上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

# Lecture 15: Backpropagation

Tao Huang

John Hopcroft Center, School of Computer Science, Shanghai Jiao Tong University

https://taohuang.info/cs3317

https://oc.sjtu.edu.cn/courses/89538

AI tools assisted in generating some figures in these slides. All such content has been reviewed, and the instructor is responsible for its accuracy.

# We Can Build MLPs. How Do We Train Them?

Last lecture:

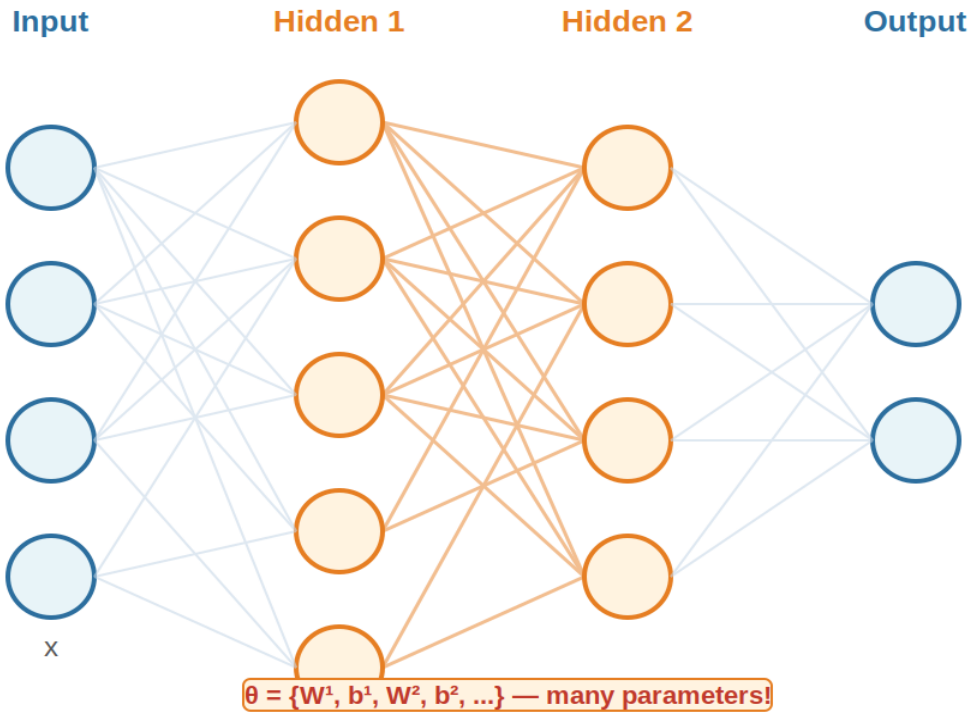
- perceptron, MLP, activation

Now the real problem:

- $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$

Question:

- How do we compute ∇θ L efficiently?



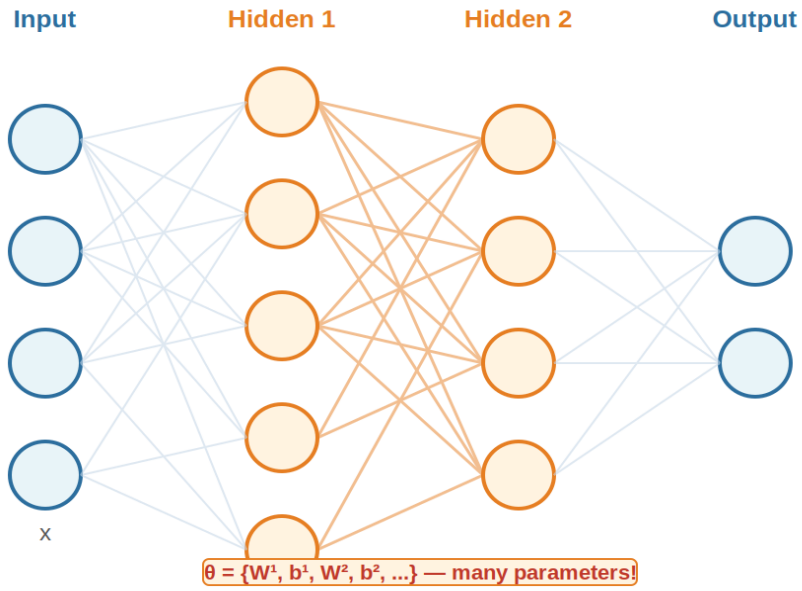
# The Gradient Computation Problem

A network is a nested function: $f(x) = f_L(f_{L-1}(\cdots f_1(x)))$

Loss depends on all parameters: $\mathcal{L} = \mathcal{L}(f(x;\theta), y)$

We need:

- exact gradients, shared computation, cost near one forward pass



# Objectives

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

- Explain what is propagated during backprop and why reverse-mode AD matches the neural network training setup
- Apply the chain rule systematically on a computational graph to derive gradients for any layer
- Distinguish backpropagation (gradient computation) from optimization (parameter update), and identify what can go wrong even when gradients are correct

