** they each work normally → only half a wall built. Manager learns the wrong baseline.
- **Good:** with keep-prob $p = 0.5$, the present workers each work $1/p = 2\times$ harder → wall finished. Manager's expectation stays correct.

On the **final project day** (test time), all 4 show up — no scaling needed. Their normal pace = correct expectation.

Inverted dropout · why divide by p

For a single neuron with activation h, keep-prob p.

Test time: dropout off → output is just h. *(target expectation)*

Training time: output h_{drop} is random:

- with probability p: scaled active output $= h/p$
- with probability $(1-p)$: dropped, output $= 0$

Expected output during training:

$$E[h_{\text{drop}}] = p \cdot (h/p) + (1-p) \cdot 0 = h$$

✓ same as test time. The rest of the network sees the **same expected activations** in both modes — training and eval behave consistently. This is **inverted dropout**.

$$\mathbf{h}_{\text{drop}} = \frac{\mathbf{h} \odot \mathbf{m}}{p}, \quad \mathbf{m} \sim \text{Bernoulli}(p)$$

Dropout · worked numeric example

Suppose hidden activations · `h = [2.0, 1.5, 0.5, 3.0]` and we use `p = 0.5` keep-prob.

DERIVATION

Train pass. Sample mask `m = [1, 0, 1, 0]` (Bernoulli p=0.5).
$\mathbf{h}_{\text{drop}} = (h \odot m)/p = [2.0, 0, 0.5, 0]/0.5 = [\mathbf{4.0}, 0, \mathbf{1.0}, 0]$
(the kept units are *amplified* to compensate)

Expected value · $E[\mathbf{h}_{\text{drop}}] = p \cdot (h/p) + (1-p) \cdot 0 = h = [2.0, 1.5, 0.5, 3.0]$
↑ same as the no-dropout output.

Eval pass. $\mathbf{h}_{\text{eval}} = h = [2.0, 1.5, 0.5, 3.0]$. No mask, no scaling.

The training-time scaling-up by $1/p$ is what lets us drop the mask at eval time without changing the network's expected output.

Dropout · the basketball-team analogy

KEY IDEA

Imagine training a basketball team where, in any given practice drill, some players randomly sit out. No one can rely too much on the star player · she might not be there.

Result · everyone becomes more versatile. The team performs more reliably with any subset on the court.

That's what dropout does to neurons · it prevents them from **co-adapting** (relying too heavily on a few specific neighbors). Each neuron has to become individually useful.

Dropout in PyTorch

```python
self.drop = nn.Dropout(p=0.1)     # typical: 0.1 for Transformers
                                  #          0.5 for MLP hidden layers
                                  #          0.0 for CNNs (usually)

def forward(self, x):
    h = F.relu(self.fc1(x))
    h = self.drop(h)              # apply after the activation
    h = self.fc2(h)
    return h
```

IN PRACTICE

`model.train()` and `model.eval()` toggle it automatically. Forgetting the mode switch is a classic bug.

WATCH OUT

Convention mismatch warning · in lecture math, p often denotes **keep** probability (Bernoulli(p)). PyTorch's `nn.Dropout(p)` uses p as the **drop** probability. Keep-prob 0.8 ↔ `nn.Dropout(p=0.2)`. Always double-check.

Where dropout lives in 2026

ARCHITECTURE	TYPICAL USE
Large MLPs	`p = 0.5` between hidden layers
CNNs	usually absent (BN + aug is enough)
RNNs / LSTMs	variational dropout (same mask per timestep)
Transformers	`p = 0.1` after attention and FFN
Fine-tuning an LLM	`p = 0.0` or very small — data is scarce

INTUITION

Dropout was the biggest regularization breakthrough of 2012. Today it is overshadowed by BN + LN + augmentation for many tasks, but still in every Transformer.

SESSION 2 · PART 7

Normalization

Same family, three flavours, one knob at a time

Hiker in a canyon · why normalization matters

KEY IDEA

Imagine a hiker descending a long, narrow, steep-sided canyon. They bounce side-to-side, making slow progress along the canyon's length.

A round bowl is much easier · the hiker walks straight to the bottom.

