Universal Approximation & Going Deep

Lecture 2 · 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. State the **Universal Approximation Theorem** and cite its caveats.
2. Explain why **depth beats width** in practice despite theoretical equivalence.
3. Diagnose **vanishing / exploding gradients** in deep nets.
4. Apply **residual connections** to train 100+ layer networks.
5. Pick **weight init** (Xavier / He) based on activation.
6. Articulate three **practical limits** UAT does not address.
7. Separate **expressivity**, **optimization**, and **generalization** claims.
8. Explain projection shortcuts and pre-activation residual blocks.

Recap · where we left off

- **MLPs** are stacks of affine + non-linearity (L1).
- **Backprop** is chain rule run right-to-left (L1).
- **Logistic / softmax classification = 1-layer MLP** under categorical NLL (L0 + L1).
- **Sigmoids vanish; ReLU un-blocks depth** (L1, end).
- **One hidden layer can approximate anything** (UAT) — we need to make that precise today.

> REFERENCE
>
> Today maps to **UDL Ch 4** (deep networks), **Ch 7** (gradients & init), **Ch 11** (residual networks). Read these three chapters before or after — whichever works for you.

> KEY IDEA
>
> Today's question · *if one hidden layer is universal, why do we ever go deeper?* Three answers · **width is exponential**, **depth is hierarchical**, and **depth is trainable** (with the right tools).

Three axes we must not mix up

Deep learning progress depends on three different questions.

AXIS	QUESTION	L02 ANSWER
Expressivity	Can this architecture represent the function?	UAT and depth separation
Optimization	Can SGD find useful weights?	ReLU, residuals, initialization
Generalization	Will it work on unseen data?	not guaranteed by UAT

WATCH OUT

A network can be expressive but untrainable. It can be trainable but overfit. It can fit train/val and still fail under distribution shift. Keep these axes separate in every DL paper you read.

Pop quiz · two architectures, same parameter budget

You have ~$10\,000$ parameters to spend on a regression task with 1-D input.

POP QUIZ

(a) **Wide-and-shallow** · 1 hidden layer with 5,000 ReLU units.
(b) **Tall-and-thin** · 50 hidden layers with 14 ReLU units each.

Which one would you bet on for fitting a *complex* function like $\sin(50x)$ on $[0, 1]$?

Stop and decide. We'll come back to this exact question when we hit Telgarsky's separation — your gut answer should change once you've seen the proof.

This is L02's central tension · **width** is universal but expensive, **depth** is exponentially more efficient *when it can be trained*. Today we earn both halves of that sentence.

PART 1

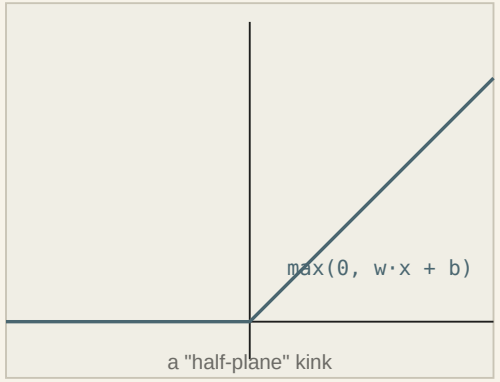
Universal approximation

What a single hidden layer can — and can't — do

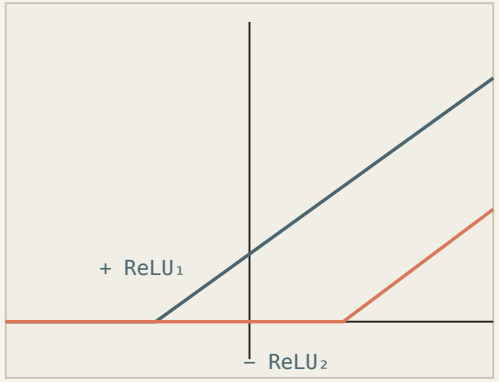
Build a bump from two ReLUs

Two ReLUs = one bump. Many bumps = any curve.

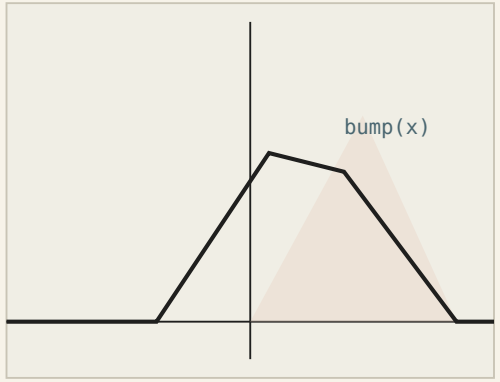
STEP 1 · ONE RELU



STEP 2 · SUBTRACT ANOTHER RELU



STEP 3 · A LOCALIZED BUMP



TAKEAWAY

*Place **N bumps** along the x-axis, weight them by α_i, and you can get ε-close to any continuous function.*

In *d* dimensions: need ~$O(1/\varepsilon^{d/2})$ bumps — the curse of dimensionality (why we need depth, coming up).

IN PRACTICE

▶

Interactive: grow a 1-hidden-layer net and watch it fit a target curve — universal-approximation.

The "LEGO brick" idea · UAT in plain English

KEY IDEA

Imagine an unlimited supply of LEGO bricks. Can you build a sculpture of *anything*? A car, a house, the Eiffel Tower? **Yes** · if your bricks are small enough, you can approximate any shape.

UAT says · a neural network with one hidden layer can do the same for **mathematical functions**. Its "LEGO bricks" are simple functions built from neurons.

UAT · unpacking the statement

DERIVATION

"For any continuous function $f(\mathbf{x})$..."
The "true" relationship in our data · e.g., `price = f(house_features)`.

"...and any small error $\epsilon > 0$..."
How close we want our approximation. Set $\epsilon = 0.01$ or whatever you need.

"...there exists a network..."
With one hidden layer of N neurons. The theorem guarantees · for any ϵ, some N exists.

"...whose output is within ϵ of f."
$|f(\mathbf{x}) - \sum_{i=1}^{N} \alpha_i \, \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i)| < \epsilon$

A weighted sum of neuron outputs · each neuron is a "LEGO brick."

UAT · the formal statement

DERIVATION

Theorem (Cybenko 1989 · Hornik 1991 · Leshno 1993)

For any continuous $f : [0,1]^d \to \mathbb{R}$ and any $\epsilon > 0$, there exist N, weights $\{\mathbf{w}_i, b_i, \alpha_i\}$ such that

$$\left| f(\mathbf{x}) - \sum_{i=1}^{N} \alpha_i \, \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i) \right| < \epsilon$$

for any non-polynomial activation σ — including ReLU.

One hidden layer suffices. The catch hides in one word: **exist.**

UAT · the proof in three moves

We won't write a full proof — but the structure is short and worth knowing.