# 1. Computational Graphs

# A Neural Network Is a Computational Graph

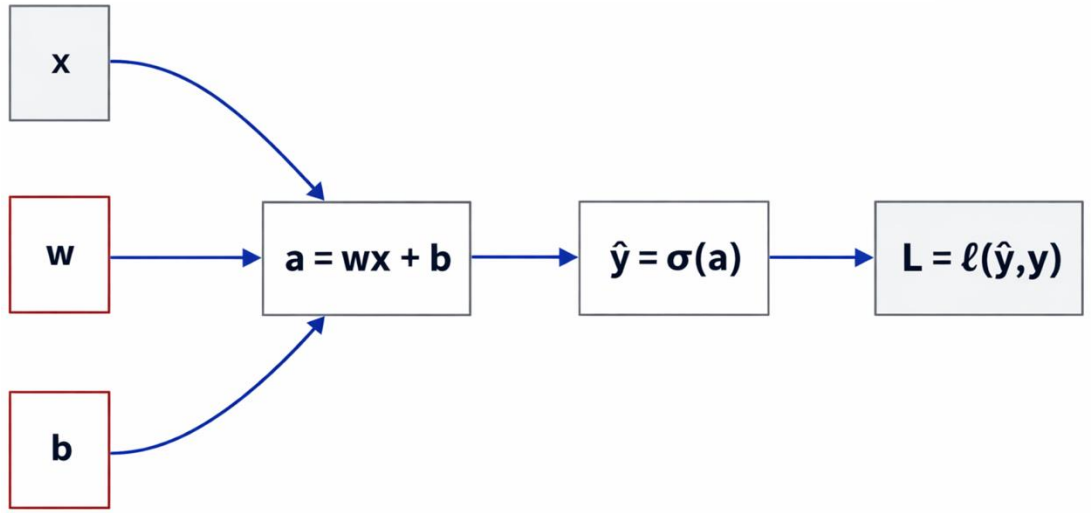
Nodes: variables, intermediate results, parameters

Edges: dependencies

Two passes:

- Forward pass computes values
- Backward pass computes gradients

Example: $a = wx + b, \qquad \hat{y} = \sigma(a), \qquad \mathcal{L} = \ell(\hat{y}, y)$



# Two Passes, Two Different Objects

**Forward pass computes:**

- activations
- prediction
- loss

**Forward = values**

**Backward pass computes:**

- gradients of loss
- wrt intermediate variables
- wrt parameters

**Backward = sensitivities**

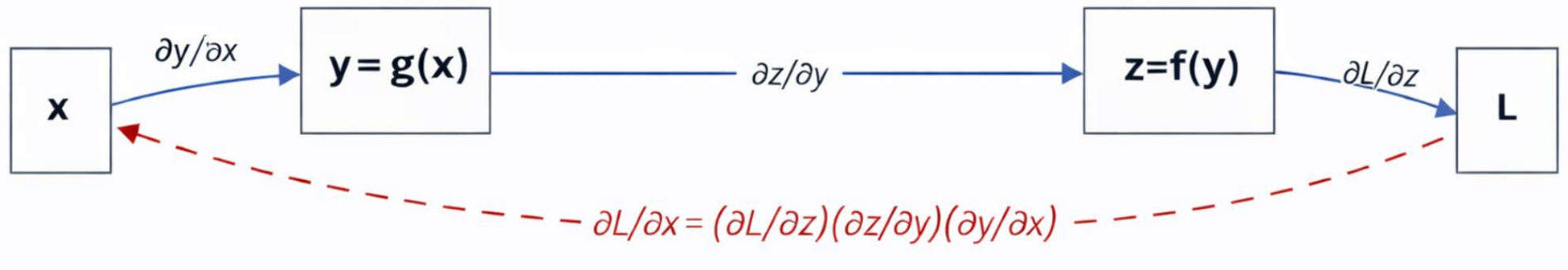
# Backprop = Chain Rule at Scale

If $z = f(y), \qquad y = g(x)$

Then $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$ → $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial u}\frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$

For a chain:

- multiply local derivatives
- accumulate effects on loss



# What Is Actually Propagated?

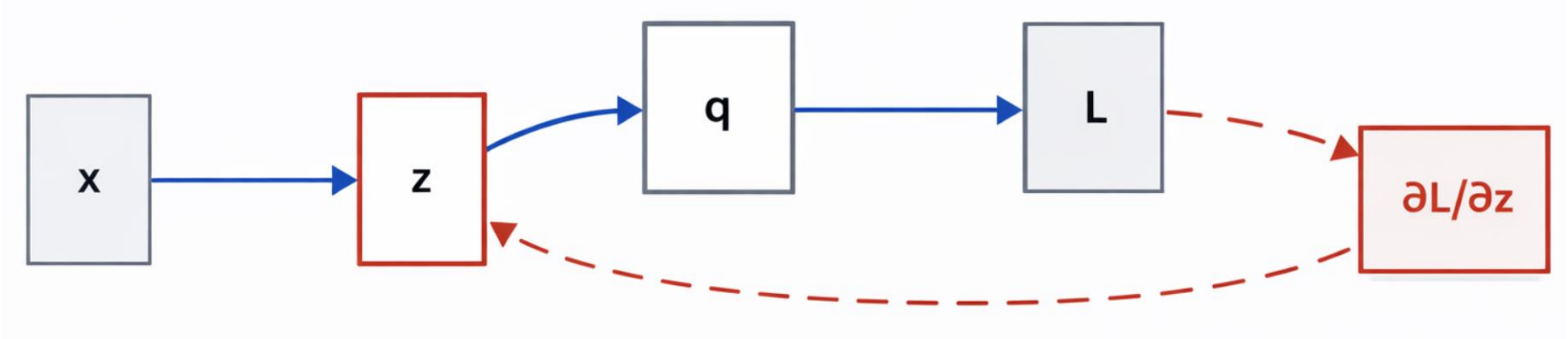
The core object: $\frac{\partial \mathcal{L}}{\partial z}$

Interpretation:

- If z changes slightly, how much does the loss change?

