SCALER School of Technology × Priyansh Saxena

# Backpropagation and Computational Graphs

## Chain Rule • Computational Graphs • Backprop Algorithm • Gradient Flow

Priyansh Saxena

Scaler School of Technology | Deep Learning I: Neural Networks | Week 2 — Session 2/3

8 May 2026

# Recap & Today's Question

## Session 2: The *How* (Forward)

- A neuron = weighted sum + bias + activation
- MLPs stack layers to learn complex boundaries
- The forward pass: matrix multiplications

## Session 3: The *How* (Learning)

- How does a network **learn**?
- How does each weight know which direction to move?
- How do gradients flow backward?

**Today: we learn backpropagation — the algorithm that made deep learning possible.**

# The Learning Problem

We want to minimize the loss $L$.

The loss depends on the weights **indirectly**:

$$\underbrace{w}_{\text{weight}} \to \underbrace{z}_{\text{pre-act}} \to \underbrace{h}_{\text{activation}} \to \cdots \to \underbrace{\hat{y}}_{\text{output}} \to \underbrace{L}_{\text{loss}}$$

We need $\frac{\partial L}{\partial w}$ for every weight $w$.

The tool: **the chain rule**.

## The Gradient

$\frac{\partial L}{\partial w}$ tells us:

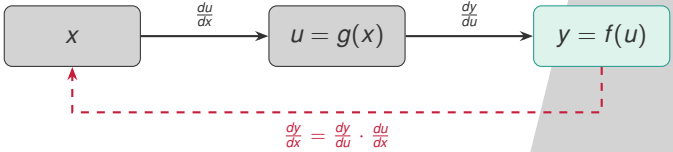
"If I wiggle this weight by a tiny amount, how much does the loss change?"

**Direction + magnitude** of the steepest change.

# The Chain Rule (Single Variable)

If $y = f(g(x))$, then:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}, \quad u = g(x)$$




**The derivative of the composition = product of derivatives along the chain.**

# Worked Example: $f(x) = \sin(x^2)$

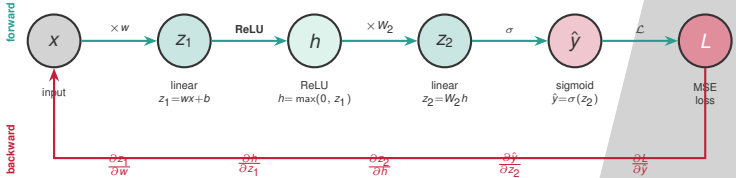
1. Let $u = x^2$, then $f = \sin(u)$
2. $\dfrac{df}{du} = \cos(u), \quad \dfrac{du}{dx} = 2x$
3. $\dfrac{df}{dx} = \cos(x^2) \cdot 2x = 2x\cos(x^2)$

## Verify at $x = 1$

$$\left.\frac{df}{dx}\right|_{x=1} = 2 \cdot \cos(1) \approx 2 \cdot 0.5403 = 1.081$$

Numerical: $\dfrac{f(1.0001) - f(0.9999)}{0.0002} \approx 1.081$ ✓

# Multi-Variable Chain Rule




**Each factor = one local gradient:**

$$\frac{\partial z_1}{\partial w} = x$$

$$\frac{\partial h}{\partial z_1} = \mathbf{1}_{z_1>0}$$

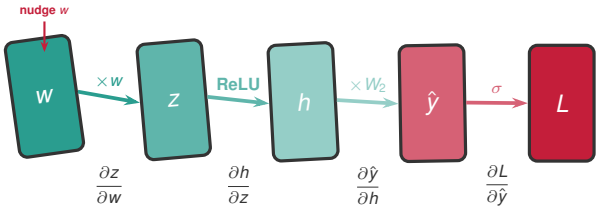
$$\frac{\partial \hat{y}}{\partial z_2} = \sigma(z_2)(1-\sigma(z_2))$$

$$\frac{\partial L}{\partial \hat{y}} = -(y-\hat{y})$$

**One hard problem → many easy ones.**

Each node only needs its own local gradient.

