Machine Learning and Optimization

Lecture 14: Backpropagation and Reverse-Mode Automatic Differentiation

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. **Explain** why training neural networks requires efficiently differentiating compositions of functions
2. **State** the multivariate chain rule in Jacobian form, interpret it as composing linearizations, and specialize to the scalar-output (gradient) case
3. **Derive** the reverse-mode AD backward rule on a general DAG, explain why it costs $O(T_{\text{fwd}})$, and contrast with forward mode
4. **Execute** the backpropagation algorithm for an MLP layer-by-layer, with correct dimensions at each step

Lecture Roadmap

Motivation and Running Example

Computation Graphs

The Chain Rule and Jacobians

Reverse-Mode AD

Backpropagation for MLPs

What Gets Stored

Autograd and Wrap-Up

Motivation: Training Requires Gradients

We train a neural network by minimizing the empirical risk:

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell\big(f_\theta(x_i),\, y_i\big)$$

Every gradient method (SGD, Adam, ...) needs $\nabla_\theta f(\theta)$.

But f_θ is a **composition of many parameterized functions**:

$$x \xrightarrow{W^1, b^1} \xrightarrow{\sigma} \xrightarrow{W^2, b^2} \cdots \xrightarrow{\text{loss}} f$$

Each layer has its own parameters. The gradient $\nabla_\theta f$ must account for *every* parameter in the chain.

Key question: How do we efficiently compute the gradient of a large composition?

End Goal

One algorithm to differentiate them all.

Today we develop a technique that computes the gradient of the empirical risk for *any* neural network architecture in $O(T_{\text{fwd}})$ time, where T_{fwd} is the cost of a single forward evaluation of the network.

The technique is far more general than neural networks: it applies to *any differentiable computation* that can be expressed as a directed acyclic graph (DAG).

The payoff: you specify the computation as a program (e.g., in PyTorch), and gradient computation is *automatic* — no manual derivation required. You design a new architecture, and gradients come for free.

Running Example: Multiclass Logistic Regression

Consider k-class multiclass classification with d-dimensional inputs.

The per-sample loss is:

$$f(W, b; x, y) = -\log p_{W,b}(y \mid x) \tag{1}$$

$$= -\log \frac{e^{(Wx+b)_y}}{\sum_{j=1}^{k} e^{(Wx+b)_j}} \tag{2}$$

$$= -o_y + \log \sum_{j=1}^{k} e^{o_j}, \qquad o = Wx + b \tag{3}$$

For simplicity, we study a single data point. The optimization problem is:

$$\operatorname*{argmin}_{W,b} \; f(W, b; x, y) = \operatorname*{argmin}_{W,b} \left[-o_y + \log \textstyle\sum_{j=1}^{k} e^{o_j} \right]$$

Running Example: Setup Summary

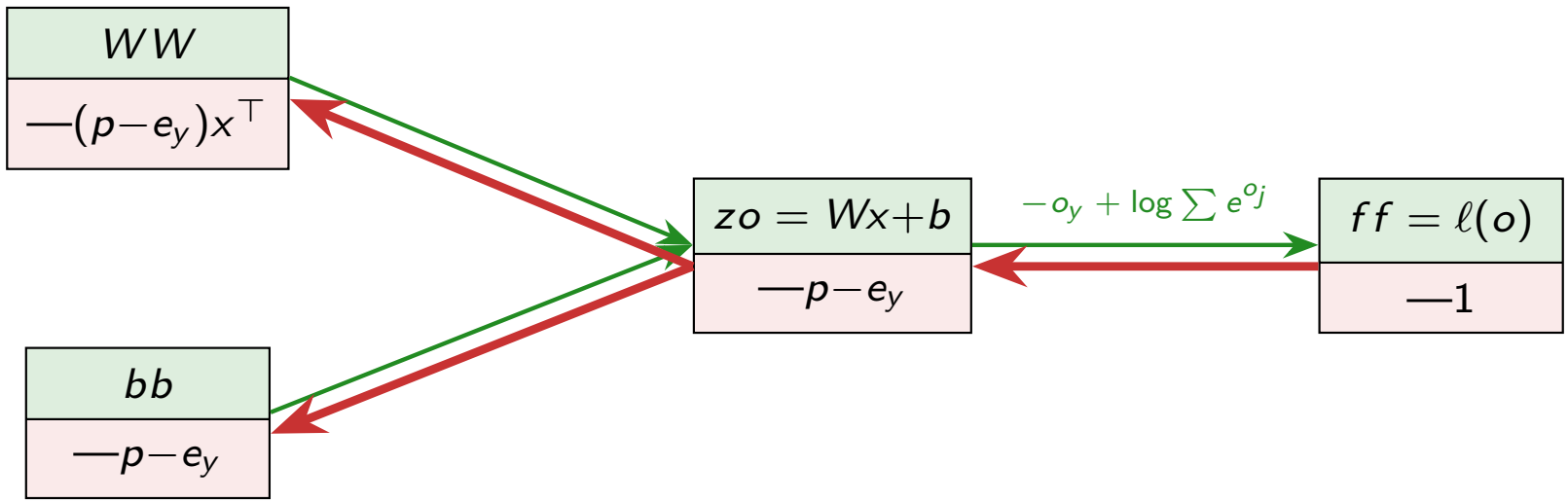
$$
\begin{aligned}
\textbf{Input:} \quad & x \in \mathbb{R}^d \\
\textbf{Label:} \quad & y \in \{1, \dots, k\} \\
\textbf{Parameters:} \quad & W \in \mathbb{R}^{k \times d}, \;\; b \in \mathbb{R}^k \\
\textbf{Logits:} \quad & o = Wx + b \in \mathbb{R}^k \\
\textbf{Softmax:} \quad & p_i = e^{o_i} / \textstyle\sum_{j=1}^{k} e^{o_j} \\
\textbf{Loss:} \quad & \ell(o) = -o_y + \log \textstyle\sum_{j=1}^{k} e^{o_j} \\
\textbf{Objective:} \quad & f = \ell \circ o \quad \text{(composed scalar)}
\end{aligned}
$$

Goal: Compute $\nabla_W f$ and $\nabla_b f$ so we can run gradient descent.

Running Example: Computation Graph

The single-point problem defines the following computational DAG.

The green (top) portion of each node shows intermediate values computed in the forward pass. The pink (bottom) portion shows $\nabla_v f$, the gradient of f w.r.t. that variable. We derive the claimed gradients in the following slides.



Remark. The approach we develop for this network will generalize to computing gradients in any computational DAG.

High-Level Overview: Gradients via the Layer Output

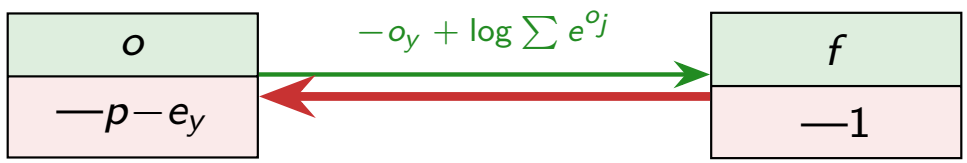
We want to compute $\nabla_W f$ and $\nabla_b f$. The composed objective $f = \ell \circ o$ depends on W and b through the intermediate variable o. Our strategy:

1. Compute how $\ell(o)$ depends on o (i.e., $\nabla_o f$).
2. Compute how o depends on W and b (local derivatives).
3. Combine via the chain rule to get $\nabla_W f$ and $\nabla_b f$.

Step 1: gradient of $\ell(o)$ w.r.t. logits. Define $p_i = e^{o_i} / \sum_j e^{o_j}$. Since $\ell(o) = -o_y + \log \sum_j e^{o_j}$:

$$\frac{\partial \ell(o)}{\partial o_i} = -\delta_{iy} + \frac{\partial}{\partial o_i} \log \sum_j e^{o_j} = -\delta_{iy} + \frac{e^{o_i}}{\sum_j e^{o_j}} = p_i - \delta_{iy}$$

In vector form: $\nabla_o f(o) = p - e_y$, where e_y is the y-th standard basis vector.