Backprop propagates:

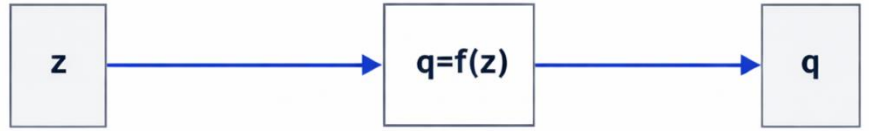
- loss sensitivity



# One Node: Upstream × Local

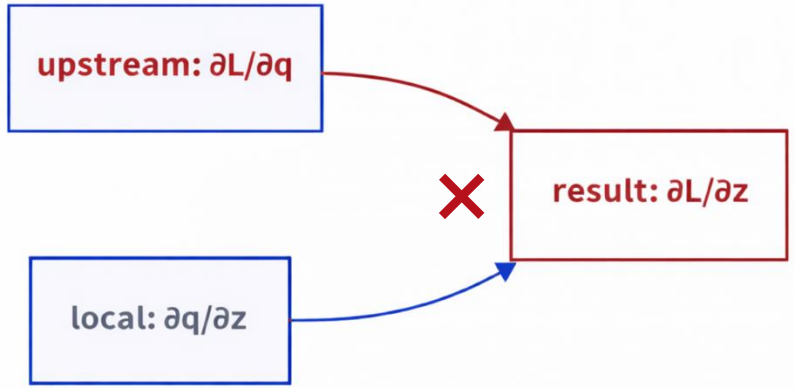
If $q = f(z)$

Then $\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial z}$




Rule:

upstream gradient × local derivative



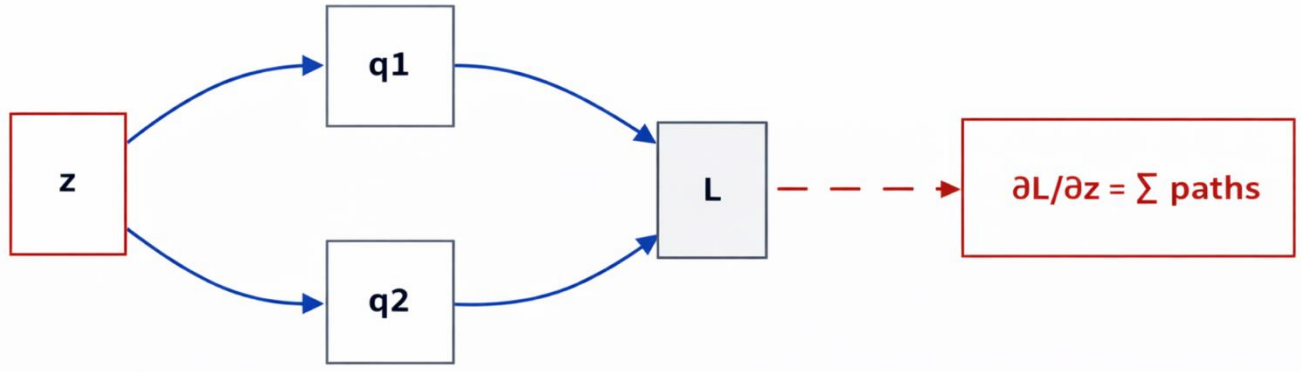
# When One Variable Affects Multiple Paths

If z influences loss through multiple children:

$$z \to q_1, \quad z \to q_2$$

Then:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial q_1}\frac{\partial q_1}{\partial z} + \frac{\partial \mathcal{L}}{\partial q_2}\frac{\partial q_2}{\partial z}$$




Rule:

- along a path: multiply
- across paths: add

# 2. Local Rules for NN Building Blocks

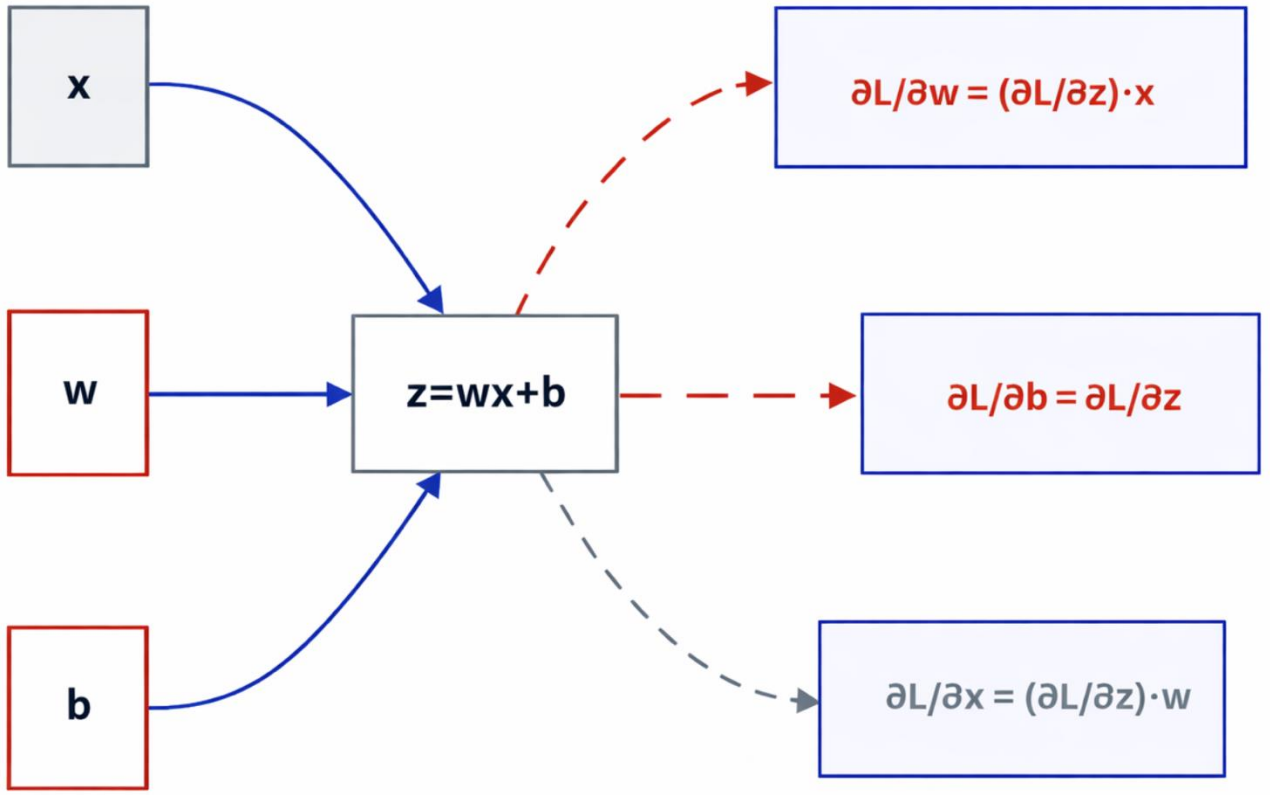
# Affine Node

For one neuron:

$$z = wx + b$$

Gradients:

- $\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z} x$
- $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z} w$
- $\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}$



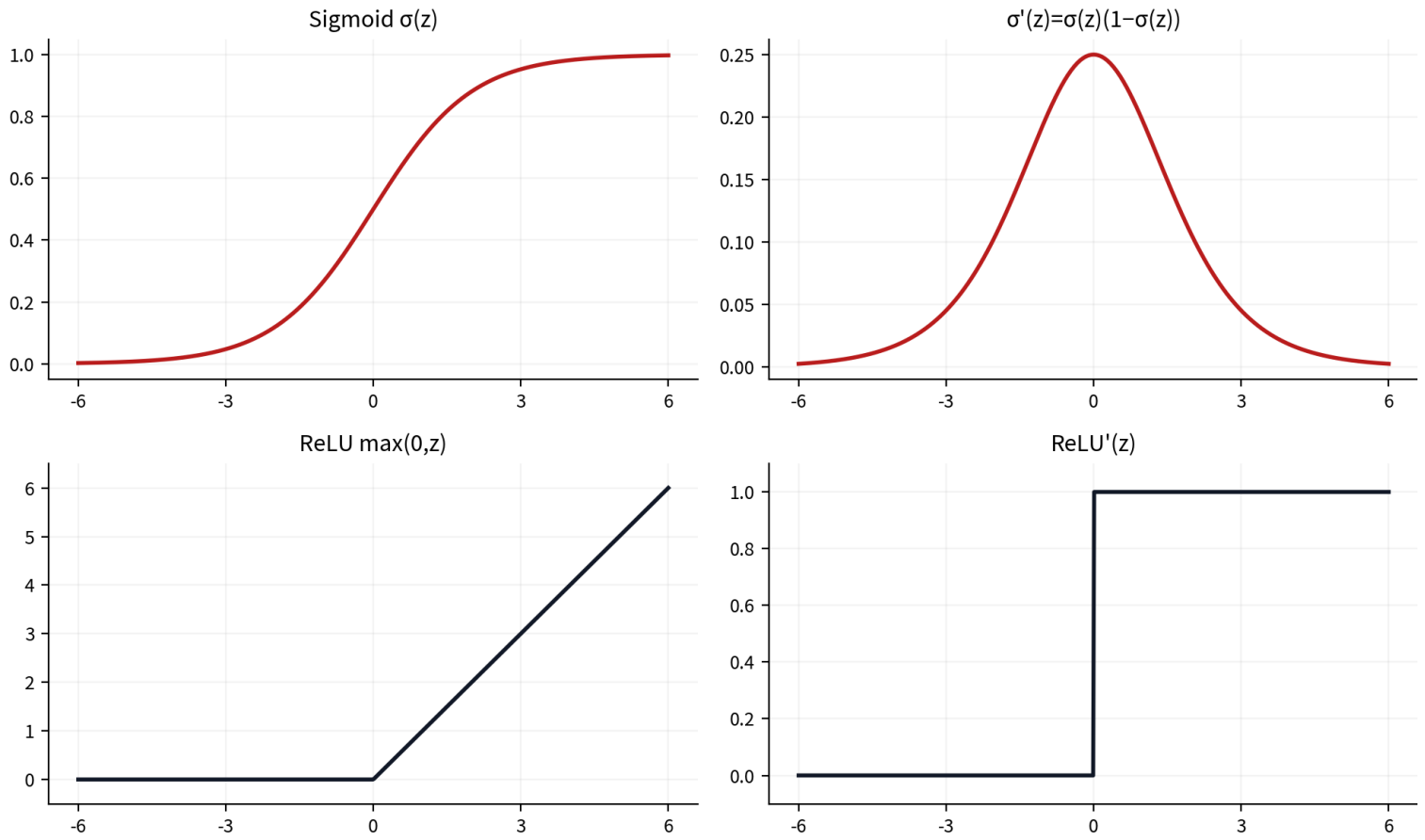
# Activation Node

For $a = \phi(z)$:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a}\phi'(z)$$

