Machine Learning and Optimization

Lecture 16: Training Deep Networks

CSCI 4140/6960 — Spring 2026

Course website: `cs.rpi.edu/~gittea/.../mlandopt.html`

Learning Objectives

By the end of this lecture, you should be able to:

1. **Identify** vanishing and exploding gradients as manifestations of poorly conditioned Jacobian products, and **state** the spectral norm bound on gradient norms.
2. **Explain** how weight initialization (Xavier, He) stabilizes forward and backward signal propagation at the start of training.
3. **Compare** activation functions (sigmoid, tanh, ReLU, Leaky ReLU, GELU) in terms of saturation, derivative behavior, and gradient flow.
4. **State** the batch normalization transform and **distinguish** its training and inference behavior.

Learning Objectives (continued)

5. **Distinguish** batch normalization and layer normalization and **identify** when each is appropriate.
6. **Explain** how residual connections provide well-conditioned gradient pathways and **state** the ResNet residual form.
7. **Describe** auxiliary losses as a mechanism for injecting gradient signal at intermediate layers.

Lecture Roadmap

- The Vanishing/Exploding Gradient Problem
- The Multiplier Lens
- Initialization
- Activation Functions and Saturation
- Normalization Layers
- Layer Normalization
- Shortening the Gradient Path
- Auxiliary Losses
- Unifying View and Summary

Challenges of Depth

Three main challenges come up in training deep networks:

1. **Overfitting:** too much capacity relative to data.
2. **Vanishing and exploding gradients:** slow or failed learning.
3. **Hyperparameter selection explosion:** many interacting design choices.

This lecture focuses on the second challenge and develops a unified view:

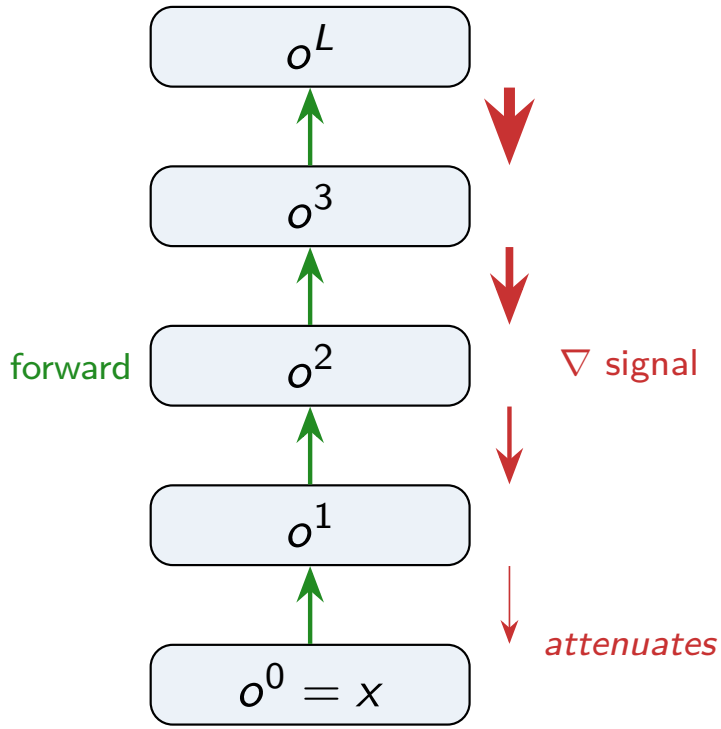
All solutions aim to stabilize signal propagation across depth — both forward (activations) and backward (gradients).

Vanishing and Exploding Gradients

As backpropagation proceeds from layer L toward layer 1, $\|\nabla_{o^s} f\|_2$ either:

- $\to 0$ (**vanishing**)
- $\to \infty$ (**exploding**)

Cause: the chain rule produces a *product of matrices* whose conditioning degrades with depth.



A Two-Layer Warm-Up

Before the general formula, consider a concrete case.

Setup: 2-layer MLP, widths $n_0 = n_1 = n_2 = 3$, sigmoid activation.

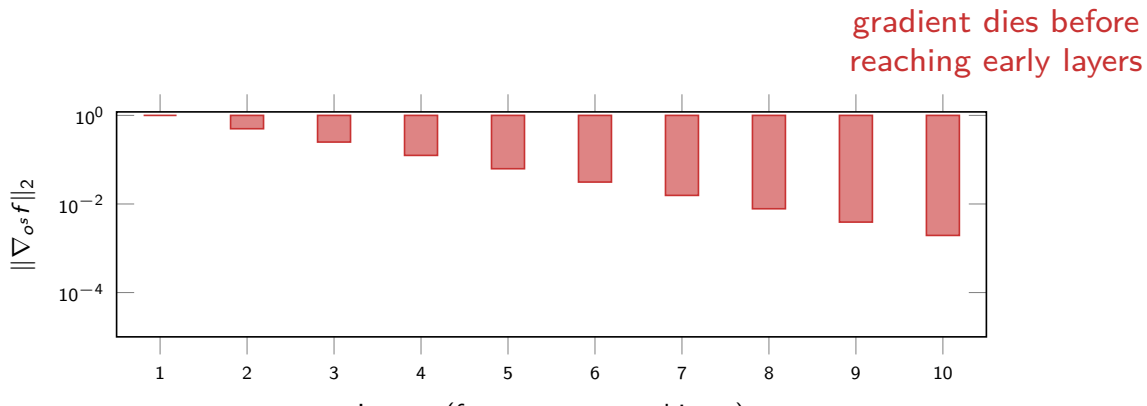
Gradient at the input:

$$\nabla_{o^0} f = \underbrace{(W^1)^\top \operatorname{diag}(\sigma'(a^1))}_{\text{layer 1 factor}} \underbrace{(W^2)^\top \operatorname{diag}(\sigma'(a^2))}_{\text{layer 2 factor}} \nabla_{o^2} f$$

Typical magnitudes with sigmoid: $\|W^s\|_2 \approx 0.8$, $\|\sigma'\|_\infty \leq 0.25$.

Each factor shrinks the signal by $\approx 0.8 \times 0.25 = 0.2$.

Two layers: $0.2^2 = 0.04\times$ attenuation. Now imagine **50 layers**: $0.2^{50} \approx 10^{-35}$.