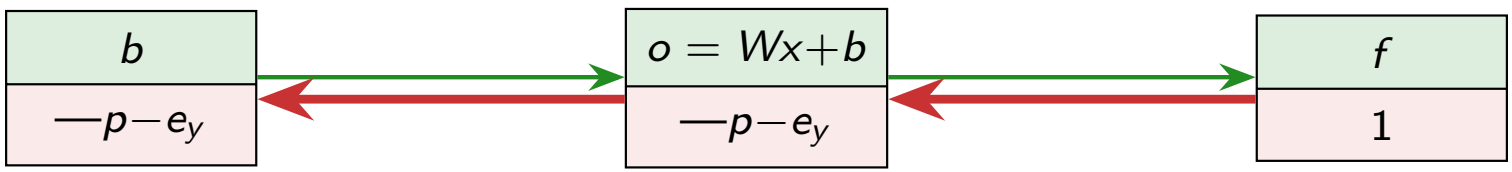
Gradient of f w.r.t. b

Since $o_j = \sum_p W_{jp} x_p + b_j$, we have $\partial o_j / \partial b_i = \delta_{ji}$.

Applying the law of total differentiation:

$$\frac{\partial f}{\partial b_i} = \sum_{j=1}^{k} \underbrace{\boxed{\frac{\partial f}{\partial o_j}}}_{\text{upstream gradient}} \underbrace{\boxed{\frac{\partial o_j}{\partial b_i}}}_{\text{local derivative}} = \sum_{j=1}^{k} \frac{\partial f}{\partial o_j} \delta_{ji} = \frac{\partial f}{\partial o_i}$$

Therefore: $\nabla_b f = \nabla_o f = p - e_y$.



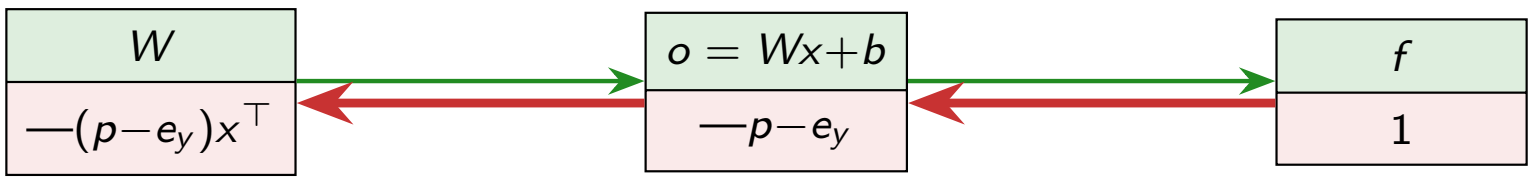
Gradient of f w.r.t. W

Since $o_m = \sum_p W_{mp} x_p + b_m$, we have $\partial o_m / \partial W_{ij} = \delta_{mi}\, x_j$.

Applying the law of total differentiation:

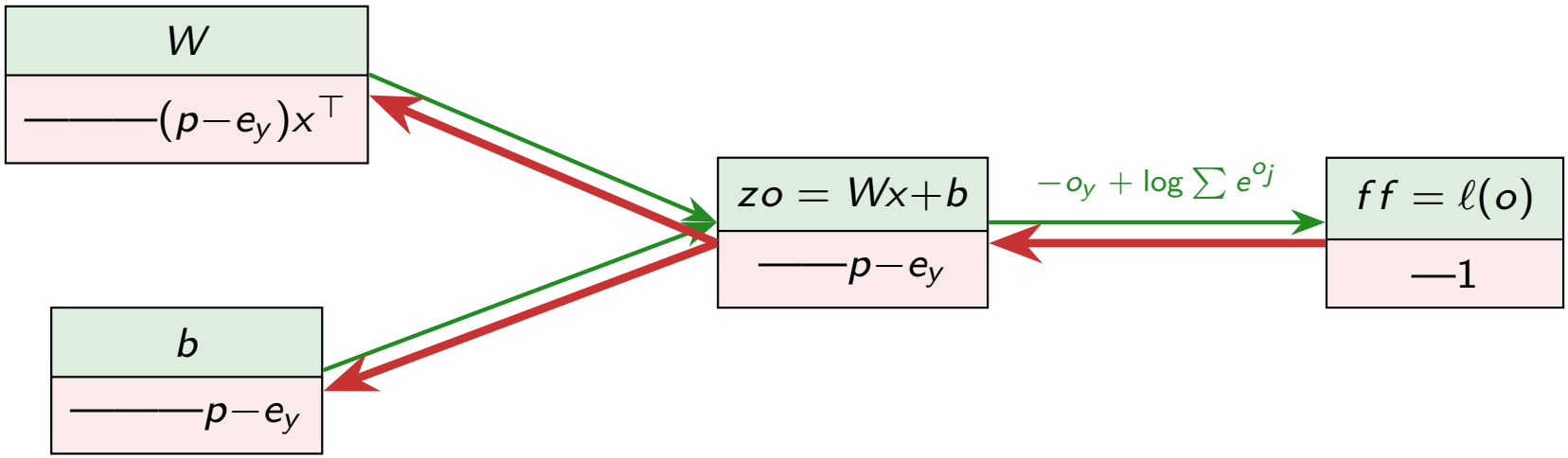
$$\frac{\partial f}{\partial W_{ij}} = \sum_{m=1}^{k} \underbrace{\boxed{\frac{\partial f}{\partial o_m}}}_{\text{upstream gradient}} \underbrace{\boxed{\frac{\partial o_m}{\partial W_{ij}}}}_{\text{local derivative}} = \sum_{m=1}^{k} \frac{\partial f}{\partial o_m}\, \delta_{mi}\, x_j = \frac{\partial f}{\partial o_i}\, x_j$$

Therefore: $\nabla_W f = (\nabla_o f)\, x^\top = (p - e_y)\, x^\top$.



Observation: We computed $\nabla_o f$ first, then reused it for both parameter gradients. This is the key pattern we will generalize.

Running Example: The Backward Pass as a DAG



This procedure — computing $\nabla_o f(o)$ first, then propagating it backward to obtain $\nabla_W f$ and $\nabla_b f$ — is called **reverse-mode automatic differentiation** (backpropagation when applied to neural networks). We will generalize this pattern to arbitrary DAGs.

The General Principle: Computation DAGs

Any **finite evaluation** of a differentiable program defines a DAG:

- **Nodes** = variables (intermediate quantities)
- **Edges** = direct dependencies ($u \to v$ means v depends on u)
- **Sources** = inputs and parameters
- **Sink** = loss (scalar output)

A **directed acyclic graph (DAG)** is a directed graph with no directed cycles. Every DAG has ≥ 1 source and ≥ 1 sink.

A **topological ordering** is a linear ordering of vertices such that $u \to v \implies u$ before v. Every DAG admits at least one topological ordering.

- **Forward pass** = topological order (sources $\to$ sink). Each node's inputs are already computed.
- **Backward pass** = reverse topological order (sink $\to$ sources). Each node's children are already processed.

Programs with loops are handled by *unrolling* each iteration into separate nodes, yielding a (potentially large) DAG for the executed trace.

Running Example: How Does o Respond to x?

To generalize the backward pass beyond our specific running example, we need tools for computing gradients of compositions of vector-valued functions. We begin by looking at how vector-valued functions respond to perturbations of inputs.

Consider the linear map $o = Wx + b$, with $o \in \mathbb{R}^k$ and $x \in \mathbb{R}^d$.

