SCALER School of Technology × Priyansh Saxena

Initialization, Normalization, and Debugging

Xavier/He Init • Vanishing Gradients • BatchNorm • LayerNorm • Debugging

Priyansh Saxena

Scaler School of Technology | Deep Learning I: Neural Networks | Week 6 — Session 2/2

07 May 2026

Recap & Today's Questions

Session 1: Regularization

- Overfitting = model memorizes training data
- **Weight decay** penalizes large weights (L_2)
- **Dropout** randomly zeros neurons during training
- **Early stopping** halts when validation loss rises
- **Data augmentation** creates synthetic variety

Session 2: Three New Questions

- **How** should we initialize weights? → Xavier / He
- **How** do we stabilize activations? → BatchNorm / LayerNorm
- **How** do we diagnose a broken network? → Debugging

Regularization prevents overfitting. Today we ensure the network can train at all.

The Hook

Network A: Dies

- Loss $\rightarrow$ NaN after 5 epochs
- Gradients explode
- Training is useless

Network B: Converges

- Loss decreases smoothly
- Gradients flow well
- Model learns perfectly

Same architecture, same data, same optimizer.

Only difference: **how the weights were initialized.**

The Core Problem

Gradients Are Products

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_n} \cdot \frac{\partial a_n}{\partial a_{n-1}} \cdot \frac{\partial a_{n-1}}{\partial a_{n-2}} \cdots \frac{\partial a_2}{\partial w_1}$$

If each factor < 1

$0.5 \times 0.5 \times 0.5 \times \ldots$

Product $\to 0$

Gradients vanish

If each factor > 1

$1.5 \times 1.5 \times 1.5 \times \ldots$

Product $\to \infty$

Gradients explode

The telephone game: messages degrade with every relay.

Repeated Multiplication: The Numbers

Factor	After 10 layers	After 50 layers
0.5	$0.5^{10} = 0.001$	$0.5^{50} \approx 10^{-15}$
0.9	$0.9^{10} = 0.35$	$0.9^{50} = 0.005$
1.0	1.0	1.0
1.1	$1.1^{10} = 2.6$	$1.1^{50} = 117$
1.5	$1.5^{10} = 57.7$	$1.5^{50} \approx 637{,}621$

Key Insight

Even **slightly** away from 1.0, repeated multiplication causes exponential growth or decay. This is why early networks (1990s) couldn't go beyond 3–5 layers.

Symptoms in Practice

Vanishing Gradients

- Early layers have near-zero gradients
- Loss plateaus (doesn't NaN)
- Only last few layers learn
- **Sneaky** — no crash, just no learning

Exploding Gradients

- Loss jumps to NaN
- Weights oscillate wildly
- Happens within first few epochs
- **Dramatic** — easy to detect

Which Is Worse?

Vanishing — because you might not notice it. At least NaN tells you something's wrong.

Quiz 1

1. All weights initialized to `0.01` in a 50-layer network. What happens to the gradients in early layers?

2. Which activation function helps with vanishing gradients, and why?

Quiz 1

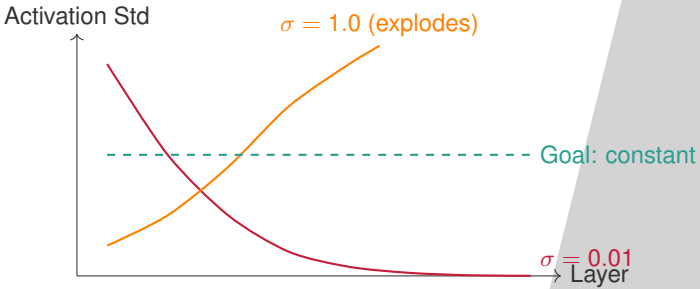
1. All weights initialized to `0.01` in a 50-layer network. What happens to the gradients in early layers?

2. Which activation function helps with vanishing gradients, and why?

Answers

1. They **vanish** — each layer multiplies by a small number, so the product shrinks exponentially.
2. **ReLU** — its derivative is exactly 1 for positive inputs (no squashing).

Why Random Init Fails



Too small: activations shrink $\rightarrow$ vanish.
Too large: activations grow $\rightarrow$ explode.
Goal: keep variance *constant* across layers.

The Goal: Preserve Variance

Design Constraint

$$\text{Var}(\text{activation at layer } l) \approx \text{Var}(\text{activation at layer } l-1)$$

For a linear layer $y = Wx$ (no activation):

- $\text{Var}(y) = n_{\text{in}} \times \text{Var}(w) \times \text{Var}(x)$
- To keep $\text{Var}(y) = \text{Var}(x)$, we need: $\text{Var}(w) = \frac{1}{n_{\text{in}}}$

Goldilocks Zone

Set weights so that signals neither vanish nor explode. We can solve this mathematically!

Xavier (Glorot) Initialization

Glorot & Bengio, 2010

$$\text{Var}(w) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

Accounts for **both** forward pass (preserve activation variance) and backward pass (preserve gradient variance).