The Jacobian Product

Recall: $o^{s+1} = \sigma(W^{s+1} o^s + b^{s+1})$, so:

$$\nabla_{o^s} f = \left(W^{s+1}\right)^\top \operatorname{diag}\left(\sigma'(a^{s+1})\right) \cdot \nabla_{o^{s+1}} f$$

Unrolling from layer s to L:

$$\nabla_{o^s} f = \prod_{i=s+1}^{L} \left[\left(W^i\right)^\top \operatorname{diag}\left(\sigma'(a^i)\right)\right] \cdot \nabla_{o^L} f$$

Spectral norm bound:

$$\|\nabla_{o^s} f\|_2 \;\leq\; \prod_{i=s+1}^{L} \underbrace{\|W^i\|_2 \cdot \|\sigma'\|_\infty}_{=:\, m_i} \cdot \|\nabla_{o^L} f\|_2$$

This formalizes what the warm-up showed numerically: each layer contributes a multiplicative factor m_i, and $L - s$ such factors compound.

The Jacobian Product (continued)

Two mechanisms that lead to poorly conditioned Jacobian products:

1. **Activation saturation:** if $\sigma'(a^i) \approx 0$ entrywise, the $\|\sigma'\|_\infty$ factor drives m_i toward zero.
2. **Weight matrix norms:** if $\|W^i\|_2$ is consistently > 1 or < 1, the product grows or shrinks exponentially in $L - s$.

Concrete example: if the gradient doubles every layer, after 50 layers it grows by $2^{50} \approx 10^{15}$. If it halves every layer, it shrinks by $2^{-50} \approx 10^{-15}$.

Consequences

Consequence: naively training deep architectures either fails or gives poor performance.

Brute-force mitigation: gradient clipping — cap $\|\nabla f\|$ at a threshold before each update. Widely used in RNN training. Treats the symptom, not the cause.

This lecture: a single diagnostic — the *per-layer multiplier* m_s — and a roadmap of principled approaches widely used to mitigate this issue.

The Per-Layer Multiplier

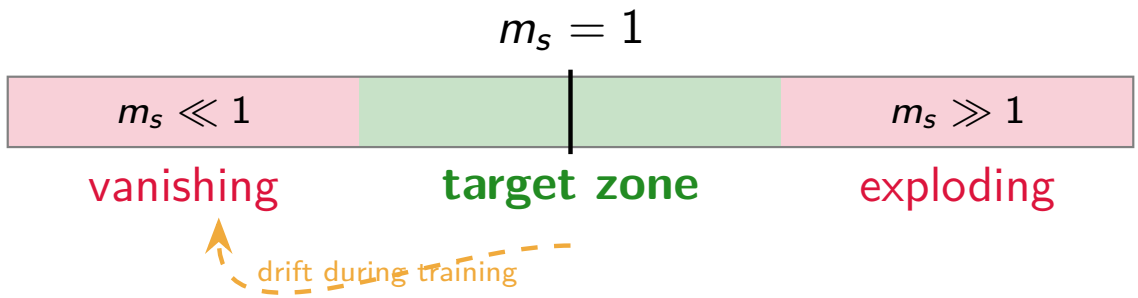
Define the **per-layer multiplier**:

$$m_s := \|W^s\|_2 \cdot \|\sigma'\|_\infty$$

Then the spectral bound becomes:

$$\|\nabla_{o^s} f\|_2 \leq \left(\prod_{i=s+1}^{L} m_i \right) \cdot \|\nabla_{o^L} f\|_2$$

Goal: keep $m_s \approx 1$ for all s, throughout training.



Two failure modes as m_s drifts during training: $m_s < 1$ (vanishing), $m_s > 1$ (exploding).

Roadmap Through the Multiplier Lens

Every technique in this lecture is a targeted intervention on m_s:

Method	**Acts on**	**Effect on m_s**
Initialization	$\|W^s\|_2$ at $t = 0$	Sets $m_s \approx 1$ initially
Activation choice	$\|\sigma'\|_\infty$ factor	Avoids $m_s \approx 0$ from saturation
Batch/Layer Norm	activations $\rightarrow$ indirectly m_s	Renormalizes scale during training
Residual conn.	Jacobian structure	Bypasses the product: adds identity
Auxiliary losses	Effective product length	Shortens $\prod$ by injecting gradient

We will revisit this table at the end of the lecture to verify each claim.

Pause and Check

Q. If every layer has $m_s = 1.05$, what happens to $\|\nabla_{o^1} f\|_2$ in a 100-layer network?

A. $(1.05)^{99} \approx 125$ — **exploding**. Even a 5% excess per layer compounds to two orders of magnitude.

Q. Which factor in $m_s = \|W^s\|_2 \cdot \|\sigma'\|_\infty$ does sigmoid make problematic? Which does poor initialization make problematic?

A. Sigmoid: $\|\sigma'\|_\infty \leq 0.25$, driving m_s well below 1. Poor initialization: $\|W^s\|_2$ off-scale (too large or too small).

Controlling Both Forward and Backward Passes

It is essential to control signal propagation in *both* directions:

1. **Saturation is a forward-pass problem that poisons the backward pass.** If pre-activations a^s drift into the saturating tails of sigmoid or tanh (a forward phenomenon), then $\sigma'(a^s) \approx 0$, which kills m_s regardless of $\|W^s\|_2$.
2. **The backprop seed depends on forward propagation.** If o^L is numerically degenerate (overflow/underflow), then $\nabla_{o^L} f$ — the initial condition of the backward pass — is garbage.

Forward and backward stability are **coupled but not equivalent**. Controlling each requires its own condition on $\mathrm{Var}(W^s)$, and these conditions generically conflict — motivating the compromise that is Xavier initialization.

Variance Propagation: Forward Pass

The original analysis (Glorot & Bengio, 2010) asks: under what condition on $\mathrm{Var}(W^s)$ does the per-coordinate activation variance stay constant across layers?

Why this is the right question forward: if $\mathrm{Var}(a_j^s)$ grows with depth, pre-activations leave the well-behaved region of σ and saturate. If it shrinks, activations collapse to zero.

Single neuron (**assuming** $\sigma = \mathrm{id}$): $a_j^s = \sum_{k=1}^{n_{s-1}} W_{jk}^s \, o_k^{s-1}$. With $W_{jk}^s \perp o_k^{s-1}$ (both zero-mean):

$$\mathrm{Var}(a_j^s) = \underbrace{n_{s-1}}_{\text{fan-in}} \cdot \mathrm{Var}(W_{jk}^s) \cdot \mathrm{Var}(o_k^{s-1})$$

Preserve per-coordinate variance: $\boxed{\mathrm{Var}(W_{jk}^s) = 1/n_{s-1}}$.

Bias terms: $b^s = 0$ at init contributes nothing; during training, bias drift is handled by normalization layers.

What Should We Preserve Backward?

Forward, we preserved *per-coordinate variance* of activations — this keeps pre-activations in σ's well-behaved region.

Backward, saturation is not a concern for gradients. What we need is for the *magnitude* of the gradient signal to neither blow up nor vanish as it propagates from layer L toward layer 1.