Perturbing x by δx:

$$\begin{aligned} o(x + \delta x) &= W(x + \delta x) + b \\ &= Wx + b + W\,\delta x \\ &= o + W\,\delta x \end{aligned}$$

So the matrix W tells us how perturbations in the input (x) map to perturbations in the output (o).

W **is the Jacobian** of the map $x \mapsto o$:

$$J_o(x) = W \in \mathbb{R}^{k \times d}$$

Row i = how o_i depends on all inputs. Column j = how all outputs respond to x_j.

The Jacobian Matrix (General)

For any general nonlinear function $g : \mathbb{R}^p \to \mathbb{R}^m$, the **Jacobian matrix** at x is:

$$J_g(x) = \begin{bmatrix} \nabla g_1(x)^\top \\ \vdots \\ \nabla g_m(x)^\top \end{bmatrix} \in \mathbb{R}^{m \times p}$$

Linear approximation: The Jacobian $J_g(x)$ is the best local linear approximation to g at x: $g(x + h) \approx g(x) + J_g(x)\, h$.

Special cases we already know:

Map	Derivative object	Size
$g : \mathbb{R} \to \mathbb{R}$	$g'(x)$	1×1
$g : \mathbb{R}^p \to \mathbb{R}$	$\nabla g(x)$	$p \times 1$
$g : \mathbb{R}^p \to \mathbb{R}^m$	$J_g(x)$	$m \times p$

Convention: $\nabla g \in \mathbb{R}^p$ is a column vector; row i of the Jacobian is $\nabla g_i^\top$.

The Chain Rule as Composing Linearizations

Consider the composition $f(x) = \ell(o(x))$, with $o : \mathbb{R}^p \to \mathbb{R}^m$ and $\ell : \mathbb{R}^m \to \mathbb{R}^q$.

Key idea. To linearize the composition, linearize each stage and compose the linearizations.

$$
\begin{aligned}
f(x + \delta x) &= \ell(o(x + \delta x)) \\
&\approx \ell(o(x) + J_o(x)\,\delta x) && \text{linearize } o \text{ at } x \\
&\approx \ell(o(x)) + J_\ell(o(x))\big[J_o(x)\,\delta x\big] && \text{linearize } \ell \text{ at } o(x) \\
&= f(x) + \underbrace{J_\ell(o(x))\,J_o(x)}_{J_f(x)}\,\delta x
\end{aligned}
$$

Therefore: $\boxed{J_f(x) = J_\ell(o(x))\; J_o(x)}$

The Jacobian of a composition is the **product of the Jacobians of each stage**. This is the multivariate chain rule.

Backprop Along a Path: Jacobians Compose, Gradients Push Back

For the composition along a single path $f = \ell \circ o^L \circ \cdots \circ o^1$, the chain rule gives:

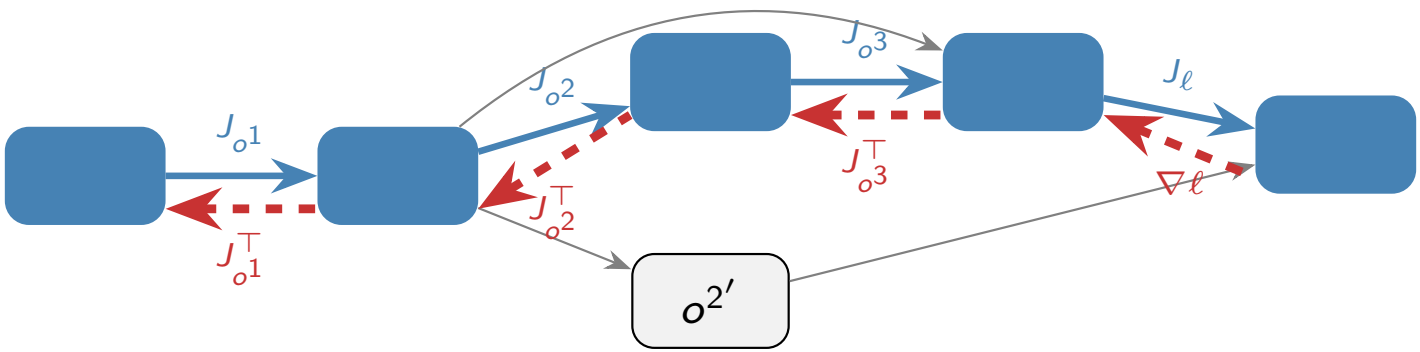
$$J_f = J_\ell \cdot J_{o^L} \cdots J_{o^1}$$

where each J_{o^s} is the Jacobian of o^s w.r.t. o^{s-1}

Since ℓ is scalar-valued, $J_\ell = \nabla \ell^\top$ (a row vector); $J_f = \nabla f^\top$ is also a row vector. Transposing the full product:

$$\boxed{\nabla f = J_{o^1}^\top \cdots J_{o^L}^\top \nabla \ell}$$

Gradients are "pushed back" through the network by Jacobian transposes.



Multivariate Chain Rule: Running Example

We now use the multivariate chain rule to compute $\nabla_x f$ for the function $f(x) = -o_y + \log \sum_{j=1}^{k} e^{o_j}$, where $o = Wx + b$.

	Function	**Jacobian / Gradient**
o	$o(x) = Wx + b$ $o : \mathbb{R}^d \to \mathbb{R}^k$	$J_o(x) = W \in \mathbb{R}^{k \times d}$
ℓ	$\ell(o) = -o_y + \log \sum_j e^{o_j}$ $\ell : \mathbb{R}^k \to \mathbb{R}$	$\nabla \ell(o) = p - e_y \in \mathbb{R}^k$
$f = \ell \circ o$	$f(x) = \ell(o(x))$ $f : \mathbb{R}^d \to \mathbb{R}$	$\nabla_x f = J_o(x)^\top \nabla \ell(o(x))$

Therefore:

$$\nabla_x f = W^\top (p - e_y)$$

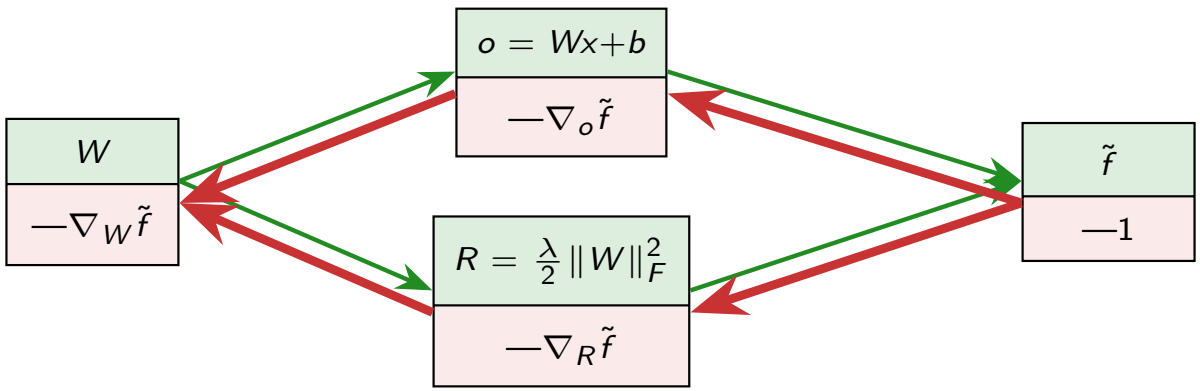
Fan-Out and Summation

We saw how to compute gradients along non-branching paths using the Chain Rule.

But branches exist in DAGs. Suppose we add a regularizer:

$$\tilde{f} = \underbrace{-o_y + \log \sum_j e^{o_j}}_{\text{loss}} + \underbrace{\frac{\lambda}{2} \|W\|_F^2}_{\text{regularizer}}$$

Now W feeds into **two operations**: the linear map and the regularizer.