DERIVATION

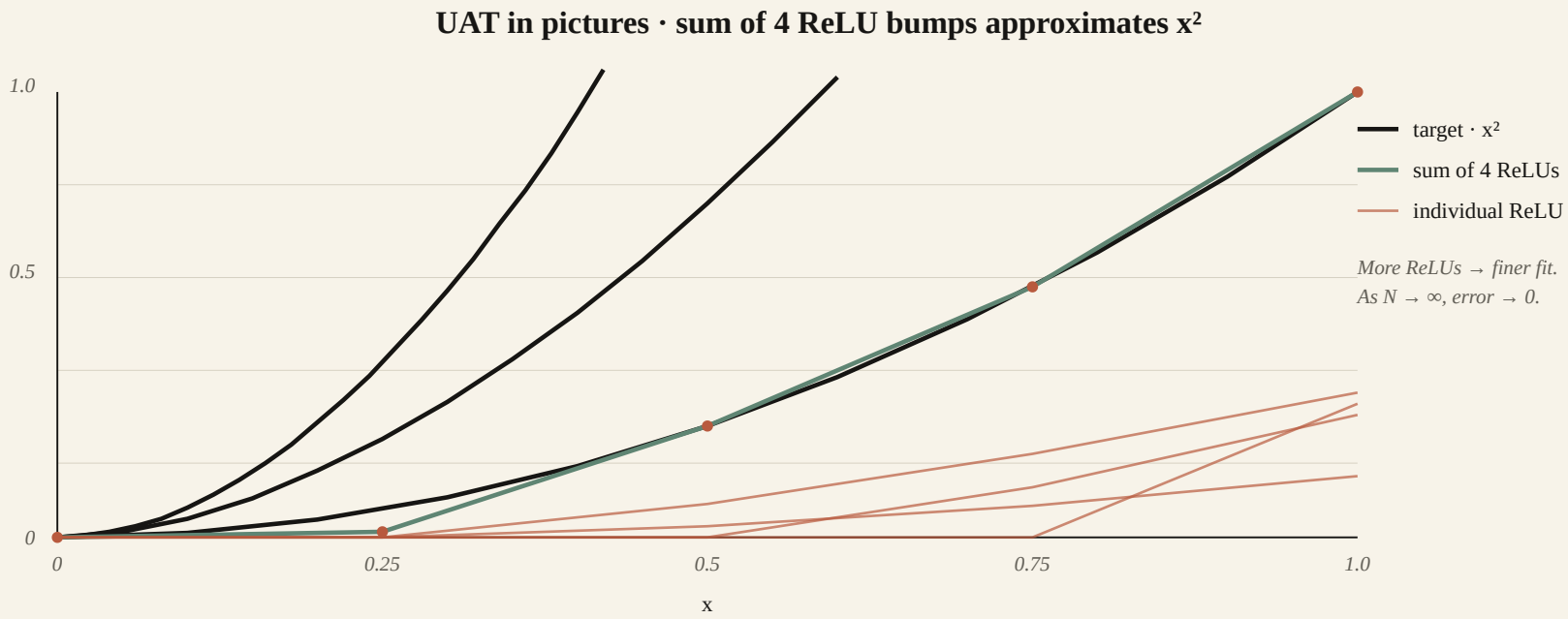
Move 1 · Approximate continuous functions by step functions. Any continuous f on $[0,1]^d$ is *uniformly continuous* (Heine–Cantor). So for any ϵ we can partition $[0,1]^d$ into small enough cells that f varies by less than $\epsilon/2$ inside each cell. Replace f by its average value on each cell · we get a step function within $\epsilon/2$ of f.

Move 2 · Approximate step functions by sums of sigmoids. A sigmoid $\sigma(w(x-b))$ with large w is essentially a step at $x=b$. A *bump* on $[a,b]$ is a difference of two such "near-steps." Each cell of the step function ⇒ one bump in the network. N cells → $2N$ hidden units.

Move 3 · Density via Stone-Weierstrass / Hahn-Banach. The space of finite sums $\sum \alpha_i \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i)$ is **dense** in $C([0,1]^d)$ — a non-trivial functional-analysis theorem. Combined with Moves 1 and 2, this gives the bound $\|f - \hat{f}\|_\infty < \epsilon$.

Bottom line · UAT is *not* a constructive recipe — it's a density theorem. It says good weights *exist*; it says nothing about N, generalization, or whether SGD finds them.

Worked example · approximate `f(x) = x²` with 4 ReLUs



Worked example · approximate `f(x) = x²` with 4 ReLUs · numbers

DERIVATION

Pick 4 ReLU bumps at $x = 0.0, 0.25, 0.5, 0.75$ on $[0, 1]$. Each is `relu(w·(x - b))` for slope $w = 1$.

RELU i	b_i	α_i	TURNS ON AT $x =$
1	0.0	0.25	0.0
2	0.25	0.50	0.25
3	0.50	0.75	0.50
4	0.75	1.00	0.75

The output is a piecewise-linear staircase that hugs x^2 . With more ReLUs, the staircase gets finer · the error $\epsilon \to 0$.

That's UAT in numbers. A weighted sum of ReLU bumps approximates any 1D continuous function.

Building a triangle bump · step-by-step

A single ReLU is a ramp going up forever. To make it come back down to zero we need **three** ReLUs.

DERIVATION

$$f(x) = \text{relu}(x-1) - 2\cdot\text{relu}(x-2) + \text{relu}(x-3)$$

x	RELU($x-1$)	$-2\cdot$RELU($x-2$)	RELU($x-3$)	SUM
0	0	0	0	**0**
1.5	0.5	0	0	**0.5**
2	1	0	0	**1.0** (peak)
2.5	1.5	-1	0	**0.5**
4	3	-4	1	**0**

A perfect triangular bump centered at $x = 2$. Place enough such bumps and you can approximate any continuous function. **This is what UAT proves.**

Two-ReLU bumps · the real building block

A single ReLU is a half-plane. Subtract two ReLUs · you get a **bump** of any width and height.

DERIVATION

$$\text{bump}(x; a, b) = \text{relu}(x - a) - \text{relu}(x - b), \quad a < b$$

This is 0 outside $[a, b]$ and rises linearly in between. Place enough bumps and you can build any continuous function · just place a bump where each fine slice is.

UAT's existence proof essentially tiles the function space with bumps. A network finds these bumps automatically through gradient descent. *Existence* is given by the construction; *training* is the open problem.

The price of width — curse of dimensionality

Piecewise-linear approximation of f to error ϵ:

- 1D: $N \approx O(1/\sqrt{\epsilon})$
- dD: $N \approx O(1/\epsilon^{d/2})$

d	$\epsilon = 0.01$	NEURONS
1		~10
10		10^{10}
100		10^{100}

WATCH OUT

UAT says good weights *exist*. Not that SGD finds them. Not that the network generalizes. Not that N is reasonable.

Three things UAT does *not* promise

Learnability.
Existence of good weights ≠ SGD finding them.

Width.
The bound on N can be astronomical.

Generalization.
A network that memorizes n training points also satisfies UAT. Works on train, fails on test.

POP QUIZ

Pop quiz. True or false: UAT guarantees a 1-hidden-layer net will perform well on unseen data, given enough neurons and data.

Pop quiz · answer

False — for three independent reasons.

1. Existence ≠ findability.
2. "Enough neurons" can mean exponentially many.
3. UAT says *nothing* about generalization.

   In practice, depth is far more parameter-efficient than width.

PART 2

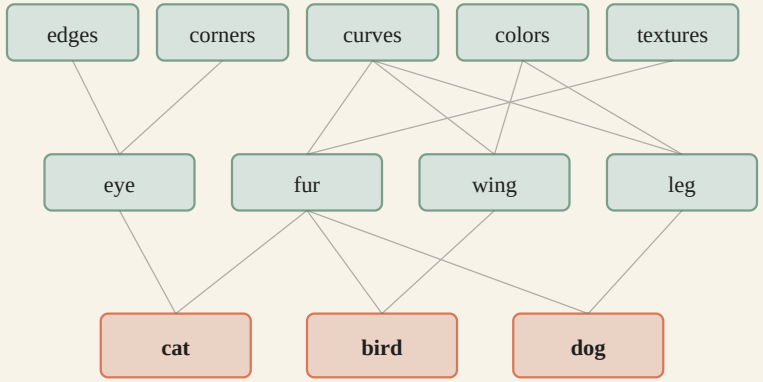
Depth vs width

Why deeper is (usually) better

Shallow enumerates, deep reuses

Compositionality: deep reuses parts, shallow enumerates cases

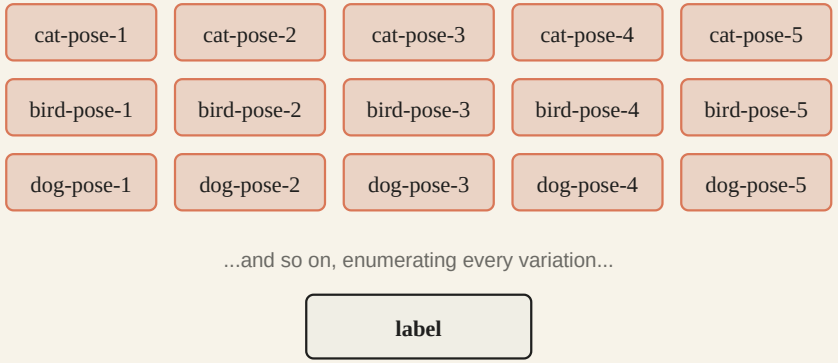
DEEP · COMPOSITIONAL



Add a **new class** "fox"? Reuse existing edges/eyes/fur — one new top-level node.