Examples:

- $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- $\text{ReLU}'(z) = \begin{cases} 1, & z > 0 \\ 0, & z < 0 \end{cases}$



# Loss Node

Example:

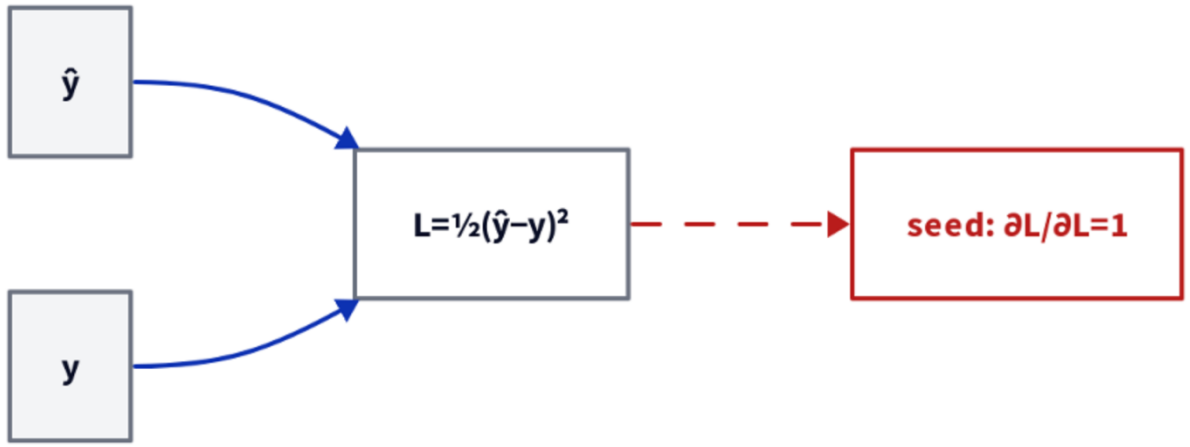
$$\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$$

then

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y$$

Backprop always begins with:

$$\frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$$



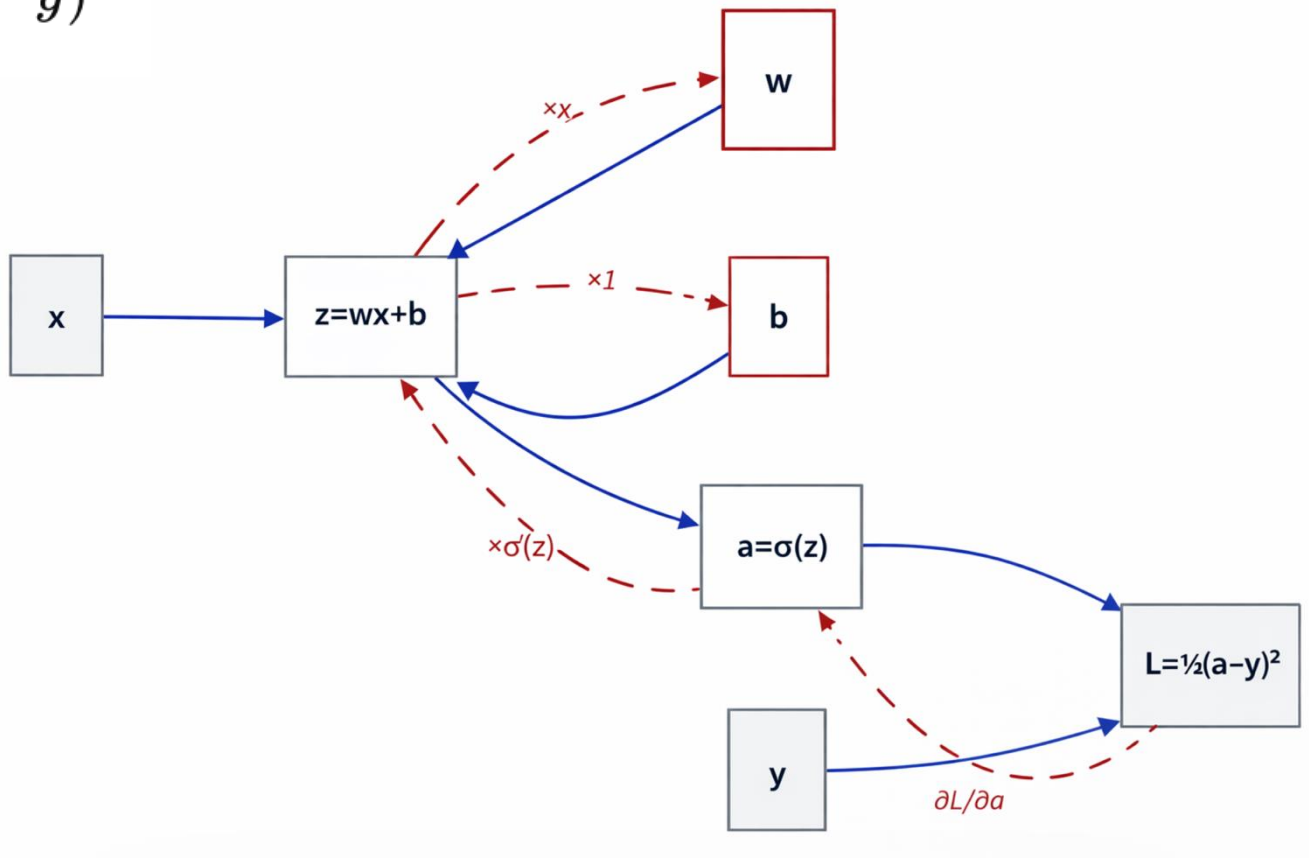
# One Neuron, End to End

Model:

$$z = wx + b, \quad \hat{y} = \sigma(z), \quad \mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$$