- Designed for **symmetric activations**: sigmoid, tanh
- Breakthrough: enabled training 5+ layer networks reliably
- In practice: $w \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}+n_{\text{out}}}\right)$ or $w \sim U\left[-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right]$

He (Kaiming) Initialization

Problem with Xavier + ReLU

ReLU sets all negative values to zero $\rightarrow$ kills **half** the activations.

Variance halves at every layer $\rightarrow$ activations still vanish!

He et al., 2015 — Fix

$$\text{Var}(w) = \frac{2}{n_{\text{in}}}$$

Double the variance to compensate for ReLU's "halving effect."

Result

Activations stay stable across 20+ layers with ReLU.

This is the modern default for almost all deep networks.

When to Use Which

Activation	Initialization	Formula
Sigmoid / Tanh	Xavier (Glorot)	$\text{Var}(w) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$
ReLU / LeakyReLU / ELU	He (Kaiming)	$\text{Var}(w) = \frac{2}{n_{\text{in}}}$

Modern Default

99% of networks use ReLU variants $\rightarrow$ **He initialization**.

Good init gets you to a reasonable starting point. It doesn't guarantee convergence, but bad init guarantees failure.

PyTorch Defaults

Good news: PyTorch uses Kaiming (He) by default!

- `nn.Linear` $\rightarrow$ Kaiming uniform
- `nn.Conv2d` $\rightarrow$ Kaiming uniform
- You rarely need to change it

If you need to override:

Custom Initialization

```
nn.init.kaiming_normal_(layer.weight, mode='fan_in')
nn.init.xavier_normal_(layer.weight)
nn.init.zeros_(layer.bias)
```

Know WHY the defaults are good, even if you rarely change them.

Coming Up Next

Part 2: Batch Normalization, Layer Normalization, and Debugging

Batch Normalization: The Idea

Ioffe & Szegedy, 2015

For each mini-batch, normalize activations to zero mean and unit variance, then apply learned scale (γ) and shift (β):

$$\text{BN}(x) = \gamma \cdot \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

Why? As earlier layers change, later layers see shifting input distributions (“internal covariate shift”).

Analogy

You’re a chef and every day your ingredients change slightly. BN standardizes your ingredients before you cook.

BN: Step-by-Step

Example: 4 samples, 1 feature. Values: $[2, 4, 6, 8]$

1. **Compute batch mean:** $\mu_B = \frac{2+4+6+8}{4} = 5$
2. **Compute batch variance:** $\sigma_B^2 = \frac{(2-5)^2+(4-5)^2+(6-5)^2+(8-5)^2}{4} = 5$
3. **Normalize:** $\hat{x} = \frac{x-5}{\sqrt{5}} = [-1.34,\ -0.45,\ 0.45,\ 1.34]$
4. **Scale & shift:** with $\gamma = 2, \beta = 1$: $y = 2\hat{x} + 1$

Key Point

This happens **independently per feature** (column-wise across the batch).

Why γ and β?

Problem Without Them

Forcing all activations to $\mathcal{N}(0, 1)$ might throw away useful information.

Solution

Let the network **learn** the optimal scale and shift:

- γ = learned standard deviation
- β = learned mean
- If $\gamma = \sigma_B$ and $\beta = \mu_B \rightarrow$ BN becomes a no-op

BN can never hurt — at worst, it learns to undo itself.

Train vs. Eval Mode

Training

- Use **current batch** μ_B, σ_B^2
- Different each batch (noisy)
- Update running averages

Inference

- Use **running averages**
- Deterministic output
- Requires `model.eval()`!

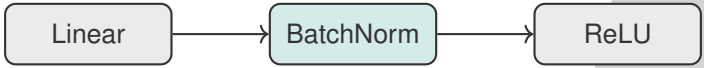
Common Bug #1

Forgetting `model.eval()` at test time $\rightarrow$ BN uses batch stats from test data $\rightarrow$ unreliable predictions!

Always call `model.eval()` before inference.

Where to Place BN

Standard pattern:



- Normalize **before** activation $\rightarrow$ inputs to ReLU are well-distributed
- **Tip:** Don't use `bias=True` in the Linear layer before BN — BN's β replaces it

In Code

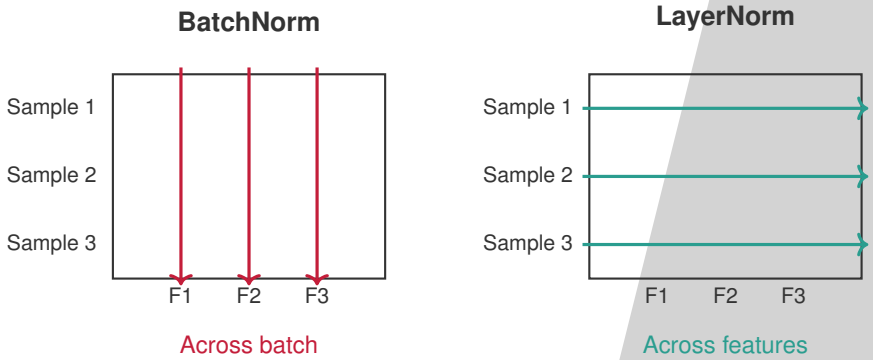
```
nn.Sequential(nn.Linear(256, 128, bias=False), nn.BatchNorm1d(128),
nn.ReLU())
```