Normalization **reshapes** the loss landscape from a canyon into a bowl · same minimum, much easier optimizer trajectory.

Concretely · BN/LN keep activations centered and unit-scale, which means the loss's curvature in different directions is roughly equal. The optimizer takes confident, direct steps.

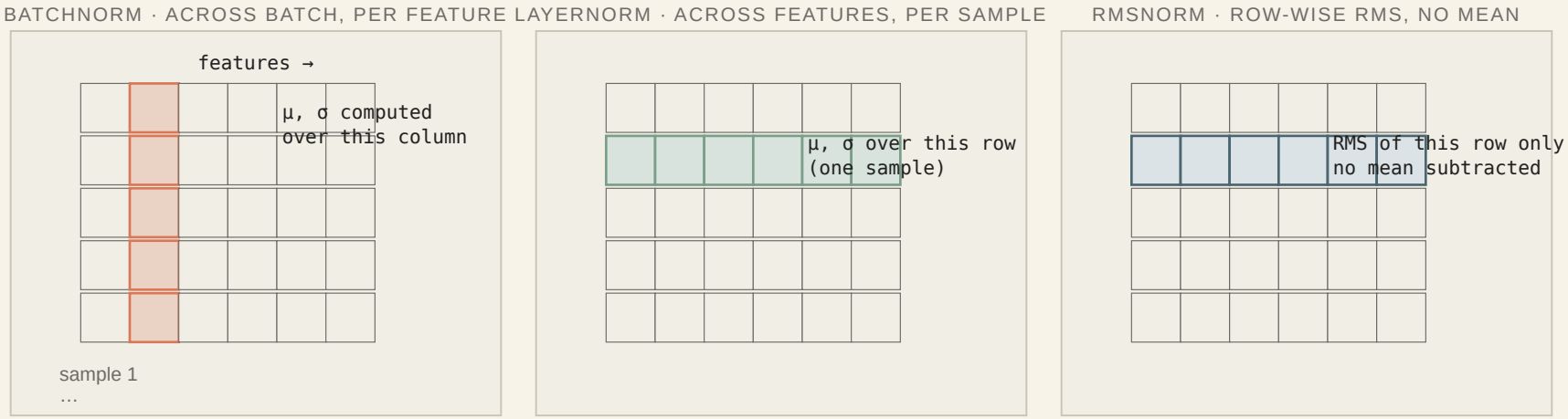
Why normalize at all?

Two problems that normalization fixes:

1. **Scale drift across layers.** Activations grow or shrink with depth. He init addresses this at *init*; normalization does it at *every step*.
2. **Internal covariate shift (original claim).** Distribution of layer inputs changes during training. This explanation turned out to be partly wrong — see next slide.