# The Domino Analogy




**Nudge $w$ slightly $\rightarrow$**

$z$ changes $\rightarrow$
$h$ changes $\rightarrow$
$\hat{y}$ changes $\rightarrow$
$L$ changes.

**How much does $L$ change?**

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h} \cdot \frac{\partial h}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Multiply all the local effects together.

# Quiz 1

**Q1** Chain Rule

Compute $\frac{df}{dx}$ for

$$f(x) = (2x + 1)^3$$

Use the chain rule step by step.

**Q2** Multiply Local Gradie

$L = f(g(h(w)))$

$f' = 0.5,\ g' = 3,\ h' = 2$

What is $\frac{dL}{dw}$?

**Q3** Gradient Chain Length

In a **5-layer** network, roughly how many local gradient terms are multiplied to compute $\frac{\partial L}{\partial W_1}$?

*Take 2 minutes — discuss with your neighbour.*

# Quiz 1 — Answers

## A1

Let $u = 2x + 1$.

$$\frac{df}{dx} = 3u^2 \cdot \frac{du}{dx}$$

$$= 3(2x+1)^2 \cdot 2$$

$$= \mathbf{6(2x+1)^2}$$

## A2

$$\frac{dL}{dw} = f' \cdot g' \cdot h'$$

$$= 0.5 \times 3 \times 2$$

$$= \mathbf{3.0}$$

Total = product of locals.

## A3

$\approx$ **10** terms

2 per layer (linear + activation) $\times$ 5 layers.

Deeper weights $\rightarrow$ longer chains.

# What Is a Computational Graph?

A **directed acyclic graph (DAG)** that represents a computation.

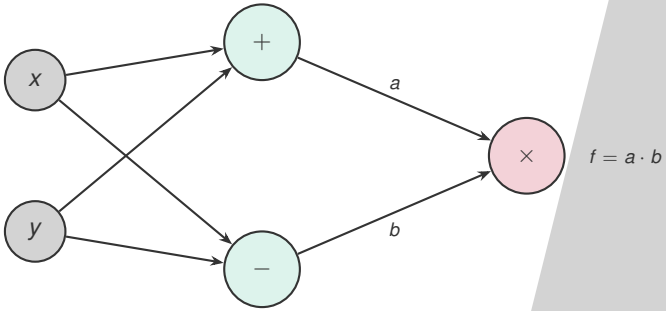
- Nodes = **operations** ($+, \times, \sigma, \text{ReLU}$)
- Edges = **data flow**
- **Forward pass**: evaluate left $\rightarrow$ right
- **Backward pass**: propagate gradients right $\rightarrow$ left

**Why?**

Makes the chain rule **mechanical**.
No staring at complex formulas.
Just follow the arrows.

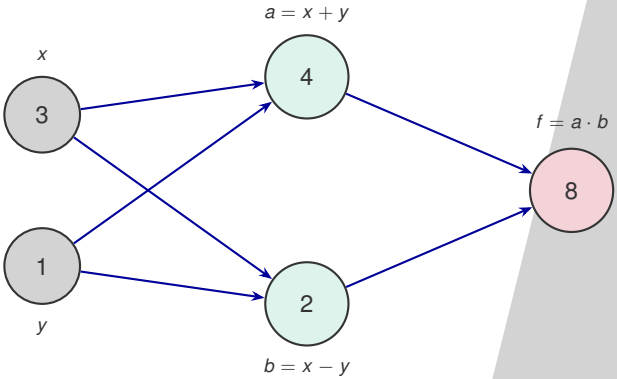
Every DL framework (PyTorch, TF, JAX) builds and differentiates computational graphs.

# Example: $f(x, y) = (x + y) \cdot (x - y)$




- $a = x + y, \quad b = x - y, \quad f = a \cdot b$
- **With** $x = 3, y = 1$: $\quad a = 4, b = 2, f = 8$

# Forward Pass: Values Flow Left $\rightarrow$ Right




We stored all intermediate values: $a = 4$, $b = 2$.