The natural quantity is therefore the **gradient norm** $\mathbb{E}[\|\nabla_{o^s} f\|^2]$, not the per-coordinate variance.

Glorot & Bengio (2010) instead preserved per-coordinate gradient variance in both directions (applying the same algebraic template as the forward pass). As we will see, this leads to an apparent conflict that vanishes when you preserve the right quantity.

Backward Pass: Per-Coordinate Calculation

Coordinate-wise backward recurrence (**assuming** $\sigma' = 1$):

$$[\nabla_{o^{s-1}} f]_j = \sum_{k=1}^{n_s} W^s_{kj} \cdot [\nabla_{o^s} f]_k$$

This is a sum of n_s independent terms (one per neuron in layer s), so:

$$\mathbb{E}\left[[\nabla_{o^{s-1}} f]_j^2\right] = n_s \cdot \mathrm{Var}(W^s) \cdot \mathbb{E}\left[[\nabla_{o^s} f]_k^2\right]$$

This tells us how a single coordinate's second moment changes across one layer.

But we want to compare $\mathbb{E}[\|\nabla_{o^{s-1}} f\|^2]$ to $\mathbb{E}[\|\nabla_{o^s} f\|^2]$ — i.e., we need to *sum over coordinates*. To do that, we need to know whether all coordinates have the same second moment.

Do they?

Gradient Coordinates Are Exchangeable

Claim (proved in Appendix A)

At initialization, the coordinates of $\nabla_{o^s} f$ are **exchangeable**: for any permutation π of neurons at layer s,

$$P_\pi \nabla_{o^s} f \overset{d}{=} \nabla_{o^s} f.$$

Intuition: at initialization, every neuron at layer s has the same weight distribution, same activation, same connectivity. Swapping any two neurons' labels (and adjusting wiring consistently) doesn't change the joint distribution of the network — but it does swap the corresponding gradient coordinates.

Consequence: all coordinates share the same second moment. Writing $c_s := \mathbb{E}\big[[\nabla_{o^s} f]_k^2\big]$ (same for all k):

$$\mathbb{E}\big[\|\nabla_{o^s} f\|^2\big] = \sum_{k=1}^{n_s} \mathbb{E}\big[[\nabla_{o^s} f]_k^2\big] = n_s \cdot c_s$$

Backward Pass: Gradient Norm Preservation

From the per-coordinate calculation: $c_{s-1} = n_s \cdot \mathrm{Var}(W^s) \cdot c_s$.

From exchangeability: $\mathbb{E}[\|\nabla_{o^s} f\|^2] = n_s \cdot c_s$.

Combining:

$$\frac{\mathbb{E}[\|\nabla_{o^{s-1}} f\|^2]}{\mathbb{E}[\|\nabla_{o^s} f\|^2]} = \frac{n_{s-1} \cdot c_{s-1}}{n_s \cdot c_s} = \frac{n_{s-1} \cdot n_s \cdot \mathrm{Var}(W^s) \cdot c_s}{n_s \cdot c_s} = n_{s-1} \cdot \mathrm{Var}(W^s)$$

Note that c_s cancels — we never need to know its value, only that it is the same for all coordinates.

Preserve gradient norm (ratio $= 1$): $\boxed{\mathrm{Var}(W^s_{jk}) = 1/n_{s-1}}$

This is the same condition as the forward pass!

There Is No Conflict

Quantity preserved	**Forward**	**Backward**	
Per-coordinate variance	$1/n_{s-1}$	$1/n_s$	(Glorot & Bengio)
Per-coordinate variance (fwd) + gradient norm (bwd)	$1/n_{s-1}$	$1/n_{s-1}$	

The "conflict" between $1/n_{s-1}$ and $1/n_s$ that motivated the Xavier compromise $2/(n_{s-1}+n_s)$ arose from preserving per-coordinate variance backward.

If you instead preserve the gradient *norm* — arguably the quantity you actually care about — both directions agree: $\mathrm{Var}(W) = 1/n_{s-1}$.

For equal-width layers ($n_{s-1} = n_s$), the distinction vanishes entirely. In practice, Xavier's harmonic mean is close enough to $1/n_{s-1}$ that the difference rarely matters.

Xavier and He Initialization

Xavier/Glorot initialization (Glorot & Bengio, 2010):

The original paper preserves per-coordinate variance in both directions, obtaining $1/n_{s-1}$ (forward) vs. $1/n_s$ (backward), then compromises:

$$W_{jk}^s \sim \mathcal{N}\left(0,\ \frac{2}{n_{s-1}+n_s}\right)$$

As we argued, preserving gradient *norm* also gives $1/n_{s-1}$, so the compromise is theoretically unnecessary (but harmless in practice).

He initialization (He et al., 2015) — accounts for ReLU:

ReLU zeroes out half the activations: $\mathbb{E}[(\mathrm{ReLU}')^2] = 1/2$. Both the forward (per-coordinate variance) and backward (gradient norm) conditions give the *same* answer:

$$\boxed{W_{jk}^s \sim \mathcal{N}\left(0,\ \frac{2}{n_{s-1}}\right)}$$

Applied in PyTorch via `kaiming_normal_`.

Initialization and the Multiplier Lens

What does initialization give us, concretely?

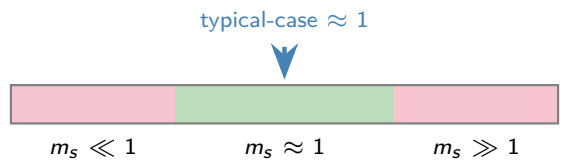
For $W \in \mathbb{R}^{n \times n}$ with i.i.d. $\mathcal{N}(0, 1/n)$ entries and any fixed vector x:

$$\mathbb{E}[\|Wx\|^2] = \|x\|^2.$$

So W acts as an **approximate isometry on typical vectors**: its RMS singular value is 1.

What it does *not* give us: $\|W\|_2 \approx 1$. For large n, $\|W\|_2 \to 2$ (Marchenko–Pastur), so the worst-case multiplier $m_s = \|W^s\|_2 \cdot \|\sigma'\|_\infty \approx 2$. The spectral bound is pessimistic because backpropagated gradients are not adversarially aligned with the top singular direction.