The total gradient **sums** the two contributions:

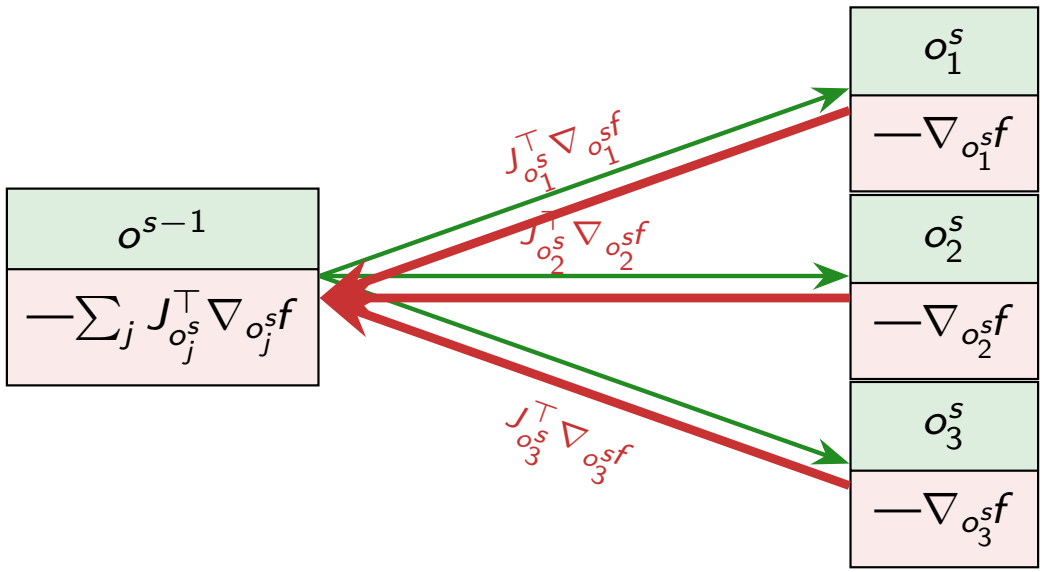
$$\nabla_W \tilde{f} = \underbrace{(\nabla_o f)\, x^\top}_{\text{from loss path}} + \underbrace{\lambda W}_{\text{from regularizer path}}$$

The Backward Rule (General)

If node o^{s-1} feeds into children $o_1^s, \ldots, o_c^s$ (e.g., the next layer's preactivation, a regularizer, a skip connection):

$$\nabla_{o^{s-1}} f = \sum_{j=1}^{c} J_{o_j^s}(o^{s-1})^\top \nabla_{o_j^s} f$$

Each term is a $J^\top$-vector product: **upstream gradient** $\times$ **local Jacobian transpose**.



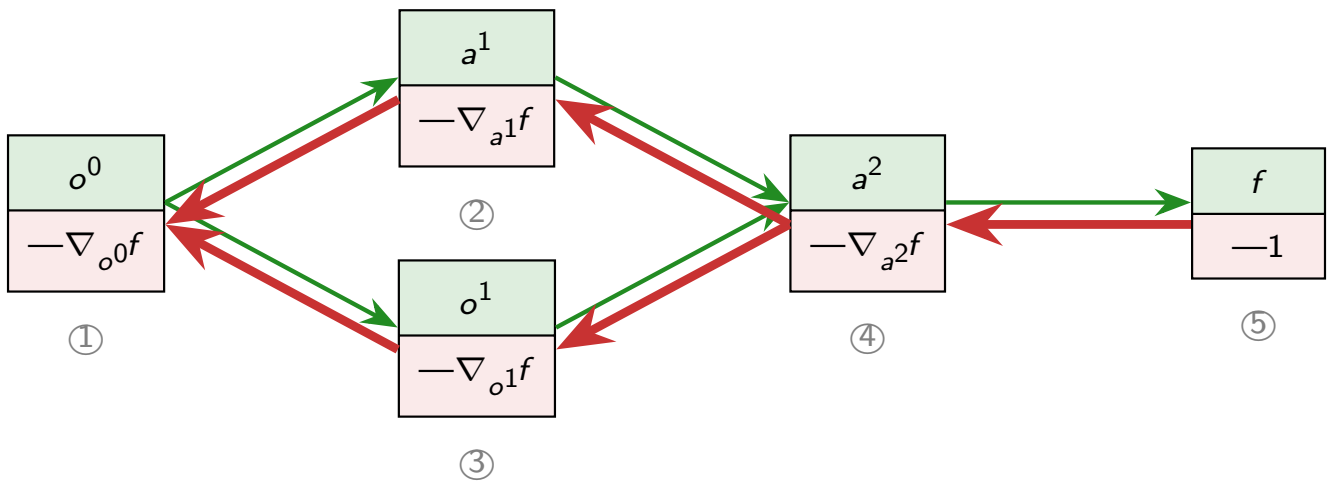
Reverse Automatic Differentiation (AD)

Algorithm. Given a computational DAG with scalar output f:

1. **Forward pass:** Process nodes in **topological order**. Compute and store the value at each node.
2. **Initialize:** Set $\nabla_f f = 1$.
3. **Backward pass:** Process nodes in **reverse topological order**. At each node o^{s-1}:

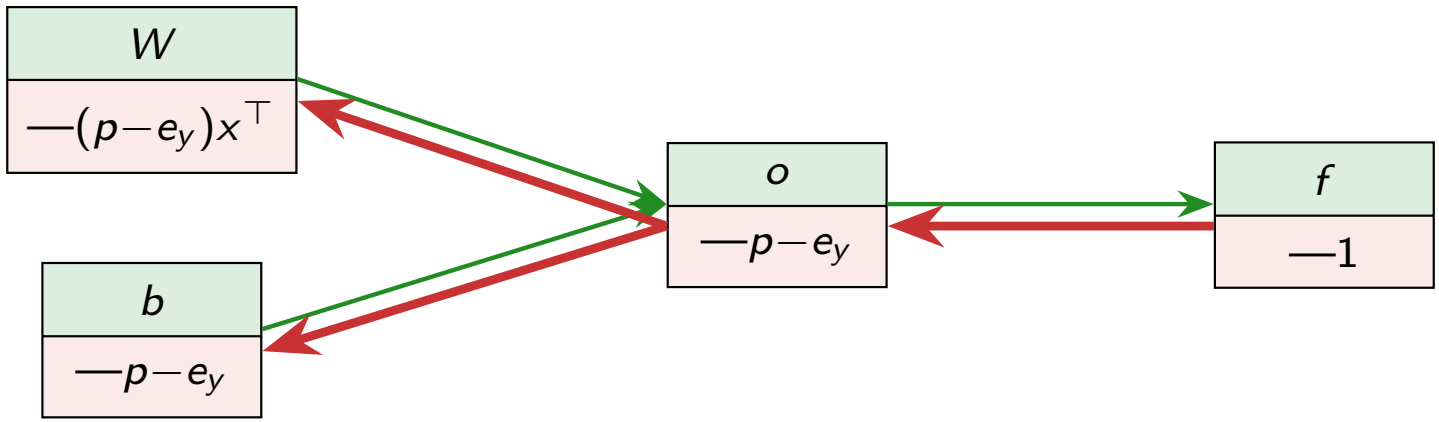
$$\nabla_{o^{s-1}} f = \sum_{j=1}^{c} J_{o_j^s}(o^{s-1})^\top \nabla_{o_j^s} f$$

where $o_1^s, \ldots, o_c^s$ are the children of o^{s-1}.