Grows linearly in classes

SHALLOW · ENUMERATIVE



Add a new class "fox"? Needs its **own** bank of pose detectors from scratch.

Grows exponentially (parity ⇒ 2^n)

*Telgarsky 2016 · there exist functions that depth-k networks compute with polynomial width, but that need **exponential width** at depth O(k^{1/3}).*

A parameter-budget exercise

Q. On a $784 \to ? \to 10$ classifier:

- **Wide shallow:** 1 hidden layer of 2048 units
- **Deep narrow:** 8 hidden layers of 128 units

Which has more parameters?

Answer — and the twist

ARCHITECTURE	PARAMETERS
$784 \to 2048 \to 10$	~1.63 M
$784 \to 128^{\times 8} \to 10$	~0.22 M

Shallow-wide has **7× more** parameters, but deep-narrow typically wins on natural data.

INTUITION

Parameter count is a crude proxy for capacity. What depth gives you, width alone cannot: **reusable hierarchy.**

When depth helps — and when it doesn't

Depth helps when the target function has reusable substructure.

DOMAIN	REUSABLE STRUCTURE	WHAT DEEPER LAYERS REUSE
images	edges → textures → parts → objects	local visual motifs
language	characters → words → phrases → discourse	compositional meaning
audio	samples → phonemes → syllables → words	local temporal patterns
tabular data	often weaker hierarchy	depth may help less

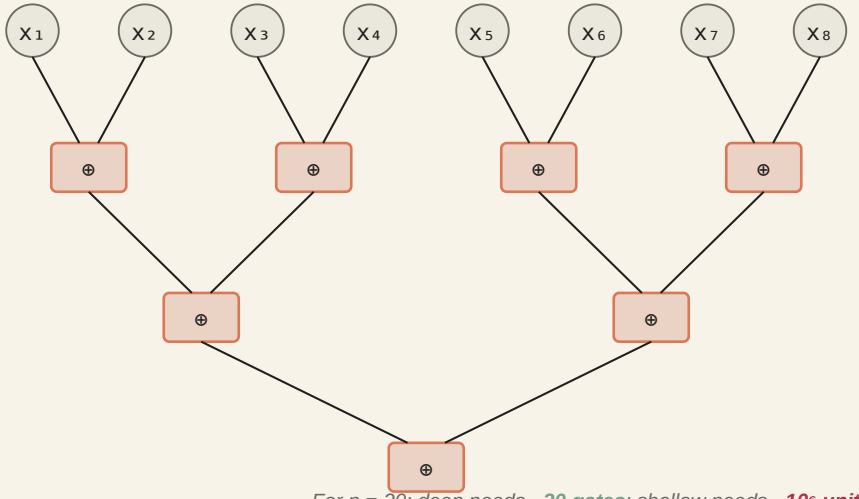
KEY IDEA

Depth is not magic. It is a strong inductive bias for **compositional** problems. If the data do not have reusable structure, a deeper model can simply be harder to optimize.

Parity — the canonical case

Parity of n bits · depth matches the recursion → O(n) neurons; shallow → O(2^n)

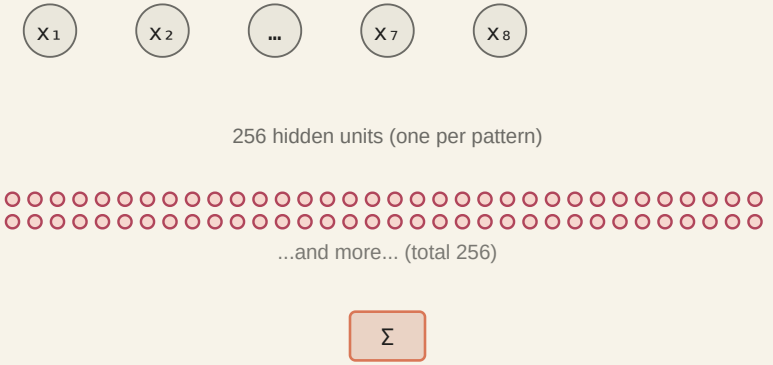
DEEP TREE · $\log_2(8) = 3$ LAYERS, 7 XOR GATES



output · 1 if odd number of 1s

7 gates · depth 3

SHALLOW · ENUMERATE ALL $2^8 = 256$ PATTERNS



output · sum of detectors matching odd-count patterns

256 units · depth 1

*For n = 20: deep needs **~20 gates**; shallow needs **~10^6 units**. Depth is how you **match structure to structure**.*

Why depth helps for parity

Parity is **recursive** — XOR of pairs, then XOR of pair-pairs, then XOR of those.

Depth matches the structure of the problem:

- $\log n$ **layers**, $O(n)$ gates.
- Shallow has no hierarchy to exploit — it must enumerate every pattern.

> REFERENCE
>
> Formal proof: Telgarsky, *"Benefits of Depth in Neural Networks,"* COLT 2016. We take the statement on faith today.

Depth-vs-width · the formal separation

REFERENCE

Theorem (Telgarsky 2016, simplified) · There exists a function $f : [0,1] \to [0,1]$ representable by a ReLU network of depth $2k$ and width ≤ 3, such that **any** ReLU network of depth $\leq k$ approximating f within constant error needs **at least** 2^k units.

The witness function is the k-fold composition of the **sawtooth** ·

$$T(x) = \begin{cases} 2x & x \leq 1/2 \\ 2 - 2x & x > 1/2 \end{cases}$$

Each composition doubles the number of "teeth" — depth k gives 2^k teeth using $O(k)$ ReLUs. A shallow net **must enumerate** every tooth · exponential width.

KEY IDEA

Depth is exponentially more parameter-efficient than width for problems with compositional / recursive structure. Real-world data (images, language) is richly compositional · so depth pays off in practice.

Enough theory — can we just stack layers?

Q. If depth is so great, should we train a 500-layer plain MLP?

Give an honest answer; then read on.

Two things break

WATCH OUT

Problem 1 — vanishing gradients (from L1).
Product of many sub-1 Jacobians → zero.

Problem 2 — the degradation problem.
Something strictly more surprising. Next section.

PART 3

Vanishing gradients · the full picture

Before we fix depth, let's see it break

Backprop · term-by-term, no Jacobians

A 4-layer scalar network · `y = w_4 · w_3 · w_2 · w_1 · x` (ignoring activations).

We want gradient of loss L with respect to the **first** weight w_1. Chain rule, one link at a time:

DERIVATION

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w_1}$$

Expand $\partial y / \partial w_1$:

- $\partial y / \partial (\text{layer 3 out}) = w_4$
- $\partial (\text{layer 3 out}) / \partial (\text{layer 2 out}) = w_3$
- $\partial (\text{layer 2 out}) / \partial (\text{layer 1 out}) = w_2$
- $\partial (\text{layer 1 out}) / \partial w_1 = x$

So · $\dfrac{\partial L}{\partial w_1} = \dfrac{\partial L}{\partial y} \cdot (w_4 \cdot w_3 \cdot w_2) \cdot x$

The key is the **product** $w_4 \cdot w_3 \cdot w_2$. If any of these is small (< 1), the product shrinks fast.