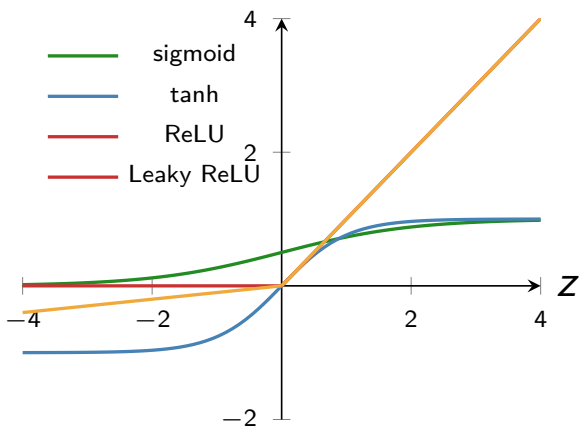
Bottom line: the multiplier lens gives correct *qualitative* diagnostics (is m_s order-1 or not?). The variance analysis gives the correct *quantitative* initialization.



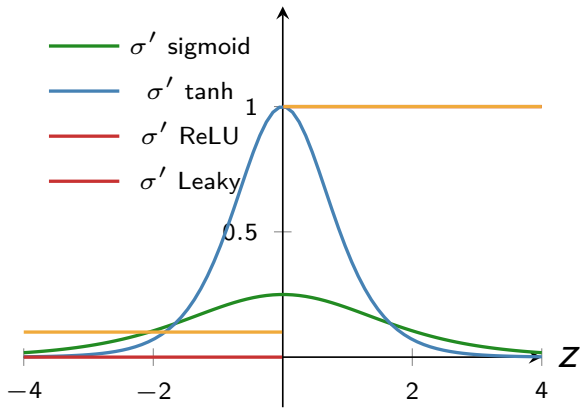
Activation Functions: A Gradient Perspective

The activation derivative $\sigma'(\cdot)$ appears in every factor of the Jacobian product. Choice of σ directly controls the $\|\sigma'\|_\infty$ factor in m_s.

Functions $\sigma(z)$



Derivatives $\sigma'(z)$



Saturation, Sparsity, and Dead Neurons

Sigmoid / tanh — saturate for $|z| \gg 0$: $\sigma'(z) \approx 0$.

- Sigmoid: $\|\sigma'\|_\infty = 0.25$, so $m_s \leq 0.25 \, \|W^s\|_2$ — well below 1 even with perfect initialization.
- Saturated neurons compound: each contributes a near-zero diagonal entry.

ReLU ($\max(0, z)$): $\sigma' \in \{0, 1\}$ — no shrinkage when active.

- **Dead neurons:** if $a_j^s < 0$ for all inputs, the j-th diagonal entry of $\mathrm{diag}(\sigma'(a^s))$ is permanently zero, zeroing out one row of the per-layer Jacobian and reducing effective rank.

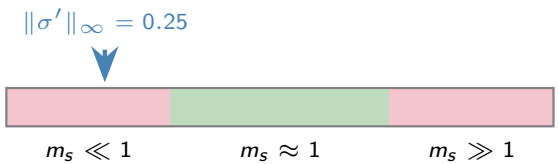
Leaky ReLU ($\max(\alpha z, z)$, $\alpha \approx 0.01$): $\sigma' \in \{\alpha, 1\}$, so $m_s > 0$ always.

GELU: $\sigma(z) = z\Phi(z)$. Smooth, nonzero derivative everywhere. Default in Transformers.

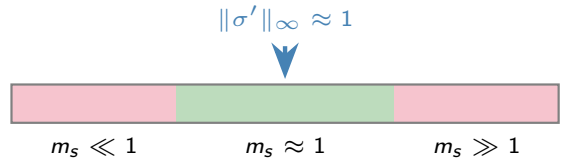
Takeaway: Activation Choice and Gradient Flow

Activation	**Saturates?**	σ' **range**	**Dead neurons?**	**Common in**
Sigmoid	Yes	$(0, 0.25]$	No	Legacy / outputs
tanh	Yes	$(0, 1]$	No	Legacy / RNNs
ReLU	No ($z > 0$)	$\{0, 1\}$	Yes	CNNs, MLPs
Leaky ReLU	No	$\{\alpha, 1\}$	No	CNNs, MLPs
GELU	No	smooth, > 0	No	Transformers

Sigmoid:



ReLU / GELU:



Pause and Check

Q. You initialize a 30-layer sigmoid network with Xavier init ($\|W^s\|_2 \approx 1$). Estimate m_s and the total gradient attenuation from layer 30 to layer 1.

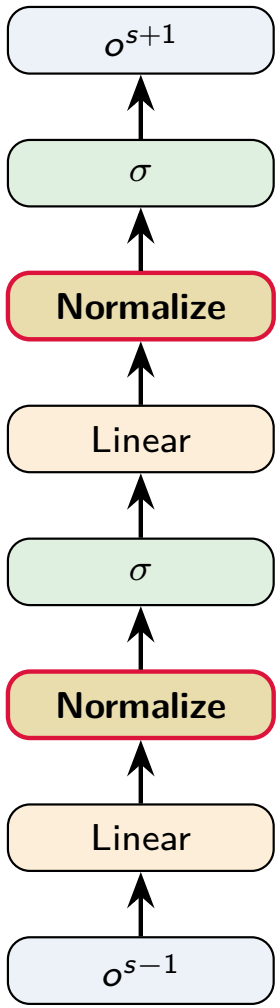
A. $m_s \approx 0.25$. Total: $0.25^{29} \approx 3 \times 10^{-18}$. The gradient is effectively zero at the early layers.

Q. Now switch to ReLU with He init. Estimate m_s.

A. He init sets $\|W^s\|_2$ so that $m_s \approx 1$ (accounting for $\mathbb{E}[\sigma'] \approx 1/2$ via the extra factor of 2). Total attenuation: $\approx 1^{29} = 1$. Gradient propagates cleanly.

Normalization Layers: The Idea

Initialization sets $m_s \approx 1$ at $t = 0$, but weights evolve during training and m_s drifts. **Normalization layers** are modules inserted between layers that dynamically re-center and re-scale activations to maintain stable signal propagation.



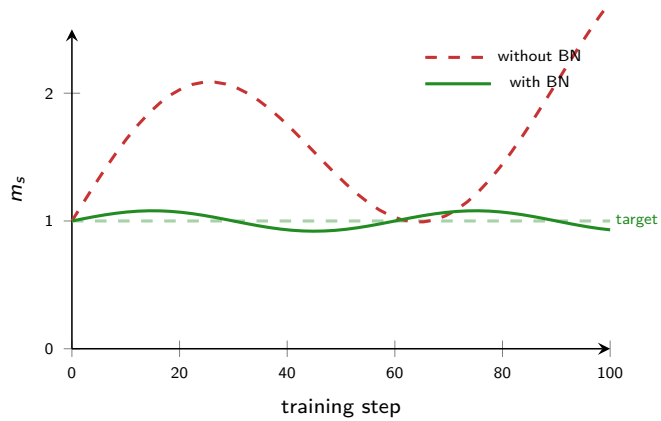
inserted between linear and activation

Batch Normalization: Stabilizing the Multiplier During Training

Batch normalization (BN) (Ioffe & Szegedy, 2015) computes normalization statistics *per minibatch*: the mean and variance used to normalize each neuron's pre-activation are estimated from the m samples in the current minibatch.