BN · LN · RMSNorm · the axes

BatchNorm vs LayerNorm vs RMSNorm — same input tensor, three different axes



BN

$\hat{x} = (x - \mu_{batch}) / \sqrt{\sigma^2_{batch} + \varepsilon}$

stats over N samples

→ CNNs

LN

$\hat{x} = (x - \mu_{sample}) / \sqrt{\sigma^2_{sample} + \varepsilon}$

stats over D features

→ Transformers

RMSNORM

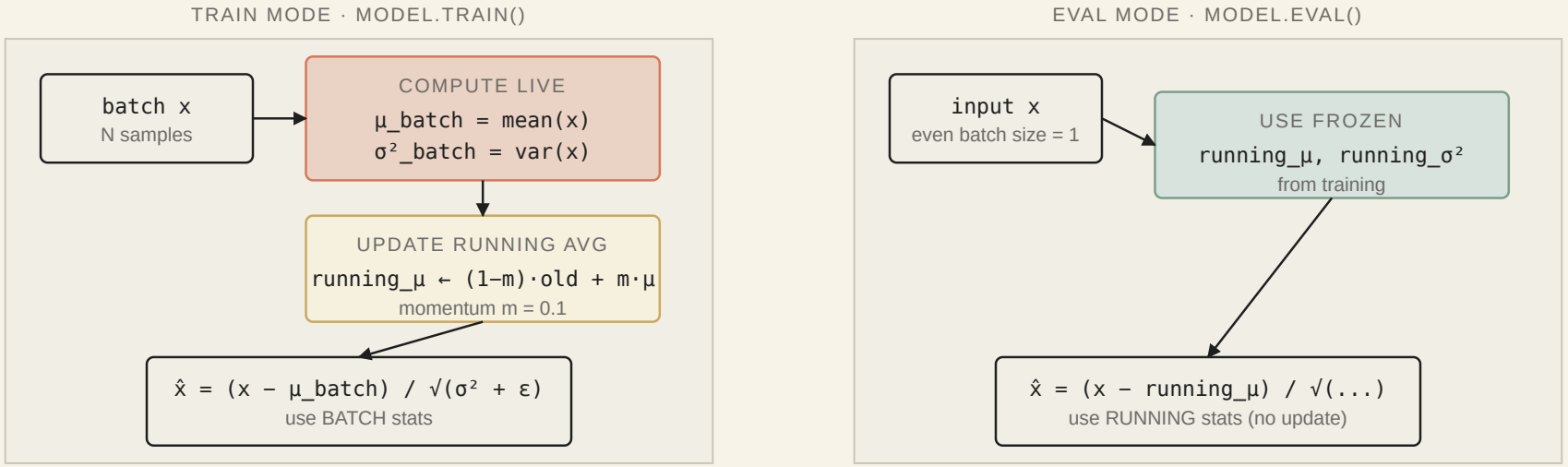
$\hat{x} = x / \sqrt{\text{mean}(x^2) + \varepsilon}$

no mean, cheaper

→ modern LLMs

BatchNorm · train vs eval modes

BatchNorm has two modes — toggle with `model.train()` / `model.eval()`



COMMON BUG

Forgot model.eval() at inference? BN uses batch-of-1 statistics → predictions unstable and often wrong.

BatchNorm · standardizing exam scores

INTUITION

Analogy. Student A's homework: $[80, 85, 90]$. Student B's: $[6, 7, 8]$ (1–10 scale). The next layer sees raw scores and is confused by the scale gap.

BatchNorm is a fair grading TA:

1. **Centre** — subtract each student's mean. A → $[-5, 0, 5]$, B → $[-1, 0, 1]$.
2. **Rescale** — divide by std. Now both have unit scale.
3. **Re-learn the right scale.** Maybe centre 0 / scale 1 *isn't* ideal for the next layer. BN adds learnable γ and β so the network can pick the best post-norm scale.

BatchNorm · worked numeric example

Mini-batch of 4 activations from one neuron: $x = [1, 3, 5, 7]$.

Step 1 · mean.
$\mu = (1 + 3 + 5 + 7)/4 = 4.0$

Step 2 · variance.

- Deviations: $x - \mu = [-3, -1, 1, 3]$
- Squared: $[9, 1, 1, 9]$
- Mean of squares: $\sigma^2 = 20/4 = 5.0 \Rightarrow \sigma \approx 2.236$

Step 3 · normalize. ($\epsilon \approx 10^{-5}$)
$\hat{x} = (x - \mu)/\sqrt{\sigma^2 + \epsilon} \approx [-1.34, -0.45, 0.45, 1.34]$

Step 4 · scale + shift with learned $\gamma = 2.0,\ \beta = 0.5$:

$$y = \gamma\hat{x} + \beta = [-2.18,\ -0.40,\ 1.40,\ 3.18]$$

This vector is what the next layer sees. At **eval** time · use the running mean/var collected during training, not batch stats. (`model.eval()` flips this switch.)

The ICS debate

Ioffe & Szegedy 2015 · BN helps by reducing **internal covariate shift** (ICS) — the changing distribution of layer inputs during training.

Santurkar et al. 2018 — showed ICS was largely a red herring:

INTUITION

BN's real benefit is that it **smooths the loss landscape** — makes gradients more predictable, enabling larger learning rates.

You don't need to remember this. You *do* need to remember: BN works, but the *reason* it works is more subtle than the original paper claimed.