Running Example: Full Circle

We began with multiclass logistic regression and computed gradients by hand.
Now we can see that hand calculation *was* reverse-mode AD all along:



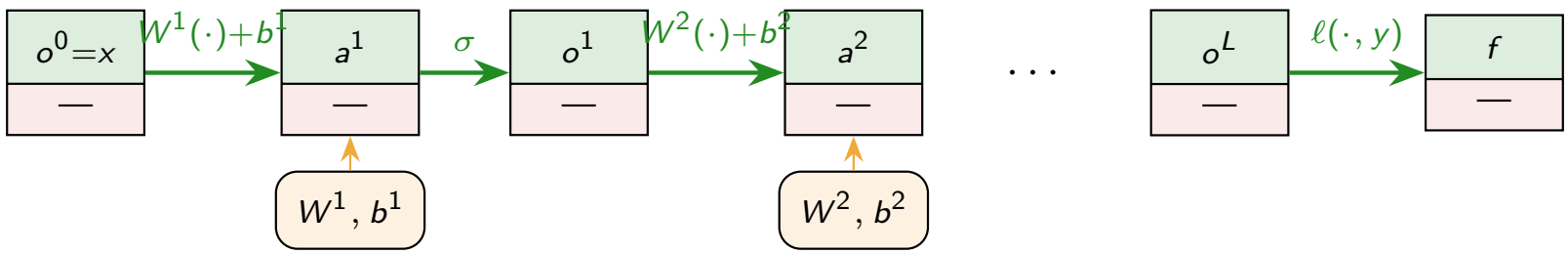
Step	**What we did**	**General principle**
$\nabla_o f = p - e_y$	Differentiated loss directly	Backward rule at sink
$\nabla_b f = \nabla_o f$	Propagated through $o = Wx + b$	$J^\top$-vector product ($J = I$)
$\nabla_W f = (\nabla_o f)\, x^\top$	Outer product with stored input	$J^\top$-vector product for W

From Logistic Regression to MLPs

Our running example was a **one-layer** network:

$$x \xrightarrow{W,b} o \xrightarrow{\text{lse}} f$$

An MLP simply **stacks more layers**, interleaving linear maps with nonlinear activations:



Reverse-mode AD applied to an MLP computational DAG is called **backpropagation**.

MLP: Forward Pass and Loss

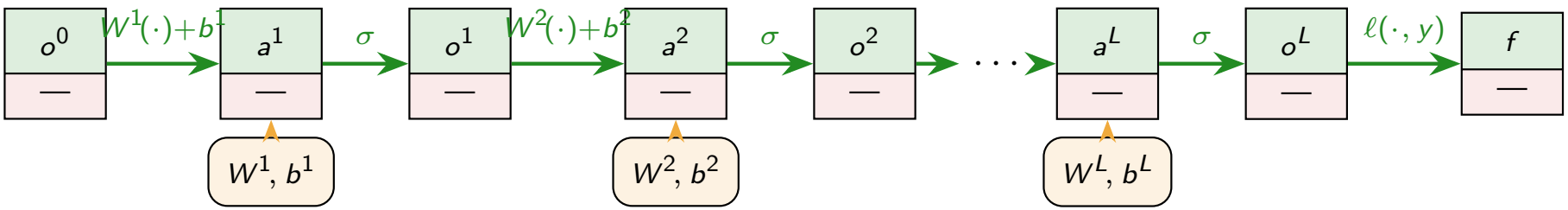
Forward pass: For $s = 1, \ldots, L$:

$$a^s = W^s o^{s-1} + b^s, \qquad o^s = \sigma(a^s)$$

with $o^0 = x$ (input).

Loss:

$$f(\theta) = \ell(o^L, y) + \lambda \sum_{s=1}^{L} R(W^s, b^s)$$

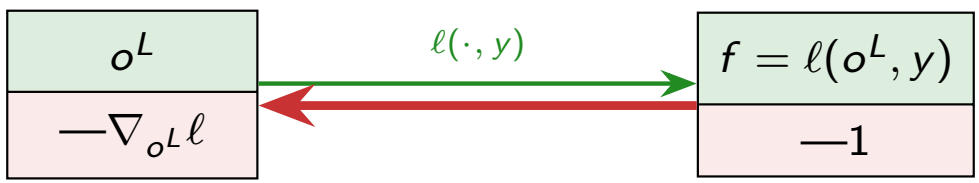


Backprop: Output Layer ($f \to o^L$)

We begin the backward pass at the output.

The loss node has no children, so we differentiate $f = \ell(o^L, y)$ directly:

$$\nabla_{o^L} f = \nabla_{o^L} \ell(o^L, y)$$



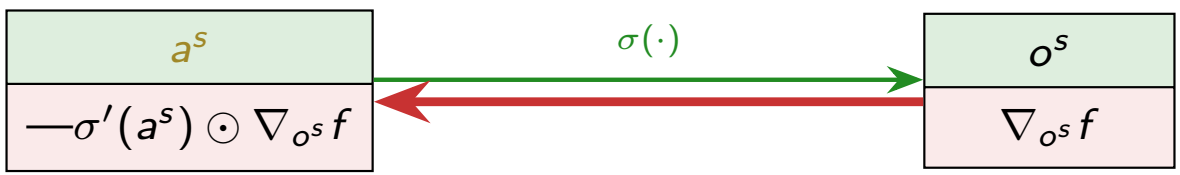
Backprop: Through the Activation ($o^s \to a^s$)

Given $\nabla_{o^s} f$ from the layer above, we pass through the activation node. Because σ acts **elementwise**, output i depends *only* on input i:

$$(J_o(a))_{ij} = \frac{\partial \sigma(a)_i}{\partial a_j} = \begin{cases} \sigma'(a_i) & i = j \\ 0 & i \neq j \end{cases} \quad \Longrightarrow \quad J_o(a) = \text{diag}(\sigma'(a))$$

So the $J^\top$-vector product is just an elementwise multiply. Since $o^s = \sigma(a^s)$:

$$\nabla_{a^s} f = J_{o^s}(a^s)^\top \nabla_{o^s} f = \sigma'(a^s) \odot \nabla_{o^s} f$$

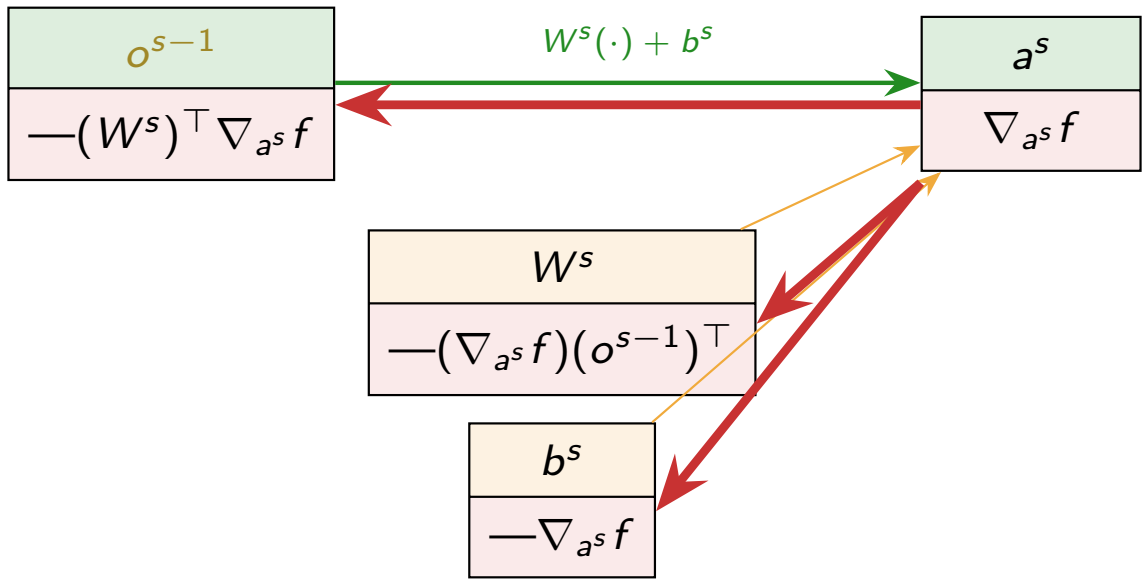


Note: the value a^s was stored during the forward pass. The backward pass through an activation layer is $O(n_s)$ — just a Hadamard product.