Effects:

- Pre-activations stay where σ' is well-behaved
- Prevents bias accumulation
- Empirically stabilizes m_s throughout training ⊙



Schematic: BN prevents multiplier drift.

The Batch Normalization Transform

Definition (Batch Normalization — BN)

Given a minibatch of m samples, for each neuron j at layer s:

$$\mathrm{BN}_{\gamma,\beta}(a^s) = \gamma \odot \frac{a^s - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \varepsilon}} + \beta,$$

where

$$\mu_{\mathcal{B},j} = \tfrac{1}{m}\textstyle\sum_i a^s_j(x_i) \qquad \sigma^2_{\mathcal{B},j} = \tfrac{1}{m}\textstyle\sum_i (a^s_j(x_i) - \mu_{\mathcal{B},j})^2$$

- $\gamma, \beta \in \mathbb{R}^{n_s}$: learned scale and shift (via backprop)
- $\varepsilon \sim 10^{-6}$: numerical stability constant
- Bias b^s is redundant when BN precedes σ (absorbed by β)

Training vs. Inference

Training (`model.train()`)

Statistics from the current minibatch:

$$\mu_s = \frac{1}{m}\sum_{i=1}^{m} a^s(x_i)$$

$$\sigma_s^2 = \frac{1}{m}\sum_{i=1}^{m}\left(a^s(x_i) - \mu_s\right)^2$$

Caveat: meaningful only when $m \gg 1$.

Inference (`model.eval()`)

Running EMA accumulated during training:

$$\hat{\mu}^s \leftarrow \beta_1\,\hat{\mu}^s + (1-\beta_1)\,\mu_s^{\mathcal{B}}$$

$$\hat{\sigma}_s^2 \leftarrow \beta_2\,\hat{\sigma}_s^2 + (1-\beta_2)\,\sigma_{s,\mathcal{B}}^2$$

Frozen at test time $\rightarrow$ deterministic.

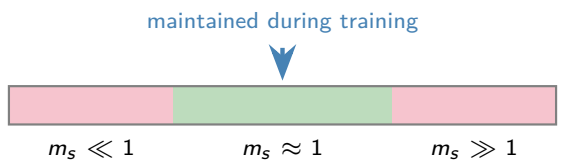
Common PyTorch bug: forgetting to switch `model.train()` $\leftrightarrow$ `model.eval()`.

Batch Normalization for CNNs

Per-channel normalization: one (μ_c, σ_c^2) pair per channel, computed over all spatial positions and all samples in the minibatch.

For feature maps of shape $m \times C \times H \times W$: each channel's statistics use $m \cdot H \cdot W$ values.

Preserves translational equivariance: every spatial position in channel c is shifted and scaled by the same γ_c, β_c.



When Batch Normalization Breaks Down

BN requires large minibatches ($m \gg 1$). This can be problematic:

- Memory constraints may force small m (large models, high-resolution inputs).
- RNNs / variable-length sequences: BN needs separate parameters per time step.

This motivates normalizing **across neurons** instead of across the batch.

Layer Normalization

Definition (Layer Normalization)

$$\mathrm{LayerNorm}_{\gamma,\beta}(a^s) = \gamma \odot \frac{a^s - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

where:

$$\mu = \frac{1}{n_s} \sum_{i=1}^{n_s} a_i^s(x), \qquad \sigma^2 = \frac{1}{n_s} \| a^s(x) - \mu \, \mathbf{1} \|_2^2$$

- Statistics per instance, across all neurons — no batch dependence.
- $\gamma, \beta \in \mathbb{R}^{n_s}$ learned as in BN.
- Identical behavior at training and inference (no EMA needed).

Batch Norm vs. Layer Norm

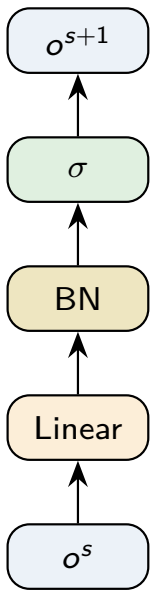
	Batch Norm	**Layer Norm**
Stats computed over	minibatch (per neuron)	single instance (per layer)
Requires large m?	Yes	No
Train/test differ?	Yes (EMA at test)	No
Natural for CNNs?	Yes (per-channel)	Less common
Natural for sequences?	No	Yes

Layer normalization is the standard in Transformer architectures over batch normalization. The **pre-norm** variant (LN applied *before* the attention/MLP sub-layer rather than after) has become dominant since GPT-2 because it provides more stable gradient flow in very deep models.

Post-Activation vs. Pre-Activation Normalization

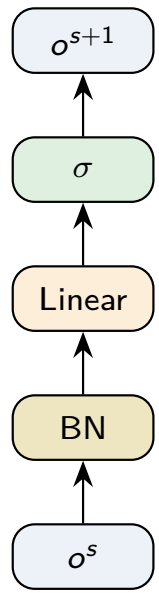
Post-activation (Ioffe & Szegedy, 2015):

$$o^{s+1} = \sigma\big(\mathrm{BN}(W^{s+1}o^s)\big)$$



Pre-activation (He et al., 2016):

$$o^{s+1} = \sigma\big(W^{s+1}\mathrm{BN}(o^s) + b^{s+1}\big)$$



Brief historical arc: BN post-activation (2015, Inception v2) $\rightarrow$ pre-activation ResNets (He et al., 2016) $\rightarrow$ pre-norm LN in Transformers (GPT-2, 2019 onward). Vanilla CNNs: post-activation BN remains standard.

Pause and Check

Q. You're training a NN with layer size 2048 and batch size 2. Why is batch normalization problematic here? What do you use instead?

A. BN computes statistics over the minibatch. With $m = 2$, the sample mean and variance are extremely noisy — the normalization does more harm than good. Use layer normalization, which computes statistics per instance across neurons and has no batch dependence.