**These are needed for the backward pass.**

# Local Gradients

Each node knows how its output depends on its inputs.

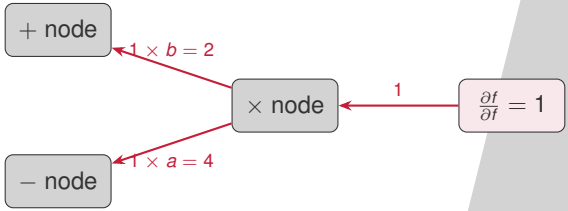
| Operation | Local Gradient | Example |
|---|---|---|
| $a + b$ | $\frac{\partial}{\partial a} = 1,\ \frac{\partial}{\partial b} = 1$ | Gradient passes through |
| $a - b$ | $\frac{\partial}{\partial a} = 1,\ \frac{\partial}{\partial b} = -1$ | Sign flips for $b$ |
| $a \cdot b$ | $\frac{\partial}{\partial a} = b,\ \frac{\partial}{\partial b} = a$ | **Swap the inputs!** |
| $\sigma(a)$ | $\sigma(a)(1 - \sigma(a))$ | Max = 0.25 |
| ReLU($a$) | $\mathbf{1}_{a>0}$ | 1 or 0 (a gate) |

**These are the building blocks.** Know these, and you can backprop through anything.

# Backward Pass: Gradients Flow Right $\rightarrow$ Left

At each node:

$$\text{downstream gradient} = \text{upstream gradient} \times \text{local gradient}$$




**Seed:** $\frac{\partial f}{\partial f} = 1$ **Then propagate backward node by node.**

# Full Backward Pass: $f(x, y) = (x + y)(x - y)$, $x = 3, y = 1$

**Through $\times$ node:**

$\frac{\partial f}{\partial a} = b = 2, \quad \frac{\partial f}{\partial b} = a = 4$

**Through $+$ node** (upstream = 2):

$\frac{\partial f}{\partial x}\Big|_{\text{add}} = 2 \times 1 = 2$

$\frac{\partial f}{\partial y}\Big|_{\text{add}} = 2 \times 1 = 2$

**Through $-$ node** (upstream = 4):

$\frac{\partial f}{\partial x}\Big|_{\text{sub}} = 4 \times 1 = 4$

$\frac{\partial f}{\partial y}\Big|_{\text{sub}} = 4 \times (-1) = -4$

**Sum paths:**

$\frac{\partial f}{\partial x} = 2 + 4 = \mathbf{6}$

$\frac{\partial f}{\partial y} = 2 + (-4) = \mathbf{-2}$

**Verify**

$f = x^2 - y^2$

$\frac{\partial f}{\partial x} = 2x = 6$ ✓

$\frac{\partial f}{\partial y} = -2y = -2$ ✓

# Key Insight: Locality

## Each node is independent

A node only needs:

1. Its **local gradient** (from its own operation)
2. The **upstream gradient** (from the node downstream)

**No node needs to know the full computation.**

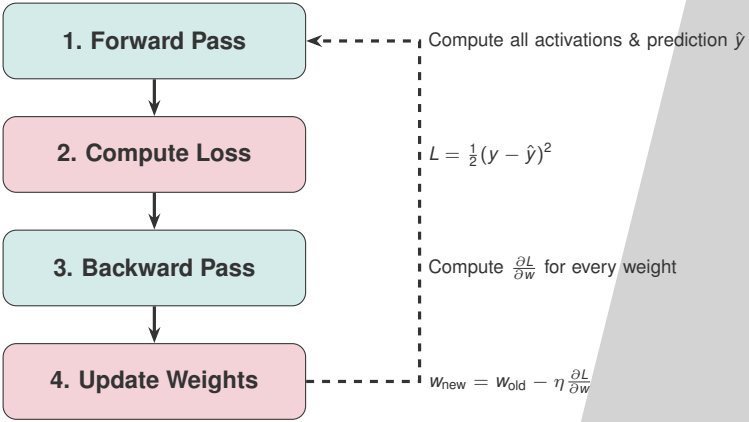
This is why backpropagation is:

- **Efficient** — scales to billions of parameters
- **Modular** — each operation defines its own backward
- **The basis of PyTorch** — each op records its gradient function; `.backward()` chains them

## Coming Up Next

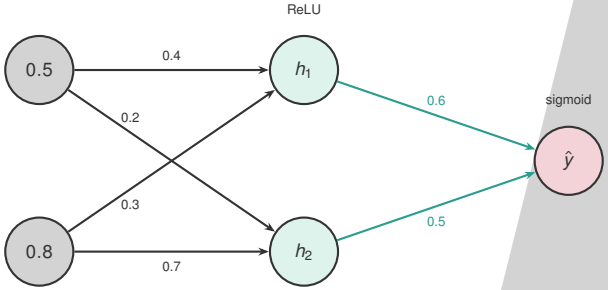
**Part 2:** Backpropagation on a Real Neural Network

# The Backpropagation Algorithm




**Forward: how wrong are you?** **Backward: who's responsible?** **Update: fix it.**

# Our Test Network (from Session 2)




$$W_1 = \begin{bmatrix} 0.4 & 0.3 \\ 0.2 & 0.7 \end{bmatrix}, \; b_1 = [0.1, 0.05], \quad W_2 = [0.6, 0.5], \; b_2 = -0.3$$

$$\text{Input } \mathbf{x} = [0.5, 0.8], \quad \text{Target } y = 1.0$$

# Step 1: Forward Pass (Recap from Session 2)

**Hidden layer:**

$z_1 = W_1\mathbf{x} + b_1 = [0.54, 0.71]$

$h = \text{ReLU}(z_1) = [0.54, 0.71]$

(both positive $\rightarrow$ ReLU passes through)

**Output layer:**

$z_2 = W_2 h + b_2 = 0.379$

$\hat{y} = \sigma(0.379) = 0.594$

The network predicts **0.594**.

Target is **1.0**. It's wrong.

We stored $z_1$, $h$, $z_2$, $\hat{y}$ — all needed for the backward pass.

# Step 2: Compute the Loss

$$L = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(1.0 - 0.594)^2 = \frac{1}{2}(0.406)^2 = \mathbf{0.082}$$

The error is 0.082. Not zero $\rightarrow$ the network needs to improve.

**The question backprop answers:**

Which weights are most responsible for this error, and how should each one change to reduce it?

# Step 3a: Backward Through the Output Layer

1. $\frac{\partial L}{\partial \hat{y}} = -(y - \hat{y}) = -0.406$
2. $\frac{\partial \hat{y}}{\partial z_2} = \sigma(z_2)(1 - \sigma(z_2)) = 0.594 \times 0.406 = 0.241$
3. $\frac{\partial L}{\partial z_2} = -0.406 \times 0.241 = -0.098$
4. $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot h^T = -0.098 \times [0.54, 0.71] = [-0.053, -0.069]$
5. $\frac{\partial L}{\partial b_2} = -0.098$

**Negative gradients** $\rightarrow$ weights should **increase** to reduce the loss.

# Step 3b: Backward Through the Hidden Layer

1. $\dfrac{\partial L}{\partial h} = \dfrac{\partial L}{\partial z_2} \cdot W_2^T = -0.098 \times [0.6, 0.5] = [-0.059, -0.049]$
2. $\dfrac{\partial h}{\partial z_1} = \text{ReLU}'(z_1) = [1, 1]$ (both $z_1$ positive)
3. $\dfrac{\partial L}{\partial z_1} = [-0.059, -0.049] \odot [1, 1] = [-0.059, -0.049]$
4. $\dfrac{\partial L}{\partial W_1} = \dfrac{\partial L}{\partial z_1} \cdot \mathbf{x}^T = \begin{bmatrix} -0.029 & -0.047 \\ -0.024 & -0.039 \end{bmatrix}$
5. $\dfrac{\partial L}{\partial b_1} = [-0.059, -0.049]$