Numeric · vanishing with sigmoids

Sigmoid derivative · $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ · max value **0.25**.

Each backward step multiplies by $w_l \cdot \sigma'$. Assume weights ≈ 1.0:

DERIVATION

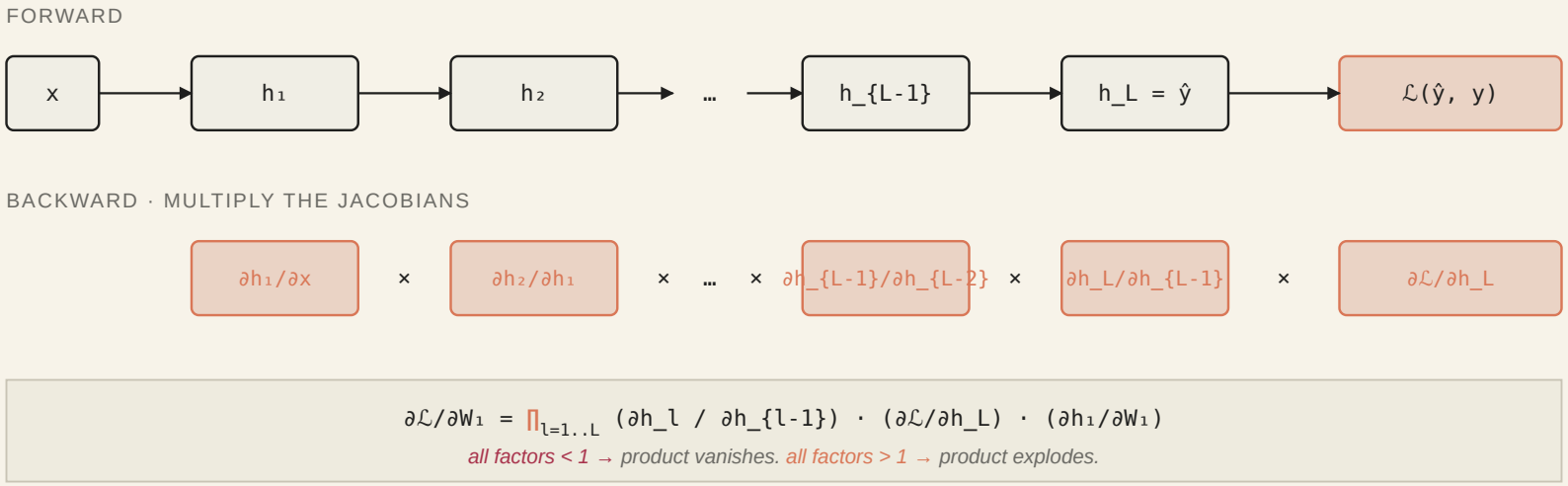
LAYER (COUNTING BACK)	CUMULATIVE FACTOR
L	0.25
L−1	$0.25^2 = 0.0625$
L−2	$0.25^3 = 0.0156$
L−5	$0.25^5 \approx 10^{-3}$
L−10	$0.25^{10} \approx 10^{-6}$

After 10 sigmoid layers, the gradient signal at the **earliest** weight is shrunk by a million. **The first layer barely updates · learning stalls.**

This is the core problem ResNets and ReLU were invented to solve.

The chain rule is a product

Chain rule = product of Jacobians · one factor per layer



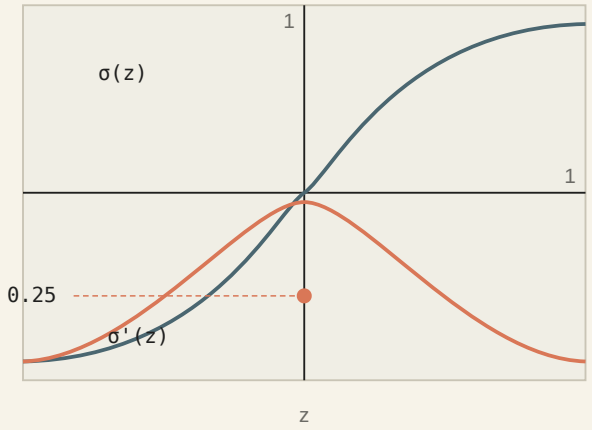
$$\partial \mathcal{L}/\partial W_1 = \prod_{l=1..L} (\partial h_l / \partial h_{l-1}) \cdot (\partial \mathcal{L}/\partial h_L) \cdot (\partial h_1/\partial W_1)$$

all factors < 1 → product vanishes. all factors > 1 → product explodes.

Sigmoid's fatal ceiling · 0.25

Why sigmoids vanish — the gradient ceiling is 0.25, then you multiply them

SIGMOID & ITS DERIVATIVE



CHAIN-RULE PRODUCT

$$\partial L/\partial W_1 \approx \prod \sigma'(\cdot)$$

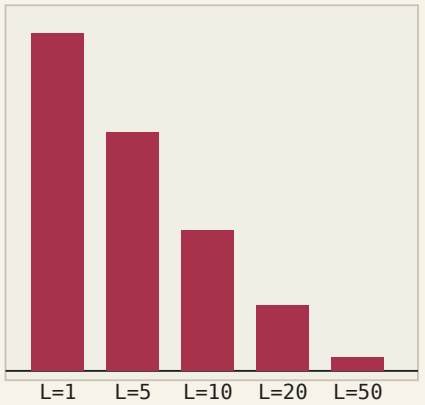
each factor ≤ 0.25

$$0.25 \times 0.25 \times 0.25 \times \ldots \times \mathbf{0.25}$$

$$= \mathbf{0.25^L}$$

shrinks exponentially in depth

GRADIENT MAGNITUDE



log-scale — early layers get ≈ 0 signal

*For 20 layers · $0.25^{20} \approx 10^{-12}$ · the gradient is literally lost in floating-point noise. This is **why depth was blocked for 20 years**.*

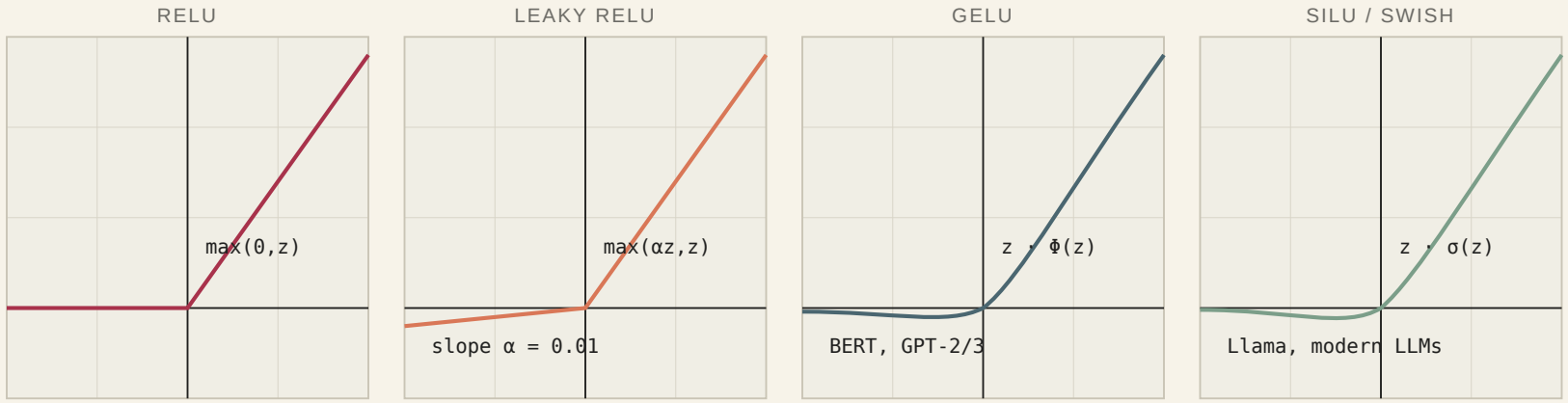
IN PRACTICE

▶

Interactive: stack sigmoids and watch the gradient evaporate with depth — vanishing-gradients.