Beyond Per-Layer Fixes: Shortening the Gradient Path

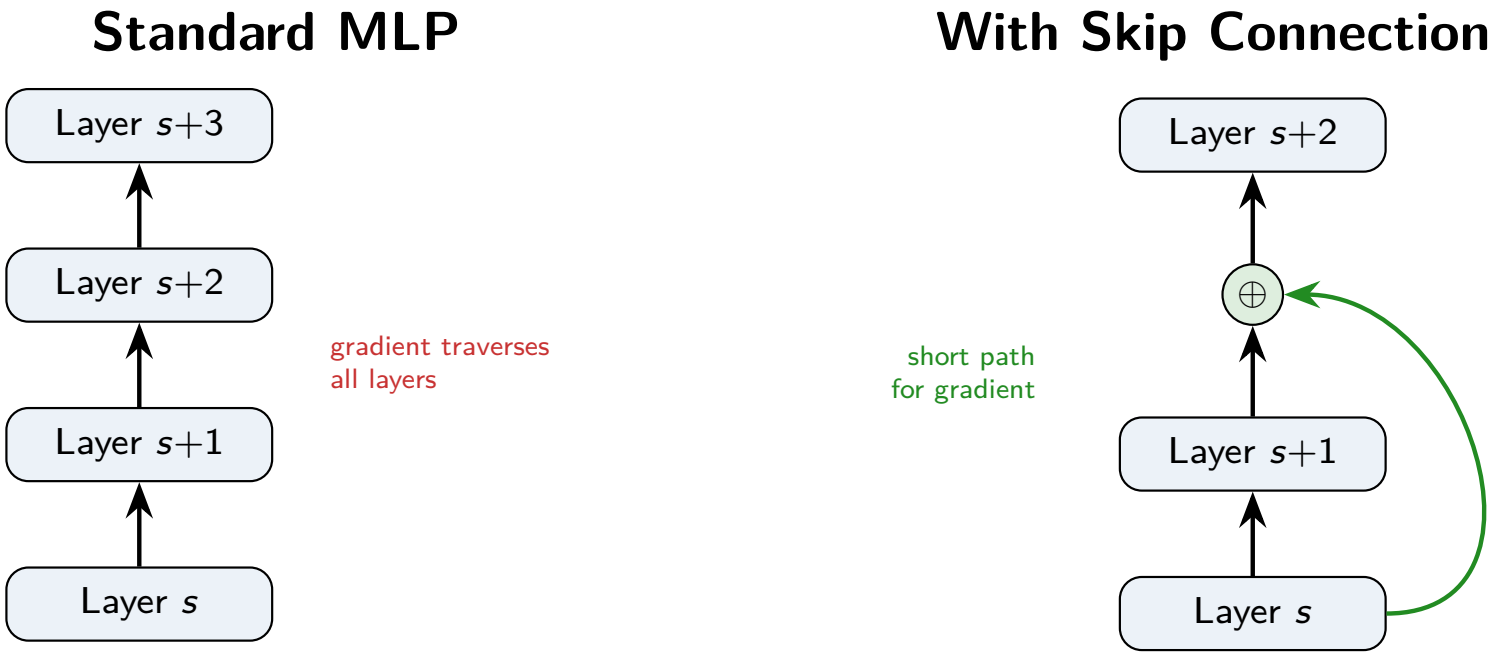
Initialization, activation choice, and normalization all act on the per-layer multiplier m_s. But even with $m_s \approx 1$ at every layer, a product of 50+ factors can still drift.

An alternative strategy: **shorten the effective path length** from the output to a given layer, so that the gradient traverses fewer multiplicative factors.

Two main approaches:

1. **Skip (residual) connections:** add architectural shortcuts that let the gradient bypass layers entirely, converting the Jacobian product into a sum over paths of varying length.
2. **Auxiliary losses:** attach extra loss heads at intermediate layers, injecting gradient signal directly without requiring it to traverse the full depth.

Skip Connections: Architectural Gradient Shortcuts



Even with proper initialization and normalization, very deep networks ($L > 50$) still struggle because the product of $L - s$ factors accumulates error. Skip connections provide architecturally guaranteed short gradient paths.

Skip Connections: Definition

Definition (Skip Connection)

A *skip connection* (or *residual connection*) is a connection from layer s to layer $s + k$ ($k \geq 2$) that bypasses intermediate transformations.

Two rationales:

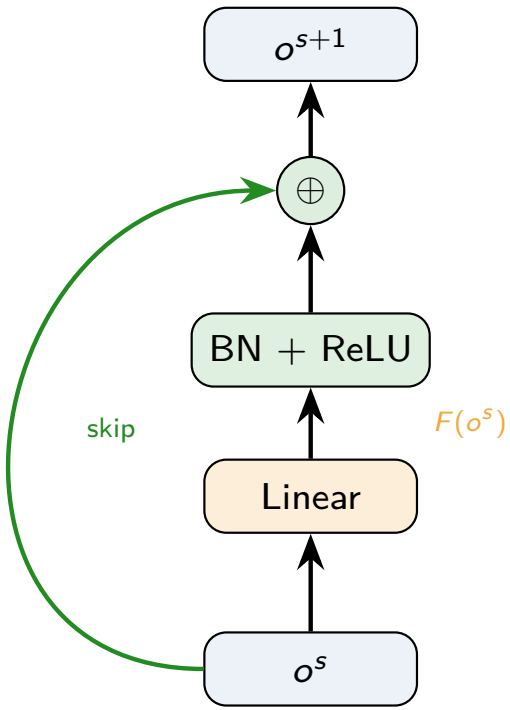
1. **Gradient flow:** short paths avoid compounding attenuation/explosion.
2. **Inductive bias:** higher layers retain access to lower-level features (e.g., U-Nets in image segmentation).

ResNet: Skip Connections in Practice (He et al., 2015)

Canonical residual form:

$$o^{s+1} = o^s + F(o^s)$$

where F is a small sub-network (e.g., Linear $\rightarrow$ BN $\rightarrow$ ReLU).



Important properties:

- If $F(o^s) \approx 0$: $o^{s+1} \approx o^s$ — the network easily represents the identity.
- The network learns the *residual* on top of identity.
- Requires matching dimensions; when $n_{s+1} \neq n_s$, use a projection Po^s on the skip branch.