Backprop: Through the Preactivation ($a^s \to o^{s-1}$)

Given $\nabla_{a^s} f$, we compute parameter gradients and propagate to the previous layer.

Since $a^s = W^s o^{s-1} + b^s$:



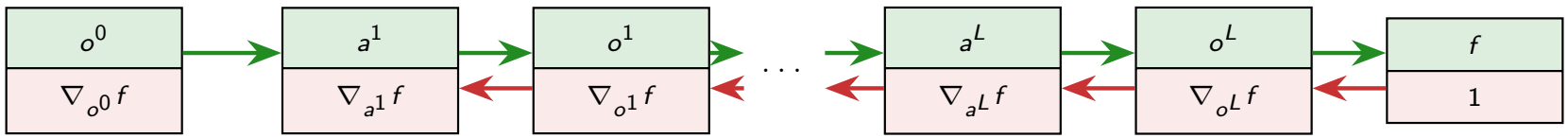
$$\nabla_{W^s} f = (\nabla_{a^s} f)\,(o^{s-1})^\top \qquad n_s \times n_{s-1}$$

$$\nabla_{b^s} f = \nabla_{a^s} f \qquad n_s$$

$$\nabla_{o^{s-1}} f = (W^s)^\top \nabla_{a^s} f \qquad n_{s-1}$$

Note: o^{s-1} was stored during the forward pass.

Backprop on the MLP DAG



Target	**Formula**
Output Loss	$\nabla_{o^L} f = \partial \ell / \partial o^L$
Linear Layer	$\nabla_{a^k} f = \sigma'(a^k) \odot \nabla_{o^k} f$
Weights/Bias	$\nabla_{W^k} f = (\nabla_{a^k} f)(o^{k-1})^\top$
Propagation	$\nabla_{o^{k-1}} f = (W^k)^\top \nabla_{a^k} f$

The Backpropagation Algorithm for MLPs

Here are the equations spelled out in full.

Forward pass. For $s = 1, \ldots, L$:

$$a^s = W^s o^{s-1} + b^s \qquad n_s$$

$$o^s = \sigma(a^s) \qquad n_s$$

Store a^s and o^{s-1}.

Initialize.

$$\nabla_{o^L} f = \nabla_{o^L} \ell(o^L, y) \qquad n_L$$

$$\nabla_{a^L} f = \sigma'(a^L) \odot \nabla_{o^L} f \qquad n_L$$

Backward pass. For $s = L, \ldots, 1$:

$$\nabla_{W^s} f = (\nabla_{a^s} f)\,(o^{s-1})^\top \qquad n_s \times n_{s-1}$$

$$\nabla_{b^s} f = \nabla_{a^s} f \qquad n_s$$

$$\nabla_{o^{s-1}} f = (W^s)^\top \nabla_{a^s} f \qquad n_{s-1}$$

$$\nabla_{a^{s-1}} f = \sigma'(a^{s-1}) \odot \nabla_{o^{s-1}} f \quad (\text{if } s > 1) \qquad n_{s-1}$$

Every step is: **upstream gradient** $\times$ **local Jacobian transpose**.

The Backward Pass Costs $O(T_{\text{fwd}})$

Let T_v^{fwd} be the cost of computing node v in the forward pass, so $T_{\text{fwd}} = \sum_v T_v^{\text{fwd}}$.

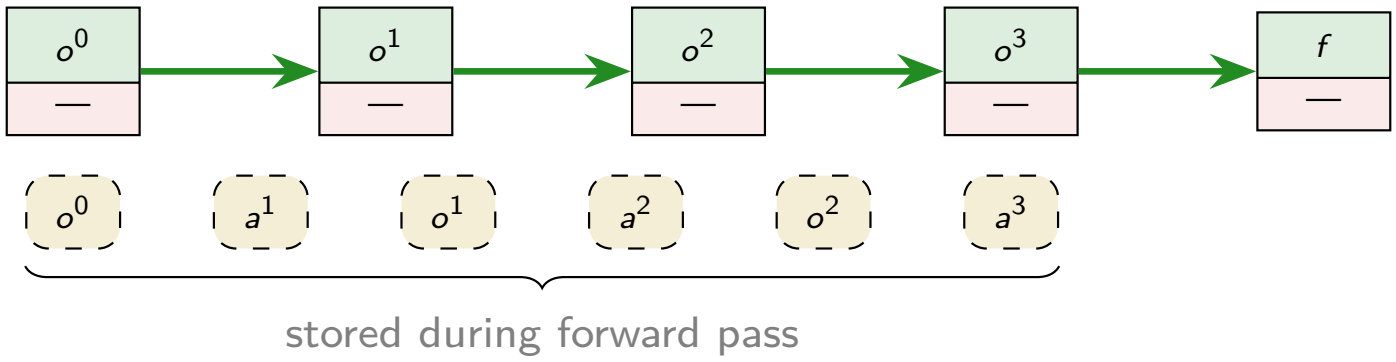
The backward pass at node v computes one $J^\top$-vector product per outgoing edge. For standard operations, each such product costs $O(T_v^{\text{fwd}})$. Summing over all nodes:

$$T_{\text{bwd}} = \sum_v O(T_v^{\text{fwd}}) = O(T_{\text{fwd}})$$

In practice, backward $\approx$ 2–3$\times$ forward (the constant depends on the mix of operations).

Backprop Is Memory-Heavy

The backward pass at layer s reads a^s and o^{s-1}. So the forward pass must **store all intermediate activations**.



During the backward pass, stored values are read and freed layer-by-layer.

Memory budget:

- Parameter memory $O\left(\sum_s n_s n_{s-1}\right)$ is fixed regardless of batch size B.
- Activation memory $O\left(B \cdot \sum_s n_s\right)$ scales linearly with B. For modern networks, activations dominate GPU memory.

Dynamic Computation Graphs

Modern frameworks (PyTorch, JAX) build the DAG **during the forward pass** by recording each operation.



You write the forward pass. The framework computes the backward pass.

Running Example in PyTorch (1/2)

Our running example implemented in PyTorch — every gradient we derived by hand is computed by `loss.backward()`.

```
import torch

d, k = 5, 3                # input dim, num classes
x = torch.randn(d)
y = 1                       # true class
lam = 0.01                  # regularization strength

W = torch.randn(k, d, requires_grad=True)
b = torch.randn(k,    requires_grad=True)

# --- Forward pass (defines the DAG) ---
o = W @ x + b                                  # layer output
ell = -o[y] + torch.logsumexp(o, dim=0)   # cross-
entropy loss
loss = ell + 0.5 * lam * W.norm()**2      # regularized objective

# --- Backward pass (reverse-mode AD) ---
loss.backward()
```

Running Example in PyTorch (2/2)

We can verify that PyTorch's automatic gradients match our hand derivation:

```
# Verify against our hand-derived formulas
p = torch.softmax(o.detach(), dim=0)
e_y = torch.zeros(k); e_y[y] = 1.0

# grad_W loss = (p - e_y) x^T + lam * W
expected_W = (p - e_y).unsqueeze(1) * x + lam * W.detach()
print(torch.allclose(W.grad, expected_W))           # True
print(torch.allclose(b.grad, p - e_y))              # True
```

Key point: You write the forward pass; the framework computes the backward pass. The gradients are exactly the $J^\top$-vector products we derived (plus the regularization contribution λW).

The General Pattern

The same pattern applies to *any* computation:

1. **Record** the computation as a DAG during the forward pass.
2. **Traverse** nodes in reverse topological order.
3. At each node: **upstream gradient** $\times$ **local** $J^\top$, summing over fan-out.

This is the algorithm that powers every deep learning framework.