Fix #1 · the ReLU family

The ReLU family — four shapes, same "let positive signal through" idea



ALL FOUR SHARE

✓ gradient = 1 on the active side → no vanishing with depth

HOW THEY DIFFER

smoothness near 0 (matters for optimization on Transformers)

PART 4

The degradation problem & ResNets

The most important architectural idea since backprop

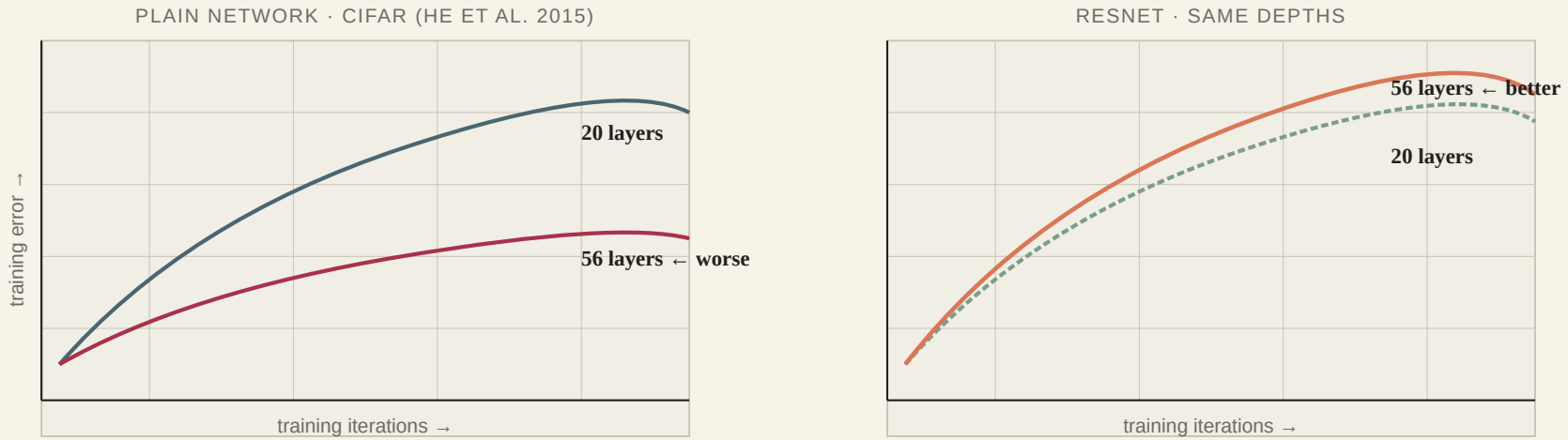
The He et al. (2015) experiment

Train plain CNNs of increasing depth on CIFAR-10 — same optimizer, same init, just more layers.

Q. What do you expect? More layers → more capacity → better, right?

The surprise

The degradation problem — deeper plain nets train worse



*Not overfitting — **training** error got worse. An optimization problem, fixed by adding a skip path.*

Why this should bother you

INTUITION

If this were overfitting, training error would **drop** with depth (more capacity to memorize) and test error would **rise** (poor generalization).

Instead, training error went *up*. The deeper net cannot even fit the training data.

This is an **optimization** problem, not a capacity problem.

Degradation is not overfitting

Compare the signatures:

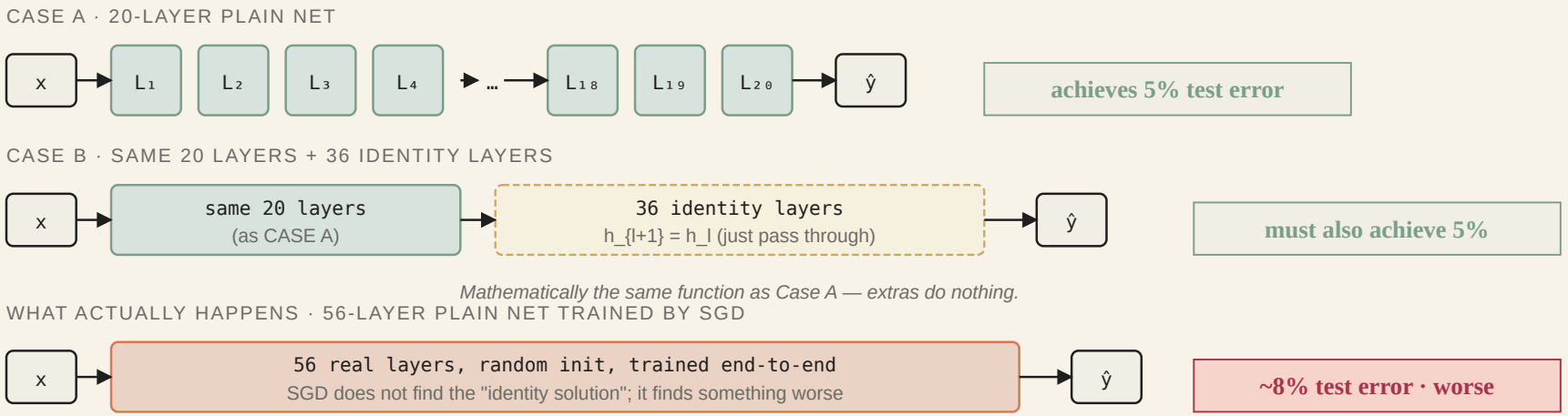
FAILURE MODE	TRAINING ERROR	VALIDATION / TEST ERROR	MAIN DIAGNOSIS
Underfitting	high	high	model or training too weak
Overfitting	low	high	generalization failure
Degradation	higher for deeper model	higher for deeper model	optimization failure

KEY IDEA

The ResNet paper mattered because it exposed a surprising fact: simply adding layers can make the **training objective itself** harder, even though the deeper model has more representational capacity.

The thought experiment

Thought experiment · why the degradation problem should be impossible



*Not a capacity problem — the capacity is **larger**. An **optimization** problem.*

Why "learning the change" is easier · steering analogy

KEY IDEA

Imagine steering a car by giving the wheel an *absolute* angle (e.g., "set wheel to 27 degrees from zero"). Hard to do.

Instead, you say "**turn a little right**" or "**stay straight.**" Much easier.

ResNets do the same · they reframe each layer's job from *"output the right thing"* to *"output a small change to your input."* The default action — change nothing, pass input through — is now trivial. The network only learns *deviations* from identity, which is much easier for SGD.

ResNet · the key insight

KEY IDEA

He et al. 2015 — don't ask the block to learn the full mapping $H(\mathbf{x})$. Ask it to learn the **residual**:

$$\mathcal{F}(\mathbf{x}) = H(\mathbf{x}) - \mathbf{x} \implies H(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$$

If the optimum is close to identity, we only need $\mathcal{F} \approx \mathbf{0}$ — which SGD finds trivially.

Why residuals are easier

- **Weight decay** pushes $\mathcal{F}$ toward zero.
- **Init near zero** starts $\mathcal{F} \approx \mathbf{0}$.
- **Small SGD updates** keep it there unless signal says otherwise.

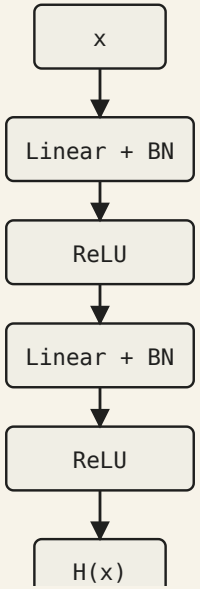
Coordinating non-linear layers to produce *exact* identity is the opposite: a delicate balance with no prior.