BN Limitations

Limitation	Why It Matters
Batch size = 1	Can't compute meaningful statistics
Small batches	Noisy estimates $\rightarrow$ unstable training
Variable-length sequences	Each timestep needs different stats
Couples samples	Output of sample A depends on batch neighbors

These limitations motivated Layer Normalization.

BN vs. LN: The Axis Difference



BN: normalize each feature across the batch (needs multiple samples).
LN: normalize each sample across features (self-contained).

Layer Normalization

Ba et al., 2016

For each sample, normalize across **all features** independently:

$$\text{LN}(x) = \gamma \cdot \frac{x - \mu_{\text{features}}}{\sqrt{\sigma^2_{\text{features}} + \epsilon}} + \beta$$

Advantages over BN

- Works with **batch size = 1**
- Works with **variable-length sequences**
- No coupling between samples in a batch
- Same behavior at train and test time

Used in virtually all Transformers (BERT, GPT, etc.)

Quiz 2

1. During **inference**, where does BatchNorm get its mean and variance?

2. Why is LayerNorm preferred over BatchNorm in Transformers?

Quiz 2

1. During **inference**, where does BatchNorm get its mean and variance?

2. Why is LayerNorm preferred over BatchNorm in Transformers?

Answers

1. From **running averages** accumulated during training (NOT the current batch).
2. Transformers process variable-length sequences with varying batch sizes, and LN has no batch dependency — works with batch size 1 at inference.

Gradient Checking

Numerical Gradient (Central Difference)

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \qquad (\epsilon \approx 10^{-5})$$

Compare with your analytical gradient:

$$\text{relative error} = \frac{|g_{\text{analytical}} - g_{\text{numerical}}|}{\max(|g_{\text{analytical}}|,\ |g_{\text{numerical}}|)}$$

Relative Error	Verdict
$< 10^{-7}$	Excellent
$< 10^{-5}$	Acceptable
$> 10^{-3}$	Something is wrong

Slow (2 forward passes per parameter) — only for debugging, never in training.

Common Failure Modes & Fixes

Symptom	Likely Cause	Fix
Loss = $\ln$(classes), stuck	Not learning at all	Check data pipeline, loss fn, LR
Loss $\rightarrow$ NaN	Exploding gradients	Lower LR, gradient clip, check init
Loss plateaus early	Vanishing gradients	Check activations, add skip connections
Train $\downarrow$, val $\uparrow$	Overfitting	Regularize (last session!)

Quick Sanity Check

For a k-class classifier with random init, loss should be $\approx \ln(k)$.
If it's much higher: something is broken before training even starts.

The Debugging Checklist

Step 1 Overfit ONE batch

8 samples, train to zero loss. **If this fails, stop.**

Step 2 Check tensor shapes

Print shapes at every layer. Silent mismatches cause mysterious bugs.

Step 3 Verify loss at init

For k-class: $L_0 \approx \ln(k)$. Wrong value $\Rightarrow$ bad init or loss.

Step 4 Gradient check

Compare numerical vs. analytical gradients. Catches custom backward bugs.

Step 5 Small subset

Train on $\sim$100 samples. Should overfit quickly — confirms pipeline.

Step 6 Scale up

Add full data, regularization, tuning. Only here after steps 1–5 pass.

Follow in order. Never skip ahead.

Final Quiz

Three pillars:

1. Initialize right

Xavier (tanh) He (ReLU)

2. Normalize

BN for vision LN for sequences

3. Debug systematically

Follow the checklist. Never skip step 1.

Quick-fire quiz:

Q1

30-layer ReLU net, dead gradients in early layers. Fix?

Q2

`model.eval()` drops accuracy by 10%. What happened?

Q3

Model is not learning at all. What is the first step?

Take 2 minutes.

Final Quiz — Answers

Three pillars:

1. Initialize right

Xavier (tanh) He (ReLU)

2. Normalize

BN for vision LN for sequences

3. Debug systematically

Follow the checklist. Never skip step 1.

Q1

30-layer ReLU net, dead gradients. Fix?

A1

He initialization

Q2

`eval()` drops accuracy 10%. Why?

A2

BN running stats not properly accumulated

Q3

Model not learning — first step?

A3

Overfit one batch

Next week: Convolutional Neural Networks!

Thank You!

Questions?

Priyansh Saxena

All lecture materials available at:

www.priyanshsaxena.com/sst-deep-learning-nn