**Every weight now has a gradient.** The backward pass is complete.

# Step 4: Update Weights

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{\partial L}{\partial w}, \qquad \eta = 0.1$$

| Parameter | Before | After |
|---|---|---|
| $W_2$ | $[0.600, 0.500]$ | $[0.605, 0.507]$ |
| $b_2$ | $-0.300$ | $-0.290$ |
| $W_1[0, :]$ | $[0.400, 0.300]$ | $[0.403, 0.305]$ |
| $W_1[1, :]$ | $[0.200, 0.700]$ | $[0.202, 0.704]$ |
| $b_1$ | $[0.100, 0.050]$ | $[0.106, 0.055]$ |

All weights increased slightly (gradients were negative).
New prediction: $\hat{y} \approx 0.601$ — closer to target 1.0.

**Repeat thousands of times $\rightarrow$ the network converges.**

# The Big Picture: One Training Step

**Forward pass** $\rightarrow$ **Loss** $\rightarrow$ **Backward pass** $\rightarrow$ **Update** $\rightarrow$ *repeat*

- One forward + backward on one example = one **iteration**
- Full pass through all training data = one **epoch**
- Typical training: 50–100 epochs

**The elegance**

**Same algorithm** for every architecture: MLPs, CNNs, RNNs, Transformers.
Only the graph structure changes.

# Quiz 2

**Q1** Backward Order

Which weights get their gradients **first** during the backward pass — $W_1$ or $W_2$?

Why?

**Q2** ReLU Gate

A hidden neuron has $z_1 = -0.5$.

What happens to the gradient flowing **through** it?

(ReLU activation)

**Q3** Reverse Engineer

After gradient descent ($\eta = 0.1$), a weight changes:

$$0.6 \rightarrow 0.605$$

What was $\frac{\partial L}{\partial w}$?

*Take 2 minutes.*

# Quiz 2 — Answers

## A1

**$W_2$** first.

Backward pass starts at loss and moves toward input. $W_2$ is closer to the output, so its gradient is computed first.

## A2

Gradient is **killed** (= 0).

ReLU grad $= 0$ for $z < 0$.
All upstream weights receive **zero gradient**.

$\Rightarrow$ **Dead neuron problem.**

## A3

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

$$0.605 = 0.6 - 0.1 \cdot \frac{\partial L}{\partial w}$$

$$\frac{\partial L}{\partial w} = \mathbf{-0.05}$$

# Gradient Flow Through Deep Networks

The chain rule multiplies local gradients:

$$\frac{\partial L}{\partial w} = \prod_{i=1}^{n} g_i \quad \text{where } g_i \text{ is a local gradient}$$

**$g_i < 1$ at each layer**

$0.5^{50} \approx 10^{-15}$

Gradient **vanishes**.
Early layers stop learning.

**$g_i > 1$ at each layer**

$2^{50} \approx 10^{15}$

Gradient **explodes**.
Training diverges.

**The fundamental challenge of deep learning: keeping gradients healthy.**

# The Vanishing Gradient Problem

**Sigmoid**: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. Maximum value: **0.25**.

| Layers | Gradient at layer 1 |
|---|---|
| 5 layers | $0.25^5 \approx 0.001$ |
| 10 layers | $0.25^{10} \approx 10^{-6}$ |
| 20 layers | $0.25^{20} \approx 10^{-12}$ |

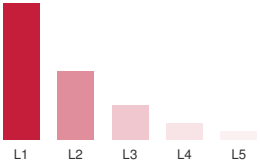
Early layers receive essentially **zero gradient**.
They cannot learn. They stay at their random initialization.