Summary

1. **Training requires gradients** of compositions of many simple functions.
2. The **Jacobian** $J_g \in \mathbb{R}^{m \times p}$ linearizes each stage. Chain rule: compose the linearizations.
3. **Reverse-mode AD**: traverse the DAG in reverse topological order; at each node, sum $J^\top$-vector products over fan-out. Cost: $O(T_{\text{fwd}})$.
4. **Memory**: backprop stores all activations ($\propto B \times L \times$ width).
5. AD frameworks build the DAG **dynamically**; `loss.backward()` runs the algorithm we derived.

Coming Up

HW4: Manually and efficiently implement backpropagation for an autoencoder in NumPy.

Next lecture: CNN architectures.

Questions?

Appendix Contents

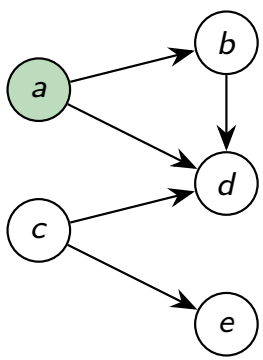
Appendix	**Title**
A	Kahn's Algorithm for Topological Sorting
B	Gradient Checkpointing
C	Softmax Jacobian (for Reference)
D	Forward Mode vs. Reverse Mode Autodiff (Details)
E	Computing $J_z(W)^\top$
F	Fan-Out Creates Summation (Proof)

Appendix A: Kahn's Algorithm for Topological Sorting

Input: DAG $G = (V, E)$. **Output:** A topological ordering, or "cycle detected."

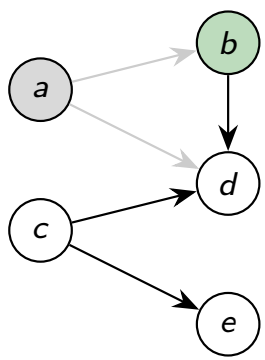
1. Compute in-degree of every node. Enqueue all sources (in-degree 0).
2. While queue nonempty: dequeue v, append to ordering. For each child u: decrement in-degree; if 0, enqueue.
3. All nodes processed $\rightarrow$ valid topological order. Otherwise $\rightarrow$ graph contains a cycle.

Step 1



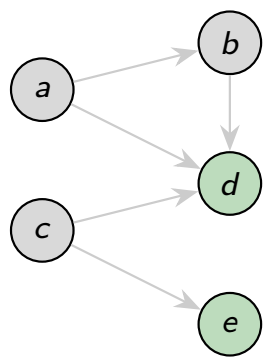
Queue: $\{a, c\}$
Order: [ ]

Step 2



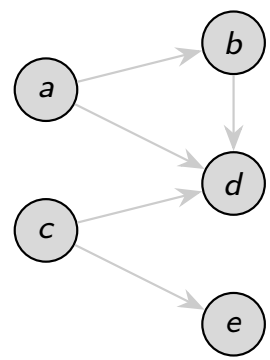
Queue: $\{c, b\}$
Order: $[a]$

Step 3



Queue: $\{d, e\}$
Order: $[a, c, b]$

Step 4



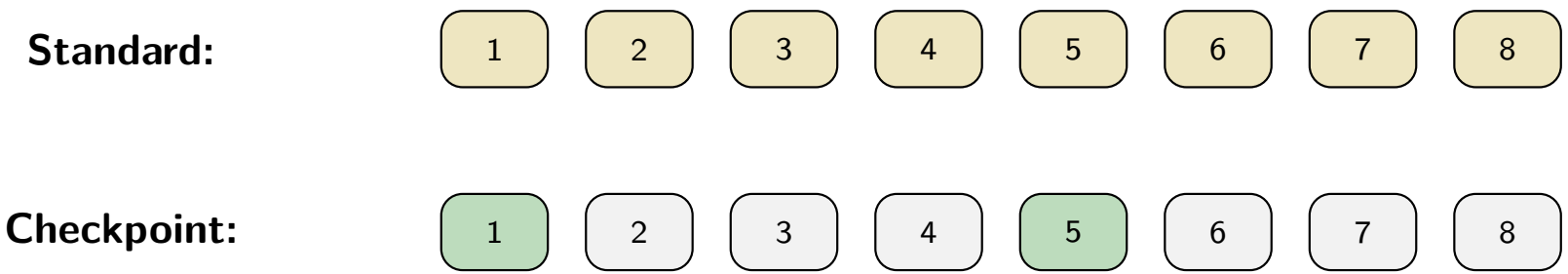
Queue: $\{\}$ ✓
Order: $[a, c, b, d, e]$

Runtime: $O(|V| + |E|)$.

Appendix B: Gradient Checkpointing

Standard backprop stores all activations: memory $\propto L \times$ width.

Gradient checkpointing stores only every k-th layer's activations and **recomputes** the rest during the backward pass.



With $k = \sqrt{L}$:

- **Memory:** $O(\sqrt{L} \times \text{width})$ (down from $O(L \times \text{width})$)
- **Extra cost:** ~ 1 additional forward pass

Useful when GPU memory is the bottleneck (large models, long sequences). A standard technique in modern training pipelines.

Appendix C: Softmax Jacobian (for Reference)

The softmax function $p_i = e^{o_i} / \sum_j e^{o_j}$ has Jacobian:

$$\frac{\partial p_i}{\partial o_j} = p_i(\delta_{ij} - p_j) \qquad \Longrightarrow \qquad J_{\text{softmax}}(o) = \text{diag}(p) - p\,p^\top$$

The cross-entropy loss $\ell = -\log p_y$ composed with softmax gives:

$$\nabla_o f = J_{\text{softmax}}^\top \nabla_p(-\log p_y) = \cdots = p - e_y$$

The log-sum-exp formulation $\ell(o) = -o_y + \log \sum_j e^{o_j}$ yields the same result directly, without computing the softmax Jacobian:

$$\frac{\partial \ell(o)}{\partial o_i} = -\delta_{iy} + \frac{e^{o_i}}{\sum_j e^{o_j}} = p_i - \delta_{iy}$$

The log-sum-exp route is simpler, numerically stabler, and avoids intermediate quantities that cancel. This is why we used it in the lecture.

Appendix D1: Reverse Mode — Computing Rows of J_f

For a composition $f = \ell \circ o^L \circ \cdots \circ o^1$ with scalar output, the general chain rule gives:

$$J_f(x) = J_\ell \cdot J_{o^L} \cdots J_{o^1}(x)$$

Reverse mode initializes with the scalar seed $\bar{f} = 1$ and multiplies the chain of Jacobians from *right to left*:

$$\nabla_x f = J_{o^1}(x)^\top J_{o^2}^\top \cdots J_{o^L}^\top \cdot 1$$

One backward pass yields the **full gradient** w.r.t. x (and all intermediate variables). Hence "reverse mode": the chain is traversed in the reverse direction to the forward computation.

More generally, for $f : \mathbb{R}^p \to \mathbb{R}^m$, seeding with $\lambda = e_i \in \mathbb{R}^m$ gives $J_f(x)^\top e_i$, the i-th **row** of $J_f(x)$ (transposed). So m reverse passes with seeds $e_1, \ldots, e_m$ recover the full Jacobian one row at a time.

Cost: For scalar output ($m = 1$), one pass suffices for all p input gradients: $O(T_{\text{fwd}})$ total.

Appendix D1 (cont.): Forward Mode — Computing Columns of J_f

For the same composition $f = \ell \circ o^L \circ \cdots \circ o^1$, the general chain rule gives:

$$J_f(x) = J_\ell \cdot J_{o^L} \cdots J_{o^1}(x)$$

Forward mode initializes with a seed direction $v \in \mathbb{R}^p$ and multiplies the chain of Jacobians from *left to right*:

$$J_f(x)\, v = J_\ell \cdot J_{o^L} \cdots (J_{o^1}(x)\, v)$$

One forward pass yields one directional derivative $J_f(x)\, v$.