Derive backwards:

- $\frac{\partial \mathcal{L}}{\partial \hat{y}}$
- $\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z}$
- $\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z} x$

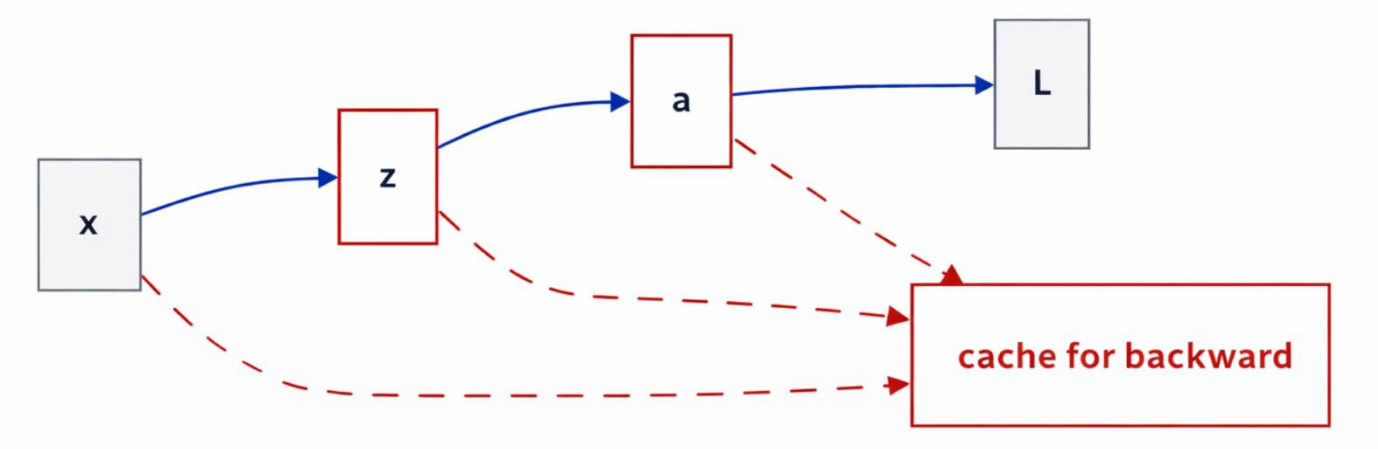


# What Must Be Cached?

Backward needs forward intermediates.

Examples:

- input x
- pre-activation z
- activation a
- ReLU mask




Question:

- Which quantities can be recomputed? (Extension of this week: **Gradient checkpointing** in large model training)
- Which should be stored?

# 3. Backprop Through a Layer

# Fully Connected Layer: Vector Form

Forward:

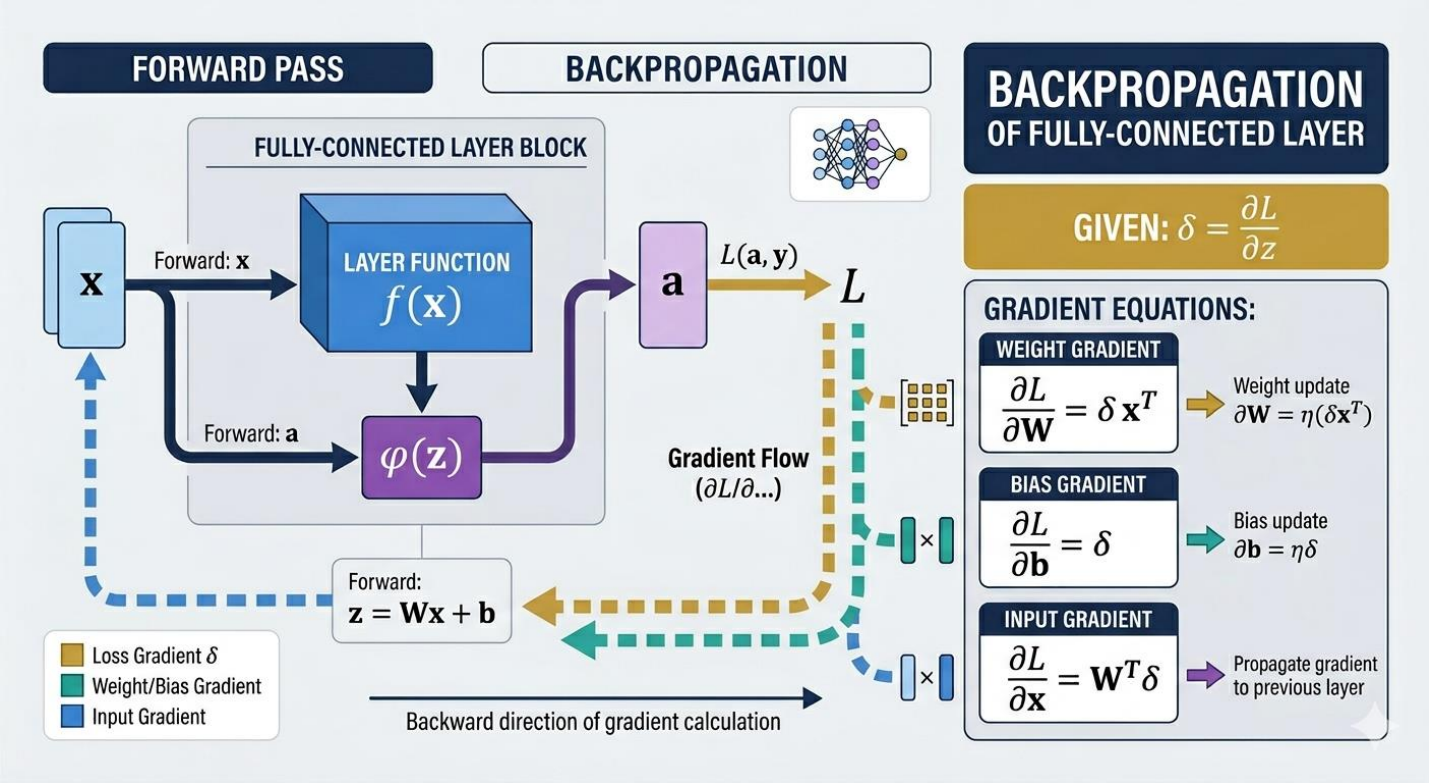
$$z = Wx + b, \qquad a = \phi(z)$$

Given $\delta = \dfrac{\partial \mathcal{L}}{\partial z}$ , then:

$$\frac{\partial \mathcal{L}}{\partial b} = \delta$$

$$\frac{\partial \mathcal{L}}{\partial W} = \delta x^T$$

$$\frac{\partial \mathcal{L}}{\partial x} = W^T \delta$$



# Backprop in an MLP

**Forward:**

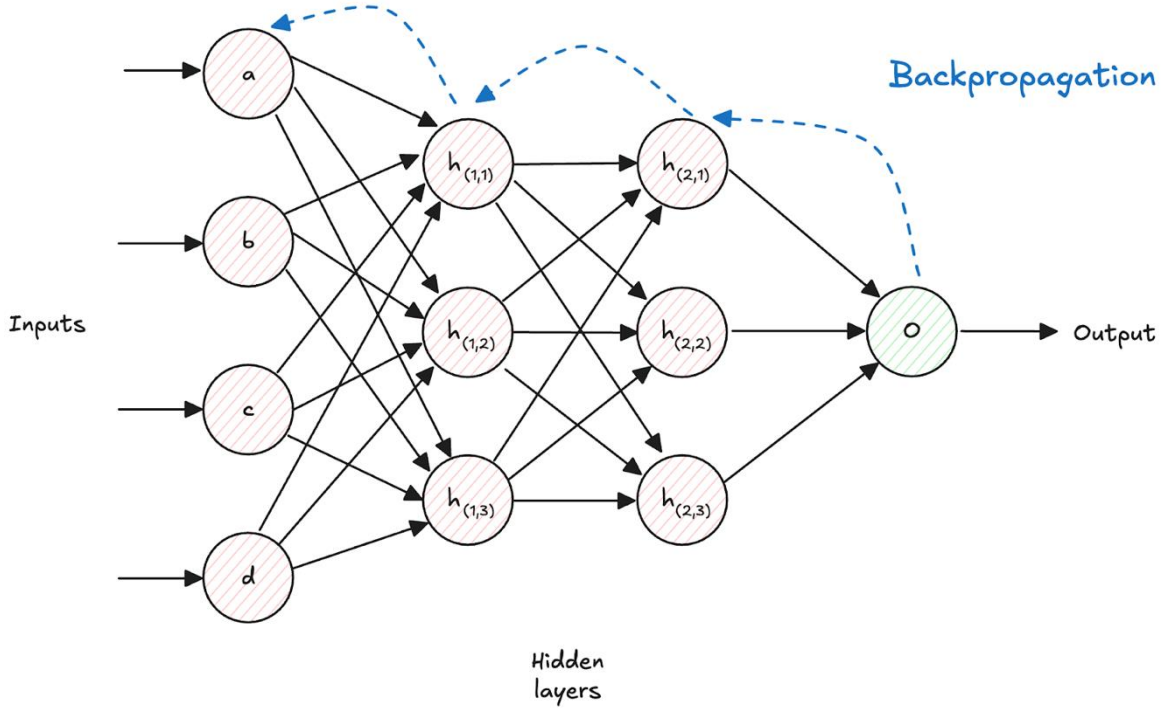
- for each layer, compute

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \phi(z^{(l)})$$

**Backward:**

1. output-layer gradient
2. hidden-layer deltas
3. parameter grads $\frac{\partial \mathcal{L}}{\partial W^{(l)}}, \frac{\partial \mathcal{L}}{\partial b^{(l)}}$



# Why Backprop Is Efficient

Naive idea: compute one derivative per parameter

Backprop:

- reuses shared subcomputations
- dynamic programming on the graph

Rule of thumb:

- backward cost ≈ forward cost

# Backprop Is Not Gradient Descent

Backprop:

- computes gradients

Optimizer:

- uses gradients to update parameters

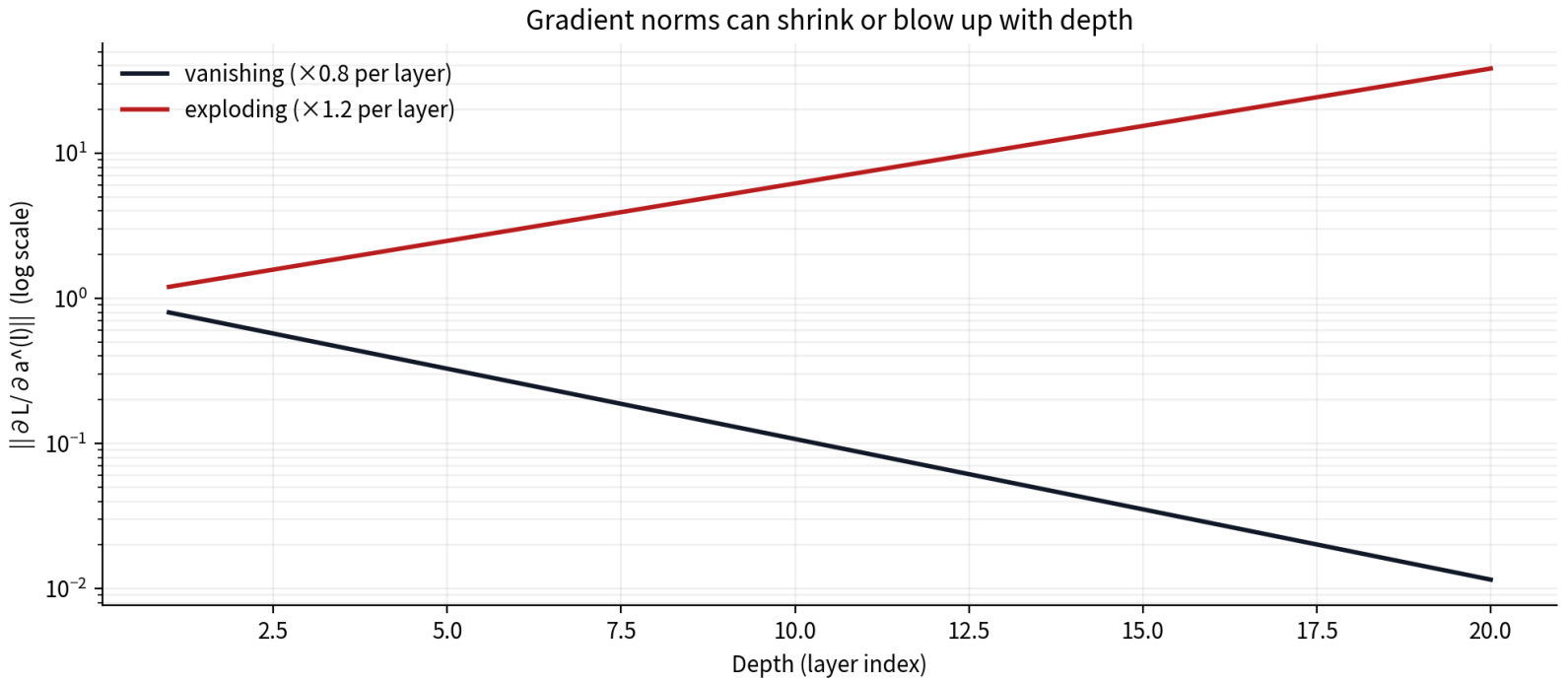
**Training loop pipeline:**

# Correct Gradients, Bad Training

Even with correct backprop, training may fail because of:

- vanishing / exploding gradients
- poor initialization, saturation, unstable optimization

These are training dynamics, not derivation errors (topic for next lecture).



# Summary

**Three key points:**

1. Backprop is reverse-mode differentiation on a computational graph.

2. Core rule: upstream $\times$ local, with addition across branches.

3. Backprop makes neural training feasible by computing all parameter gradients efficiently in one reverse pass.

# Coursework 3

- Implement backpropagation

### What You Need to Implement

| File | Function / Method | Description |
|---|---|---|
| layers.py | Linear.forward(X) | $\mathbf{z} = \mathbf{XW} + \mathbf{b}$, cache $\mathbf{X}$ |
| layers.py | Linear.backward(grad_output) | Compute grad_W, grad_b, return grad_input |
| layers.py | ReLU.forward(X) | $\max(0, \mathbf{X})$, cache input |
| layers.py | ReLU.backward(grad_output) | Zero gradient where input $\leq 0$ |
| layers.py | Sigmoid.forward(X) | $\sigma(\mathbf{X}) = 1/(1 + e^{-\mathbf{X}})$ |
| layers.py | Sigmoid.backward(grad_output) | $\nabla = \text{grad} \cdot \sigma \cdot (1-\sigma)$ |
| model.py | MLP.__init__(...) | Build list of layers |
| model.py | MLP.forward(X) | Sequential forward |
| model.py | MLP.backward(grad_output) | Reverse-order backward |
| losses.py | CrossEntropyLoss.forward(logits, labels) | Softmax + cross-entropy (reuse CW2) |
| losses.py | CrossEntropyLoss.backward() | Return cached gradient |
| optimizer.py | SGD.__init__(params, lr, momentum) | Initialize velocities |
| optimizer.py | SGD.step() | Update params with momentum |
| optimizer.py | SGD.zero_grad() | Reset all gradients to zero |