BatchNorm in PyTorch

```
# For 2D / FC: nn.BatchNorm1d
# For 4D / conv: nn.BatchNorm2d

layer = nn.Sequential(
    nn.Conv2d(64, 128, 3, padding=1),
    nn.BatchNorm2d(128),          # ← after the conv, before ReLU
    nn.ReLU(),
)
```

- Two learnable parameters per channel · γ (scale), β (shift)
- Two buffers per channel · `running_mean`, `running_var`
- Initialized to identity · $\gamma = 1, \beta = 0$

When BatchNorm fails

WATCH OUT

Small batch sizes — stats are noisy, BN hurts more than it helps. `batch_size < 32` → prefer GroupNorm or LayerNorm.

Sequence models — variable-length sequences have inconsistent statistics along the batch axis.

Online / streaming — can't collect meaningful batch stats.

Distributed training — each replica computes its own batch stats unless you use SyncBN.

These are exactly the situations that birthed LayerNorm.

LayerNorm · fix for sequences

Normalize across the **feature dimension** instead of the batch dimension.

$$\hat{x}_i = \frac{x_i - \mu_{\text{sample}}}{\sqrt{\sigma^2_{\text{sample}} + \epsilon}}$$

No dependence on batch size or other samples.

```
norm = nn.LayerNorm(d_model)    # applied at every Transformer block
```

KEY IDEA

Every modern Transformer (BERT, GPT, Llama, Claude) uses LayerNorm or its cheaper cousin RMSNorm — *not* BatchNorm.

RMSNorm · the cheap modern cousin

Drop the mean subtraction — keep only the scale:

DERIVATION

$$\hat{x}_i = \frac{x_i}{\sqrt{\text{mean}(x^2) + \epsilon}} \cdot \gamma$$

- ~15–30% cheaper than LayerNorm.
- Empirically no worse at convergence.
- Used in Llama, Mistral, PaLM, and most open LLMs from 2023 onward.

```
# PyTorch 2.4+
norm = nn.RMSNorm(d_model)
```

SESSION 2 · PART 8

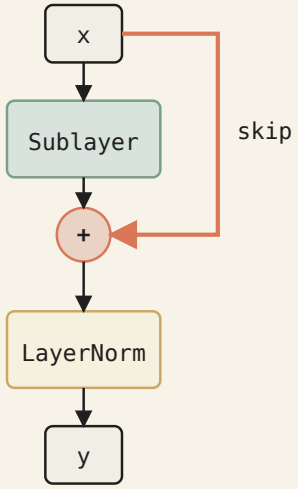
Where to put the norm

Pre-norm vs post-norm

The placement matters

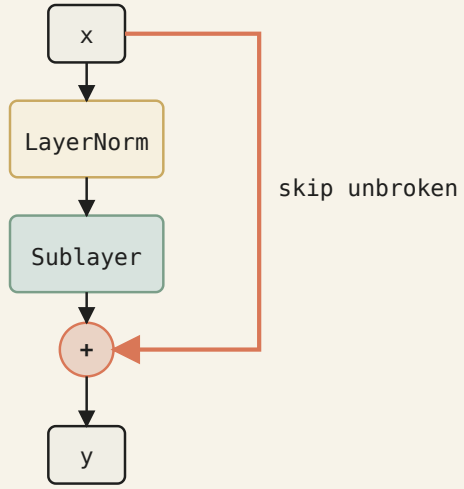
Pre-norm vs post-norm — where LayerNorm sits inside a Transformer block

POST-NORM · ORIGINAL (VASWANI 2017)



Norm is AFTER residual addition.
Gradient through residual is modulated by LayerNorm — can vanish at depth.
Needs careful warmup.

PRE-NORM · MODERN (GPT-2, LLAMA)



Norm is INSIDE the sublayer branch.
Residual gradient has direct path — scales cleanly to very deep models.
Needs less warmup.

*Everything modern (GPT-2+, Llama, Claude, Gemini) uses **pre-norm**. Only the 2017 original Transformer used post-norm.*

Why pre-norm won · the highway analogy

INTUITION

Analogy · highway and side road. The residual connection is a multi-lane **highway** that lets the gradient flow easily from the end of the network to the beginning. Sub-layers (attention, MLP) are winding **side roads**.