Seeding with $v = e_j \in \mathbb{R}^p$ gives $J_f(x)\, e_j$, the j-th **column** of $J_f(x)$. So p forward passes with seeds $e_1, \ldots, e_p$ recover the full Jacobian, one column at a time.

Cost: For a single seed v, one pass costs $O(T_{\text{fwd}})$. To compute the full Jacobian (p columns), we need p passes.

Appendix D1 (cont.): Forward vs. Reverse Mode Summary

	Forward mode	**Reverse mode**
One pass gives	$J_f(x)\,v$ (one column-direction of J_f)	$J_f(x)^\top \lambda$ (one row-direction of J_f)
Full J_f	p passes ($v = e_1, \ldots, e_p$)	m passes ($\lambda = e_1, \ldots, e_m$)
Efficient when	p small (few inputs)	m small (few outputs)
ML use case	sensitivity analysis	gradient computation

For ML: $m = 1$ (scalar loss), $p =$ millions of parameters. Reverse mode needs **one pass** vs. forward mode's p passes.

Column vs. row interpretation: Forward mode builds J_f column by column (one column per seed e_j). Reverse mode builds J_f row by row (one row per seed e_i). For scalar output, J_f has one row, so reverse mode computes it in a single pass.

Appendix D2: What Is Sensitivity Analysis?

Given a function $f : \mathbb{R}^p \to \mathbb{R}^m$, **sensitivity analysis** asks: how does each output f_i respond to perturbations in each input x_j? This requires the full Jacobian $J_f(x) \in \mathbb{R}^{m \times p}$.

Why we care: In engineering, control, and scientific modeling, f maps a small number of design parameters (p small) to many observable outputs (m large). Understanding which parameters most affect which outputs is critical for design and debugging.

Example. An airfoil is described by $p = 5$ shape parameters. A CFD simulation produces $m = 10{,}000$ pressure measurements on its surface. Which shape parameter most affects pressure at the trailing edge?

Appendix E1: The Problem — When the Jacobian Is a Tensor

In our running example, $z = Wx + b$ with $z \in \mathbb{R}^k$ and $W \in \mathbb{R}^{k \times d}$.

The Jacobian $J_z(W)$ should describe how perturbations $\delta W \in \mathbb{R}^{k \times d}$ map to perturbations $\delta z \in \mathbb{R}^k$. Since the "input" W has kd entries and the output z has k entries, $J_z(W)$ is formally a $k \times (k \times d)$ "matrix": a third-order tensor.

The good news: We never need to write down or store this tensor. We only need to compute $J_z(W)^\top h$ for a given vector $h \in \mathbb{R}^k$.

Strategy:

1. Compute $J_z(W) \cdot \delta W$ (the Jacobian *applied to* a perturbation).
2. Use this to figure out $J_z(W)^\top h$ via the definition of the adjoint.

Appendix E2: The Trace Inner Product

To work with matrix-valued quantities, we need an inner product on $\mathbb{R}^{k \times d}$.

The **trace inner product** (a.k.a. Frobenius inner product) is:

$$\langle A, B \rangle_F = \text{tr}(A^\top B) = \sum_{i=1}^{k} \sum_{j=1}^{d} A_{ij} B_{ij}$$

This is the "dot product of matrices" — just sum the elementwise products.

Dimensions: $A, B \in \mathbb{R}^{k \times d}$, and $\langle A, B \rangle_F \in \mathbb{R}$.

Key property (cyclicity of trace):

$$\text{tr}(A^\top B) = \text{tr}(B^\top A) = \text{tr}(BA^\top) \quad \text{(when the products are square)}$$

A useful special case: if $u \in \mathbb{R}^k$ and $v \in \mathbb{R}^d$, then $uv^\top \in \mathbb{R}^{k \times d}$ and

$$\langle A, uv^\top \rangle_F = \text{tr}(A^\top uv^\top) = v^\top A^\top u = (Av)^\top u = \langle Av, u \rangle_{\mathbb{R}^k}$$

Appendix E3: Computing $J_z(W)^\top h$ via the Adjoint

Step 1: Apply the Jacobian to a perturbation.
Since $z(W) = Wx + b$, perturbing W by $\delta W \in \mathbb{R}^{k \times d}$:

$$z(W + \delta W) = (W + \delta W)x + b = z + (\delta W)\, x$$

So $J_z(W) \cdot \delta W = (\delta W)\, x$. Input: $\delta W \in \mathbb{R}^{k \times d}$. Output: $(\delta W)x \in \mathbb{R}^k$.

Step 2: Use the definition of the adjoint.
For any linear map L, the adjoint $L^\top$ satisfies: $\langle Lv, w \rangle = \langle v, L^\top w \rangle$ for all v, w.

Here $L = J_z(W)$, $v = \delta W \in \mathbb{R}^{k \times d}$, and $w = h \in \mathbb{R}^k$. The inner products are:

Left side: $\langle J_z(W)\, \delta W, h \rangle_{\mathbb{R}^k}$ (standard dot product in $\mathbb{R}^k$).
Right side: $\langle \delta W, J_z(W)^\top h \rangle_F$ (trace inner product in $\mathbb{R}^{k \times d}$).

Appendix E3 (cont.): The Adjoint Computation

We need: $\langle J_z(W)\,\delta W,\ h\,\rangle_{\mathbb{R}^k} = \langle \delta W,\ J_z(W)^\top h\,\rangle_F$ for all δW.

Step 3: Simplify the left side.

$$
\begin{aligned}
\langle J_z(W)\,\delta W, h\rangle_{\mathbb{R}^k} &= \langle (\delta W)\,x, h\rangle_{\mathbb{R}^k} && \text{(Step 1: } J_z(W)\cdot\delta W = (\delta W)x\text{)}\\
&= h^\top\big((\delta W)\,x\big) && \text{(dot product} = u^\top v\text{)}\\
&= \mathrm{tr}\big(h^\top(\delta W)x\big) && \text{(}1\times 1 \text{ matrix} = \text{its trace)}\\
&= \mathrm{tr}\big(x\,h^\top(\delta W)\big) && \text{(cyclicity: } \mathrm{tr}(ABC)=\mathrm{tr}(CAB)\text{)}\\
&= \mathrm{tr}\big((hx^\top)^\top \delta W\big) && \text{(}(xh^\top)^\top = hx^\top\text{; rearrange)}\\
&= \langle hx^\top, \delta W\rangle_F && \text{(definition of trace inner product)}
\end{aligned}
$$

So $\langle \delta W, J_z(W)^\top h\rangle_F = \langle \delta W, hx^\top\rangle_F$ for all δW.
Therefore:

$$\boxed{J_z(W)^\top h = h\,x^\top} \qquad \underbrace{h}_{k\times 1}\ \underbrace{x^\top}_{1\times d} = \underbrace{hx^\top}_{k\times d}$$

Setting $h = \nabla_o f = p - e_y$: $\nabla_W f = J_z(W)^\top \nabla_o f = (p - e_y)\,x^\top$.

Appendix F: Fan-Out Creates Summation (Proof)

Let v be a node that feeds into children $u_1, \ldots, u_c$, so $f = f(u_1(v), \ldots, u_c(v))$.

The **local Jacobian** at a node with input v and output u is $J_u(v) := \partial u / \partial v$.

By the law of total differentiation, for each coordinate v_ℓ:

$$\frac{\partial f}{\partial v_\ell} = \sum_{j=1}^{c} \sum_{m} \frac{\partial f}{\partial (u_j)_m} \frac{\partial (u_j)_m}{\partial v_\ell} = \sum_{j=1}^{c} \left[J_{u_j}(v)^\top \nabla_{u_j} f \right]_\ell$$

Therefore in vector form:

$$\boxed{\nabla_v f = \sum_{j=1}^{c} J_{u_j}(v)^\top \nabla_{u_j} f}$$

Fan-out creates summation: when v feeds into multiple children, its gradient accumulates one $J^\top$-vector product per child.