The skip connection turns a hard default into a free default.

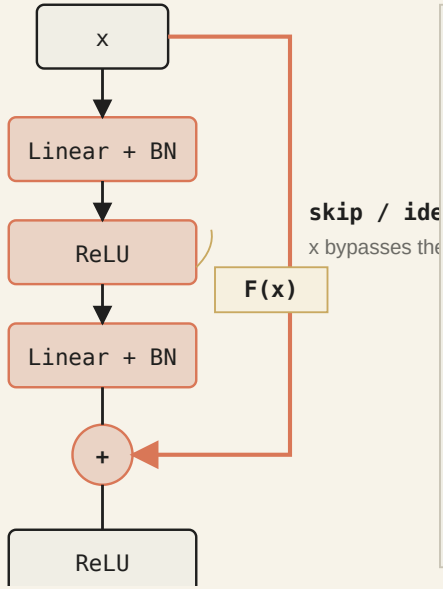
The residual block

A residual block — learn the update, not the whole mapping

PLAIN BLOCK



RESIDUAL BLOCK



skip / ide

x bypasses the

WHY IT WORKS

Forward:

`y = F(x) + x`

Jacobian:

`∂y/∂x = ∂F/∂x + I`

Consequence:

even if ∂F/∂x collapses to 0, the identity term I still carries gradient back to early layers.

If the optimum ≈ identity:

only need F ≈ 0 (easy).
vs. learning H ≈ identity through nonlinearities (hard).

BatchNorm in the ResNet story

The original ResNet block was not just "add a skip connection." It used a stack like:

$$\text{Conv} \rightarrow \text{BatchNorm} \rightarrow \text{ReLU} \rightarrow \text{Conv} \rightarrow \text{BatchNorm} \rightarrow +x \rightarrow \text{ReLU}$$

COMPONENT	WHAT IT HELPS WITH
Skip connection	direct signal and gradient path
BatchNorm	stable activation scale across layers
ReLU	nonlinearity with active-side gradient 1
He initialization	variance scale matched to ReLU

KEY IDEA

Do not learn the wrong lesson: ResNets train because several engineering choices work together. The skip connection is the central idea, but scale control is part of the system.

How the skip-connection makes a gradient highway

Forward · $\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{x}$.

We want the gradient flowing back through the block · $\partial L / \partial \mathbf{x}$.

DERIVATION

By the chain rule · $$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

Compute the local gradient:
$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial}{\partial \mathbf{x}}(\mathcal{F}(\mathbf{x}) + \mathbf{x}) = \frac{\partial \mathcal{F}}{\partial \mathbf{x}} + \mathbf{I}$$

So · $$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \cdot \left(\frac{\partial \mathcal{F}}{\partial \mathbf{x}} + \mathbf{I}\right) = \frac{\partial L}{\partial \mathbf{y}} \frac{\partial \mathcal{F}}{\partial \mathbf{x}} + \underbrace{\frac{\partial L}{\partial \mathbf{y}}}_{\text{the highway!}}$$

The "+I" gives an **uninterrupted express lane** for the gradient · the second term is the original signal **passing straight through** even if the first term vanishes. The early layers always get a clean signal.

Numeric · gradient flow with vs without skip

Same 5-layer net. Each layer's $\partial \mathcal{F} / \partial \mathbf{x}$ has tiny norm 0.1 (saturated regime).

DERIVATION

LAYER	PLAIN (MULTIPLICATIVE)	RESNET (ADDITIVE)
L (output)	1.0	1.0
L−1	0.1	1.1
L−2	0.01	1.21
L−3	0.001	1.33
L−4	**0.0001**	**1.46**

The plain net's signal vanishes · ResNet keeps the full unit signal **plus** a small contribution from each block. **Hundred-layer ResNets train; 100-layer plain nets don't.**

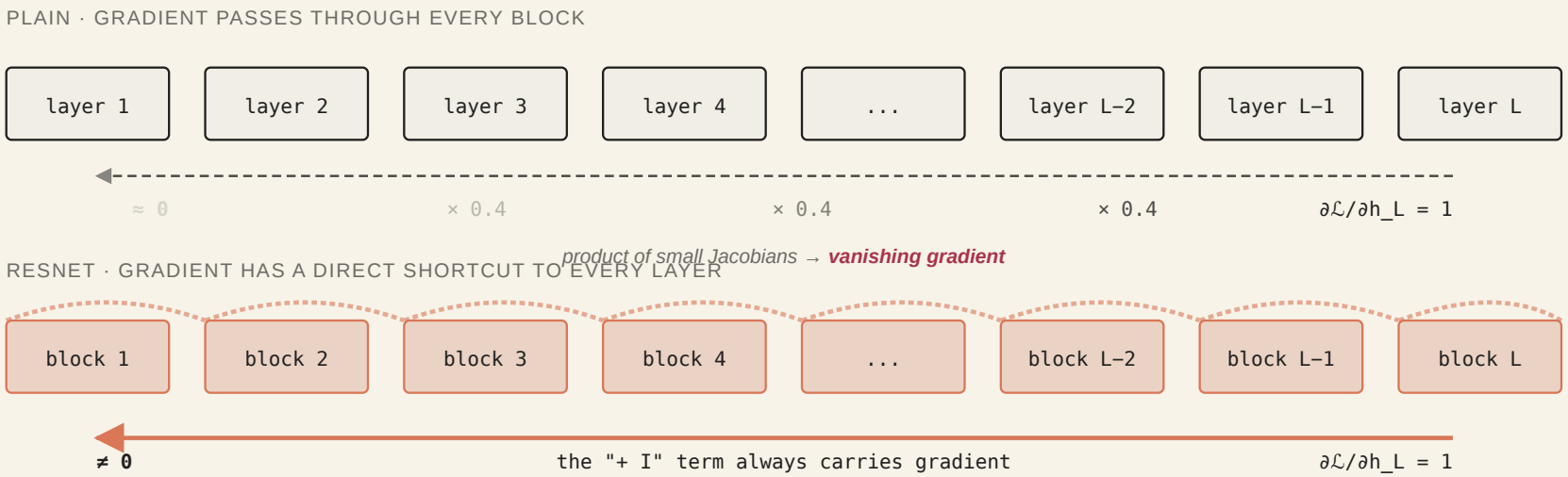
Skip connections fix gradient flow

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{x} \implies \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathcal{F}}{\partial \mathbf{x}} + \mathbf{I}$$

Even if $\partial \mathcal{F} / \partial \mathbf{x}$ collapses to zero, the identity $\mathbf{I}$ survives — a direct path back to every early layer.

Gradient highway · plain vs ResNet

The skip connection is a gradient highway



Precisely: $\partial y/\partial x = \partial F/\partial x + I$. The identity term never shrinks — it's a parallel path, not a per-block multiplier.

ResNet in PyTorch · 12 lines

```
class ResidualBlock(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.block = nn.Sequential(
            nn.Linear(dim, dim), nn.BatchNorm1d(dim), nn.ReLU(),
            nn.Linear(dim, dim), nn.BatchNorm1d(dim),
        )
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.relu(self.block(x) + x)    # ← the skip
```

Q. What constraint does `self.block(x) + x` impose on dimensions?

Projection shortcuts when shapes change

The addition requires matching shapes. If the block changes width, channels, or spatial resolution, the skip path must also transform x.

```
class ResidualBlock(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.block = nn.Sequential(
            nn.Linear(d_in, d_out), nn.ReLU(),
            nn.Linear(d_out, d_out),
        )
        self.skip = nn.Identity() if d_in == d_out else nn.Linear(d_in, d_out)

    def forward(self, x):
        return torch.relu(self.block(x) + self.skip(x))
```

In CNNs this is often a 1×1 convolution, sometimes with stride 2, so both branches land on the same shape before addition.