Why Skip Connections Stabilize Gradient Flow

Gradient through a residual layer:

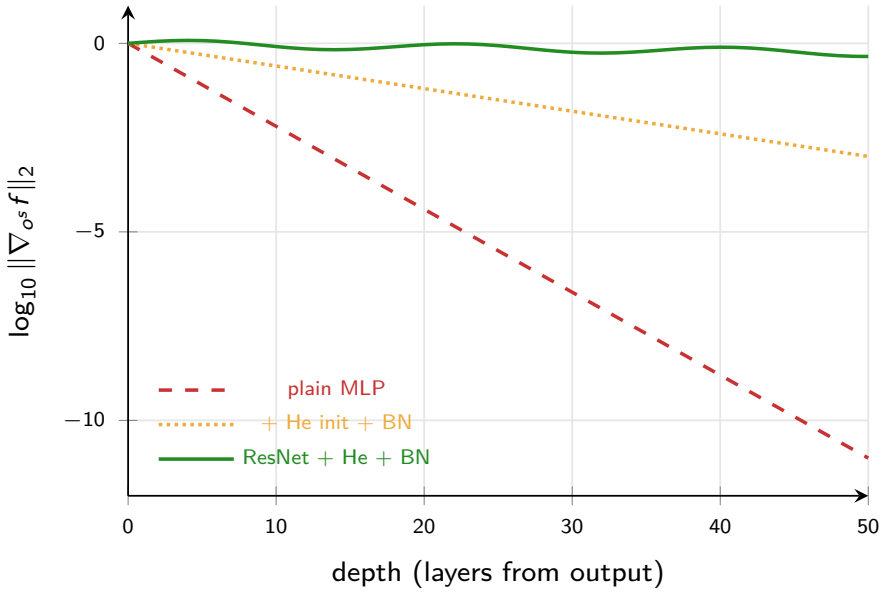
$$\nabla_{o^s} f = \underbrace{\nabla_{o^{s+1}} f}_{\text{skip (identity)}} + \underbrace{\nabla_{o^s} F(o^s)^\top \cdot \nabla_{o^{s+1}} f}_{\text{main branch}}$$

The first term passes through the skip **without attenuation** — no weight matrices, no activation derivatives.

The plain network computes a *product* of Jacobians. The residual network computes a *sum over paths* — each path is a product of a *subset* of Jacobian factors. This converts **multiplicative instability** into **additive structure**.

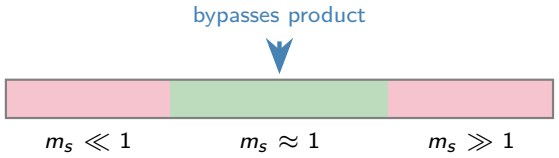
Over k residual layers, the gradient is a sum of 2^k terms. Even if long paths (many main-branch factors) attenuate, the short paths (mostly skips) preserve gradient magnitude.

Gradient Flow: The Payoff

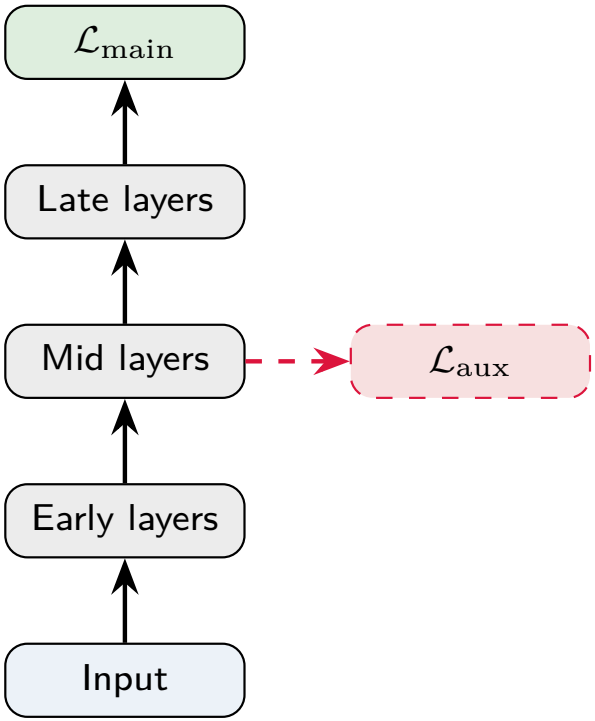


Schematic comparison (not real data):

- Plain MLP: exponential decay ($m_s < 1$ compounds)
- + He + BN: slower decay ($m_s \approx 1$ but products still accumulate)
- ResNet: roughly flat (identity paths bypass products)



Auxiliary Losses: Direct Gradient Injection



In very deep networks (e.g., GoogLeNet, 22 layers; Szegedy et al., 2014), early layers can receive weak gradient signal even with normalization and careful initialization.

Idea: attach auxiliary classification heads at intermediate points to directly inject gradient information.

Training objective:

$$\mathcal{L} = \mathcal{L}_{\text{main}} + \lambda \mathcal{L}_{\text{aux}}$$

Through the multiplier lens: auxiliary losses don't change m_s — they **shorten the effective product length** by injecting $\nabla_{o^L} f$-scale signal at an intermediate layer.

Unifying View: Did We Deliver?

We promised to keep $m_s \approx 1$. Let's verify:

Method	**Acts on**	**Effect on m_s**	**Status**
Initialization	typical amplification at $t{=}0$	RMS singular value ≈ 1	✓
Activation choice	$\|\sigma'\|_\infty$	Avoids $m_s \approx 0$	✓
Batch/Layer Norm	activations $\to m_s$	Renormalizes during training	⊙
Residual connections	Jacobian structure	Bypasses $\prod$: adds identity	✓
Auxiliary losses	Product length	Shortens $\prod$	✓

Central insight: we are studying products of random (or evolving) matrices and asking whether their singular values concentrate near 1. Initialization controls typical-case amplification; the other techniques address worst-case and dynamic drift.

Summary

- **Vanishing/exploding gradients** arise from poorly conditioned Jacobian products $\prod_i (W^i)^\top \operatorname{diag}(\sigma'(a^i))$. The per-layer multiplier $m_s = \|W^s\|_2 \cdot \|\sigma'\|_\infty$ is a worst-case diagnostic: if $m_s \neq 1$, the effect is exponential in depth.
- **Forward and backward stability are coupled but not equivalent:** forward instability poisons σ' and the backprop seed $\nabla_{o^L} f$.
- **Initialization** (Xavier, He): preserves per-coordinate activation variance (forward) and gradient norm (backward) — both yield $\mathrm{Var}(W) = 1/n_{\text{in}}$ (or $2/n_{\text{in}}$ for ReLU).
- **Activation choice** (ReLU, GELU): avoids saturation that drives $\sigma' \to 0$.
- **Batch normalization**: re-centers and re-scales pre-activations; prevents m_s drift during training. Requires $m \gg 1$.
- **Layer normalization**: batch-independent alternative; standard in Transformers (pre-norm variant).
- **Residual connections**: $o^{s+1} = o^s + F(o^s)$. Converts

Questions?