This is why deep sigmoid networks could not be trained.
The chain rule guarantees the signal fades.

# Why ReLU Fixes the Vanishing Gradient

**Sigmoid**: gradient $\leq 0.25$

Through $n$ layers: $0.25^n \to 0$




Signal fades to nothing.

**ReLU**: gradient $= 1$ (positive)

Through $n$ layers: $1^n = 1$


Signal stays at full strength.

**ReLU was the breakthrough that enabled deep learning.**

(Dead neurons: gradient = 0 for negative inputs. Manageable in practice.)

# Exploding Gradients (Preview)

## The Problem

Gradients grow exponentially through layers.

**Symptom:** Loss jumps to $\infty$ or NaN. Weights oscillate wildly.

## Same root cause

Chain rule multiplies many local gradients.

$g_i < 1$ each layer $\Rightarrow$ **vanishing**
$g_i > 1$ each layer $\Rightarrow$ **exploding**

## Fixes (Week 3)

1. **Gradient clipping**
   Cap gradient norm at a threshold
2. **Careful initialization**
   Xavier / He initialization
3. **Batch normalization**
   Normalize activations per layer

# PyTorch Autograd — How It Works

1. Create tensors with `requires_grad=True`
2. Perform operations $\rightarrow$ PyTorch builds a computational graph
3. Call `loss.backward()` $\rightarrow$ traverse graph in reverse
4. Gradients stored in `.grad`

**The key insight**

**You define the forward pass. PyTorch gives you the backward pass for free.**

Even `if`-statements and `for`-loops are handled correctly.
This is why PyTorch can train *any* architecture.

# Autograd in Action

**Simple example:**

```
x = torch.tensor(3.0, requires_grad=True)
y = x**2 + 2*x + 1
y.backward()
print(x.grad)   # 8.0   (dy/dx = 2x + 2 = 8)
```

**Training loop:**

```
output = model(input)
loss = criterion(output, target)
loss.backward()     # Compute all gradients
optimizer.step()     # Update all weights
optimizer.zero_grad() # Reset for next step
```

**Four lines.** Everything we learned today happens inside `loss.backward()`.

# Recap — 5 Key Takeaways

1. **The chain rule** decomposes gradients into products of local gradients.
2. **Computational graphs** make derivative computation systematic and visual.
3. **Backprop = forward + loss + backward + update.** The training loop for all neural networks.
4. **Vanishing gradients** limit depth (sigmoid $\leq 0.25$). **ReLU** keeps gradients alive (gradient = 1).
5. **Autograd** automates the backward pass. `loss.backward()` is backprop in one line.

# Final Quiz

**Q1** The Four Steps

What are the four steps of the backpropagation algorithm?

**Q2** Local Gradient

In a computational graph, $z = x \cdot y$.

What is $\frac{\partial z}{\partial x}$?

**Q3** Vanishing Gradients

Why does sigmoid cause vanishing gradients? What is its maximum derivative?

**Q4** PyTorch Autograd

What does `loss.backward()` compute?

Where are the results stored?

**Q5** Scale of Backprop

An MLP has **109,386 parameters**.

After `loss.backward()`, how many gradient values exist?

*Take 3 minutes.*

# Final Quiz — Answers

## A1

Forward pass $\rightarrow$ loss $\rightarrow$ backward pass $\rightarrow$ update weights.

## A2

$\frac{\partial z}{\partial x} = y$ Gradient w.r.t. one input = the **other input's value**.

## A3

$\sigma'(z) = \sigma(z)(1-\sigma(z))$. Max = **0.25** at $z = 0$. Through $n$ layers: $0.25^n \rightarrow 0$.

## A4

Computes $\frac{\partial L}{\partial w}$ for every parameter with `requires_grad=True`. Stored in `tensor.grad`.

## A5

**109,386** gradients — one per parameter. Scales linearly with model size.

# Thank You!

Questions?

**Priyansh Saxena**

All lecture materials available at:

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