Pre-activation ResNets

Later ResNets moved normalization and activation **before** the weight layers:

BLOCK STYLE	FORMULA SKETCH	WHY IT MATTERS
Post-activation	$y = \mathrm{ReLU}(x + F(x))$	original ResNet
Pre-activation	$y = x + F(\mathrm{BN}/\mathrm{ReLU}(x))$	cleaner identity path

KEY IDEA

Pre-activation keeps the skip path as close to a pure identity as possible. This makes very deep residual networks easier to optimize because the gradient highway is less obstructed.

Transformers use the same idea in another form: residual stream plus normalization around each block.

Empirical impact

YEAR	MODEL	DEPTH	IMAGENET TOP-5
2012	AlexNet	8	16.4%
2014	VGG-19	19	7.3%
2014	GoogLeNet	22	6.7%
2015	**ResNet-152**	**152**	**3.6%**

Skip connections are now in virtually every modern architecture — CNNs, Transformers, diffusion U-Nets.

PART 5

Initialization

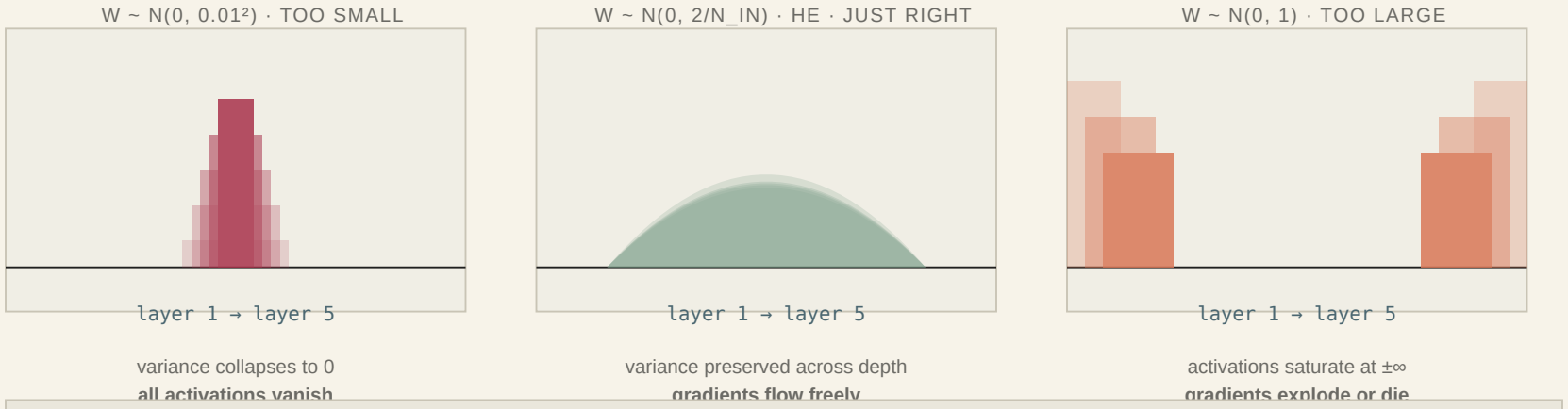
From first principles

The goal

Keep activations — and gradients — at roughly **constant variance** across layers.

- Variance grows → exploding activations.
- Variance shrinks → vanishing activations.

Initialization controls whether activations live or die



`Rule · Var(W) = 2/n_in → ReLU (He). Var(W) = 2/(n_in+n_out) → Sigmoid/Tanh (Xavier).`

Derivation: layer variance = n_in · Var(W) · Var(x). ReLU kills half, so multiply numerator by 2.

What to check in a real model

Initialization theory is useful because it gives a concrete diagnostic: **activation statistics by layer**.

For one batch, log:

QUANTITY	HEALTHY EARLY SIGNAL	RED FLAG
activation mean	near 0 for normalized layers	large drift
activation std	roughly stable across depth	shrinks to 0 or explodes
gradient norm	nonzero in early layers	first layers near 0
fraction ReLU active	neither 0% nor 100%	many dead units

```
for name, p in model.named_parameters():
    if p.grad is not None:
        print(name, p.grad.norm().item())
```

KEY IDEA

Before changing architecture, check whether the signal survives the forward pass and the gradient survives the backward pass.

Why initialization matters · the failure mode

If we initialize weights from $\mathcal{N}(0, 1)$ in a deep ReLU net:

Activations

Each layer multiplies by a matrix of N(0,1) values. Activation magnitudes either **grow exponentially** with depth (explode) or **shrink exponentially** (vanish) depending on shape.

Gradients

Same product, going backwards. A single bad scale → all early layers train at 10^{-30} effective rate · they never move.

WATCH OUT

Symptom · loss is NaN at step 1, or loss flat with all weights stuck. Always-and-only an init problem if the loop is otherwise correct.

The variance argument on the next slide gives a one-line fix · scale init by $1/\sqrt{n_{\text{in}}}$.

Vanishing gradient · numeric example

Suppose · 10-layer sigmoid network. Sigmoid derivative max is 0.25 (at $x = 0$).

DERIVATION

Even at the *best* point, gradient through one layer multiplies by ≤ 0.25.

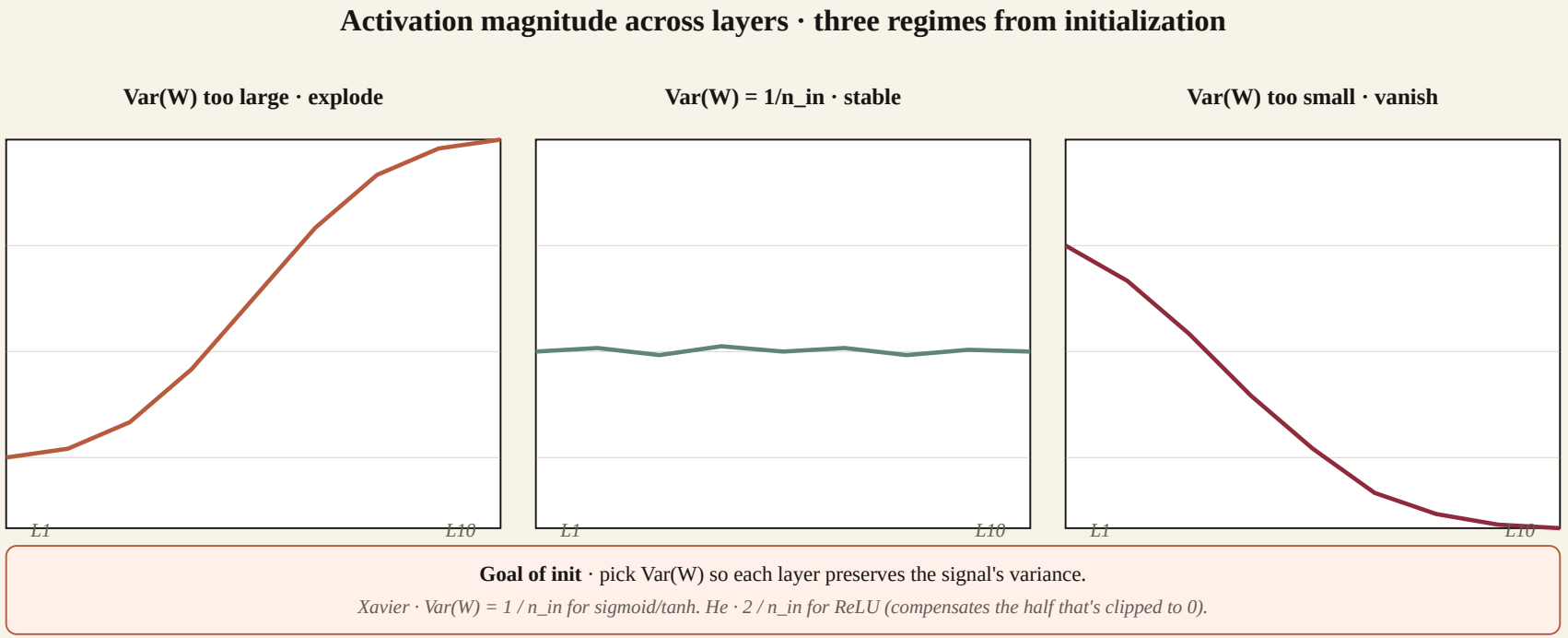
After 10 layers · $0.25^{10} \approx 10^{-6}$

After 20 layers · $0.25^{20} \approx 10^{-12}$

The first layer's effective learning rate is **a million times smaller** than the last layer's. It barely updates · network never learns features in early layers.