Next Time: Sequence Learning

- Learning on sequences: why standard MLPs fall short
- Recurrent neural networks (RNNs): architecture and limitations
- Sequence-to-sequence architectures
- The attention mechanism
- Tokenization for NLP problems

Gradient Coordinates Are Exchangeable at Initialization

Theorem (Gradient Coordinate Exchangeability)

Consider an L-layer network with i.i.d. weight entries at each layer (e.g., Xavier or He initialization) and coordinate-wise activation σ. Fix any input x and any layer index $s \in \{1, \dots, L-1\}$. Then the coordinates of $\nabla_{o^s} f$ are **exchangeable**: for every permutation π of $\{1, \dots, n_s\}$,

$$P_\pi \, \nabla_{o^s} f \overset{d}{=} \nabla_{o^s} f$$

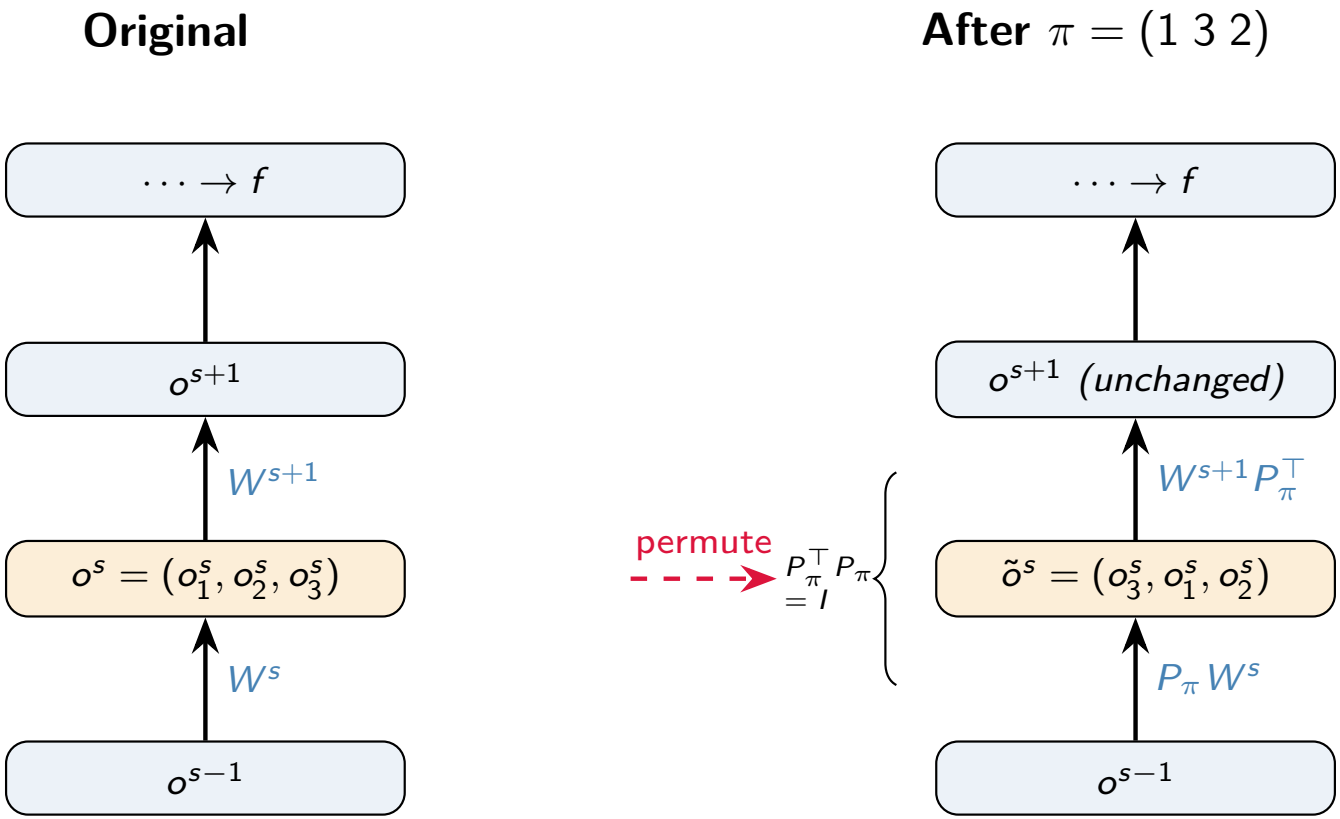
where P_π is the permutation matrix and the distribution is over the random initialization.

Consequence: all coordinates share the same marginal distribution, so

$$\mathbb{E}\big[\|\nabla_{o^s} f\|^2\big] = \sum_{k=1}^{n_s} \mathbb{E}\big[[\nabla_{o^s} f]_k^2\big] = n_s \cdot \mathbb{E}\big[[\nabla_{o^s} f]_1^2\big].$$

Idea: Exploit Neuron Permutation Symmetry

The proof exploits the fact that permuting all neurons at layer s leaves the joint distribution of the weights unchanged (because each W^i has i.i.d. entries), but permutes $\nabla_{o^s} f$ accordingly.



The permutations of W^s (rows) and W^{s+1} (columns) **cancel at layer $s+1$**: $a^{s+1} = W^{s+1} P_\pi^\top \cdot P_\pi\, o^s = W^{s+1} o^s$.

Proof

Fix a permutation π and let $P = P_\pi$. Define the permuted weights:

$$\tilde{W}^s = PW^s, \qquad \tilde{W}^{s+1} = W^{s+1}P^\top, \qquad \tilde{W}^i = W^i \text{ for } i \neq s, s{+}1.$$

Step 1 (layers $\leq s$): Since o^{s-1} depends only on $W^1, \ldots, W^{s-1}$ (unchanged), o^{s-1} is unchanged. Then $\tilde{a}^s = \tilde{W}^s o^{s-1} = P\, a^s$, so $\tilde{o}^s = \sigma(P\, a^s) = P\, o^s$ (coordinate-wise σ).

Step 2 (layers $> s$):
$\tilde{a}^{s+1} = \tilde{W}^{s+1}\tilde{o}^s = W^{s+1}P^\top P\, o^s = W^{s+1}o^s = a^{s+1}$. So $o^{s+1}, o^{s+2}, \ldots, o^L$ and hence f are all unchanged.

Step 3 (gradient): $\nabla_{o^{s+1}} f$ is unchanged, and
$\nabla_{\tilde{o}^s} f = (\tilde{W}^{s+1})^\top \operatorname{diag}(\sigma'(a^{s+1}))\, \nabla_{o^{s+1}} f =$
$P\,(W^{s+1})^\top \operatorname{diag}(\sigma'(a^{s+1}))\, \nabla_{o^{s+1}} f = P\, \nabla_{o^s} f$.

Step 4 (distribution): Since each W^i has i.i.d. entries, $PW^s \overset{d}{=} W^s$ and $W^{s+1}P^\top \overset{d}{=} W^{s+1}$. Therefore $(\tilde{W}^1, \ldots, \tilde{W}^L) \overset{d}{=} (W^1, \ldots, W^L)$, so $P\, \nabla_{o^s} f \overset{d}{=} \nabla_{o^s} f$. □

Visual Summary: What the Permutation Does