- **Post-norm** = put a toll booth (LayerNorm) on the highway *after* the side road merges back in. *All* traffic — highway and side road — must pass through it. Bottleneck.
- **Pre-norm** = move the toll booth to the *entrance* of the side road. The highway flows freely. Only side-road traffic gets normalized.

Why pre-norm won · the gradient path

Residual update: $\text{out} = x + \text{Sub}(x)$. The x term is the **gradient highway**.

Post-norm. Forward: $\text{out} = \text{LN}(x + \text{Sub}(x))$.

- Backward to x goes **through** LN.
- LN's gradient is a complicated, scale-dependent term in the input statistics.
- Gradient on the skip is modulated → can vanish or explode at depth → needs aggressive warmup.

Pre-norm. Forward: $\text{out} = x + \text{Sub}(\text{LN}(x))$.

- $\partial\,\text{out}/\partial x$ has a **direct** $+1$ **term** from the skip.
- LN sits on the side branch; its complicated gradient affects only the sub-layer.
- Highway is **clean** → trains stable at depth.

> IN PRACTICE
>
> Pre-norm is the default for every modern Transformer (GPT-2 onwards). If you're building a new Transformer in 2026, use pre-norm.

Decision table · which norm · which model

ARCHITECTURE	NORMALIZATION	WHY
CNN for images	**BatchNorm**	large batches, fixed shapes
Small-batch CNN (`< 32`)	**GroupNorm**	batch-independent
Transformer	**LayerNorm** (pre-norm)	batch/seq-length independent
Modern LLM (Llama, Mistral)	**RMSNorm** (pre-norm)	cheaper, no loss of quality
RNN / LSTM	**LayerNorm** across features	same reason as Transformer

Decision rule in two sentences

KEY IDEA

Fixed-size dense data with big batches → BatchNorm.

Anything else → LayerNorm (or RMSNorm if you want cheap).

Putting it together

The full regularization stack for 2026

The stack for a real vision training run

```
# 1. Architecture regularization
model = ResNet50(dropout=0.0)      # BN already in ResNet

# 2. Optimizer regularization (from L5)
opt = AdamW(model.parameters(), weight_decay=0.05)

# 3. Data augmentation
train_tfm = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandAugment(num_ops=2, magnitude=9),
    transforms.ToTensor(),
    transforms.RandomErasing(p=0.25),
    transforms.Normalize(MEAN, STD),
])

# 4. Mixup inside the training loop (conditional)
# 5. Label smoothing in the loss
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)

# 6. Early stopping — checkpoint on val loss
```

Summary · Lecture 6 — summary

Session 1 · classical + data-centric

- **Double descent** — more params past the threshold can help; generalization in the overparameterized regime is the modern story.
- **L2 / L1 / early stopping** — you know these. Use `weight_decay` in AdamW; checkpoint on val.
- **Data augmentation** — single highest-value regularizer for vision. Must preserve labels.
- **Mixup / CutMix** — interpolate inputs *and* labels. Free ~1–2% accuracy.
- **Label smoothing** — softens hard targets; better calibration. One flag.

Session 2 · architectural

- **Dropout** — implicit ensemble; inverted rescaling. `p = 0.1` for Transformers, `0.5` for MLPs.
- **BN · LN · RMSNorm** — same family, three axes. BN for CNNs, LN for Transformers, RMSNorm for LLMs.
- **Pre-norm > post-norm** for deep Transformers.

Summary · Lecture 6 — what's next

Read before Lecture 7

Prince — Ch 10 · *Convolutional networks*.

Next lecture

CNN deep dive — brisk on convolution mechanics (prereq covered LeNet); deep on receptive fields, modern architectures, and inductive-bias framing.

NOTEBOOK

Notebook 6a · `06a-regularization-stack.ipynb` — sweep weight decay, RandAugment, label smoothing on CIFAR-10.

Notebook 6b · `06b-batchnorm-by-hand.ipynb` — implement BatchNorm1d forward + backward manually (Karpathy-style); verify against `nn.BatchNorm1d`.