This is the practical reason ReLU (derivative = 0 or 1) replaced sigmoid in deep nets · it doesn't shrink the gradient by a factor every layer.

Variance · three regimes across layers



Goal of init · pick Var(W) so each layer preserves the signal's variance.

Xavier · Var(W) = 1 / n_in for sigmoid/tanh. He · 2 / n_in for ReLU (compensates the half that's clipped to 0).

Forward-pass variance

Layer: $y = \sum_{i=1}^{n_{\text{in}}} w_i \, x_i$. Assume w_i, x_i independent, zero-mean.

For independent zero-mean A, B:

$$\operatorname{Var}(AB) = \operatorname{Var}(A)\operatorname{Var}(B)$$

Therefore:

$$\operatorname{Var}(y) = n_{\text{in}} \cdot \operatorname{Var}(w) \cdot \operatorname{Var}(x)$$

To preserve variance we need $n_{\text{in}} \cdot \operatorname{Var}(w) = 1$.

Xavier · for sigmoid/tanh

Forward: $\mathrm{Var}(w) = 1/n_{\text{in}}$.
Backward: $\mathrm{Var}(w) = 1/n_{\text{out}}$.

DERIVATION

Xavier / Glorot (2010) — compromise:

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

He · for ReLU

ReLU zeros half the pre-activations:

$$E[h^2] = \tfrac{1}{2}\,\mathrm{Var}(y)$$

To keep this constant across layers:

DERIVATION

He / Kaiming (2015):

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

Factor of 2 compensates the ReLU halving.

Worked numeric · variance flow over 10 layers

10-layer fully-connected ReLU net with $n_{\text{in}} = 512$ everywhere. Initial activation variance $\text{Var}(x) = 1$.

DERIVATION

Naive init $W \sim \mathcal{N}(0, 1)$ (no scaling) ·

- Each layer · $\text{Var}(z) = n_{\text{in}} \cdot \text{Var}(W) \cdot \text{Var}(x) = 512 \cdot 1 \cdot 1 = 512$.
- After ReLU · ≈ 256.
- Pass through layer 2 · $\text{Var}(z) = 512 \cdot 1 \cdot 256 = 131{,}072$. **Explodes.**
- After 10 layers · $(256)^{10} \approx 10^{24}$. NaN at step 1.

He init $W \sim \mathcal{N}(0,\, 2/512)$ ·

- Each layer · $\text{Var}(z) = 512 \cdot (2/512) \cdot 1 = 2$.
- After ReLU · ≈ 1.
- Stable for 10, 100, or 1000 layers. ✓

This is **why initialization is not optional**. Bad init → loss is NaN at step 1, or weights are stuck at 10^{-30} scale and never move. Good init keeps signal magnitude constant across depth.

Initialization in PyTorch

```
# Default for nn.Linear — Kaiming uniform (sensible for ReLU)
model = nn.Linear(784, 256)

# Explicit He init
nn.init.kaiming_normal_(model.weight, mode='fan_in', nonlinearity='relu')
nn.init.zeros_(model.bias)

# Explicit Xavier (for sigmoid/tanh)
nn.init.xavier_normal_(model.weight)
```

INTUITION

Rule of thumb · ReLU family → He, sigmoid / tanh → Xavier.

POP QUIZ

Pop quiz. You build a 10-layer MLP with **Tanh** activations and **He** initialization. Loss oscillates, activations saturate. Why?

Pop quiz · answer

He doubles variance to compensate for ReLU's halving. Tanh doesn't halve — He gives **too large** a variance → activations saturate at ± 1 → gradients die.

Fix: Xavier.

ACTIVATION	INIT
ReLU, Leaky	He
Sigmoid, Tanh	Xavier
GELU, SiLU	He (convention)

Putting it all together · the L02 master sentence

DERIVATION

Depth is mathematically expressive (UAT, Telgarsky), but practically fragile.
Three forces conspire against naive deep nets · vanishing/exploding **gradients**, vanishing/exploding **activations**, and the **degradation problem** even when both are tame.

SYMPTOM	ROOT CAUSE	FIX INTRODUCED
Gradient → 0 in early layers	σ' ceiling 0.25 + product of small Jacobians	ReLU family · skip connections
Activation variance blows up / shrinks	Wrong init scale per layer	Xavier (tanh) · He (ReLU)
Deeper net is *worse* at training loss	Optimization, not capacity	Residual blocks $\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x})$

The single insight underneath all three fixes · **make every layer easy to leave alone.** Identity-friendly initialization, identity skip-connections, and activations that pass gradients through unchanged in their linear regime. That's what made depth practical in 2015 and onward.

Practice problems

DERIVATION

P1. UAT says $f(x) = x^2$ on $[0, 1]$ can be approximated to error ϵ by a sum of N ReLUs. Estimate N as a function of ϵ. (Hint · piecewise-linear with N breakpoints has error $\sim 1/N^2$ for smooth f.)

P2. A 5-layer plain MLP with sigmoid activations is failing to train. Without changing the architecture, name **two** changes that would help and explain why each works.

P3. Show that for a ResNet block $\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x})$, $\partial \mathbf{y}/\partial \mathbf{x} = I + \partial \mathcal{F}/\partial \mathbf{x}$. Use this to argue that even if $\mathcal{F}$ has tiny Jacobian, gradient through the residual block does not vanish.

P4. A 100-layer ReLU MLP with $n_{\text{in}} = 256$ everywhere uses Xavier init $W \sim \mathcal{N}(0, 1/n_{\text{in}})$. Will the activation variance grow, shrink, or stay constant? Why is this wrong for ReLU? What's the fix?

P5. Telgarsky's separation says depth-$2k$ ReLU nets need $\geq 2^k$ units to be matched by depth-k nets. Plug in $k = 10$ · how many shallow units? Why does this argue for going deep?

P6. You replace ReLU with leaky ReLU ($\max(0.01z, z)$) in a 50-layer net. (a) What changes for forward-pass variance? (b) What changes for vanishing gradients on initially-negative pre-activations?

Summary · Lecture 2 — summary

- **UAT** is an existence theorem. Width can be exponential; depth is the practical knob.
- **Compositionality** (and parity) shows depth can replace exponential width.
- **Vanishing gradients** come from products of sub-1 Jacobians; ReLU unlocks depth by giving gradient 1 on the active side.
- **Degradation problem** — plain deep nets *train* worse.
- **ResNets** — $\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{x}$. Identity-in-the-Jacobian gradient highway + smoother landscape.
- **ResNet practice** — BatchNorm, projection shortcuts, and pre-activation blocks keep the identity path usable.
- **Xavier / He** — both derived from variance preservation.
- **Always separate axes:** expressivity, optimization, and generalization are different claims.

Read before Lecture 3

Prince — Ch 4, Ch 11. Free at udlbook.github.io.

Next lecture

Tensors, autograd, `nn.Module`, `DataLoader`, the full training recipe, debugging ladder, error analysis.

NOTEBOOK

Notebook 2 · `02-depth-and-resnets.ipynb` — shallow-wide vs deep-narrow on spirals; build a residual block; visualize gradient norms across depth.