CS 170: Backpropagation — Lecture Notes

Course: CS 170 — Efficient Algorithms and Intractable Problems, Spring 2026
Instructors: Lijie Chen, Umesh V. Vazirani
Lecture 10 (follows Dynamic Programming, precedes FFT)

The Goal

Suppose we have a function $L = L(x_1, x_2, \ldots, x_d)$ that depends on many input variables — potentially millions or more. We want to compute **all** partial derivatives:

$$\frac{\partial L}{\partial x_1}, \quad \frac{\partial L}{\partial x_2}, \quad \ldots, \quad \frac{\partial L}{\partial x_d}$$

This is a fundamental computational task with many applications. Most notably, it is the core subroutine of **gradient descent** in modern machine learning, where L is a loss function depending on billions of parameters, and the partial derivatives tell us how to update each parameter to reduce the loss.

Why is this hard? A naive approach is to compute each partial derivative separately: perturb x_i by a tiny ϵ, re-evaluate L, and approximate $\frac{\partial L}{\partial x_i} \approx \frac{L(\ldots, x_i+\epsilon, \ldots) - L(\ldots, x_i, \ldots)}{\epsilon}$. This requires $d+1$ evaluations of L — one per input variable. When d is in the billions, this is far too slow.

Can we do better? The key idea is to represent L as a **computational graph** (a DAG) and use dynamic programming to compute all d gradients in a single backward pass over the graph — at roughly the same cost as evaluating L once. This algorithm is called **backpropagation**.

Part 1: Computational Graphs

1.1 Definition

A **computational graph** is a directed acyclic graph (DAG) $G = (V, E)$ where:

- **Source nodes** (in-degree 0) are **inputs**: variables $x_1, x_2, \ldots, x_d$ and constants
- **Internal nodes** represent **elementary operations**: $+, \times, \exp, \log, \sigma, \max$, etc.
- There is a single **sink node** (out-degree 0) representing the **output** L
- Each node v computes a function of its children's values

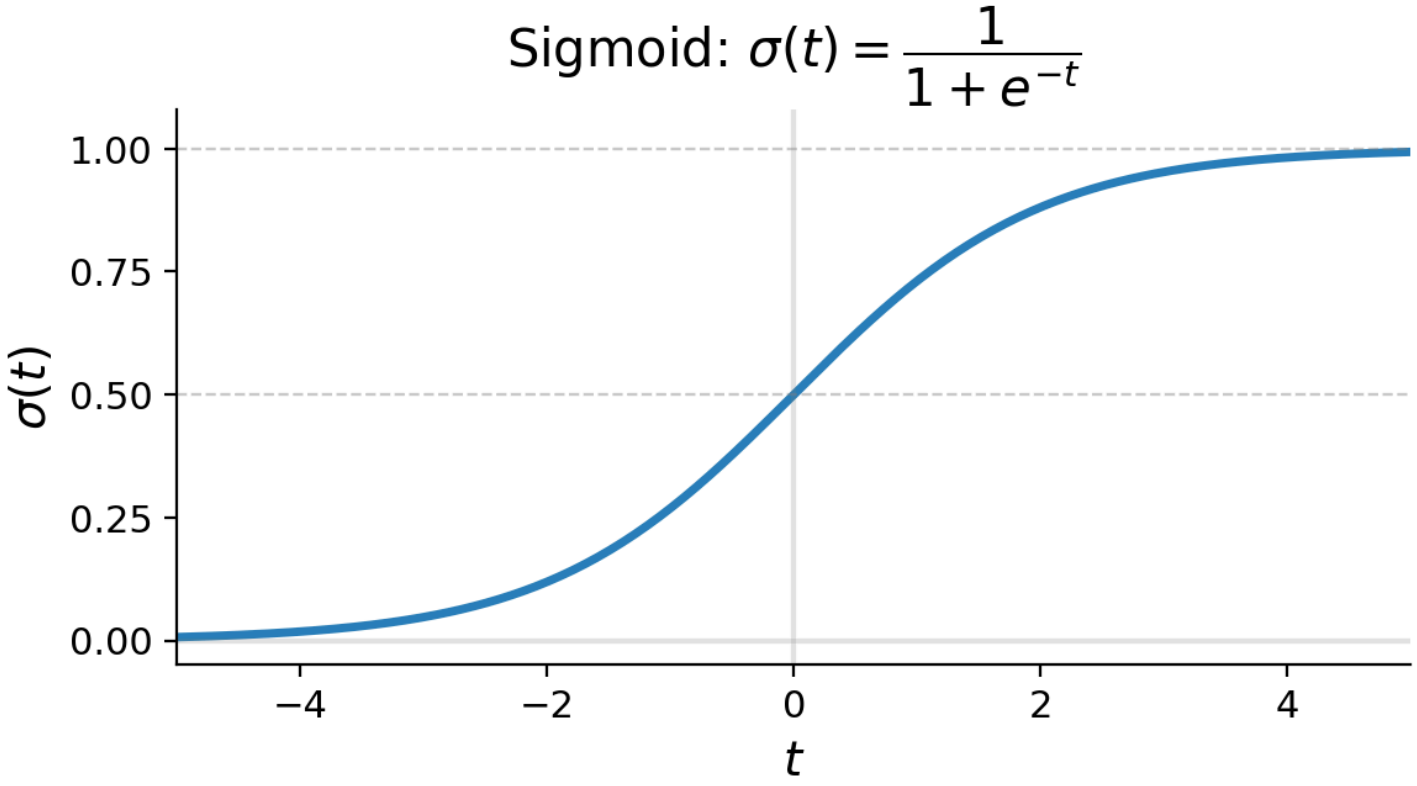
Terminology: We think of the graph as rooted at L. If there is a directed edge $u \to v$ (meaning u is used to compute v), we call u a **child** of v and v a **parent** of u. A node can have multiple children (e.g., $z = x + y$ has two children x and y) and multiple parents (e.g., x may be used in several computations). Source nodes (inputs) are the **leaves** and the output L is the **root**.

1.2 Example

Consider the function:

$$L(x, y) = (x + y) \cdot \sigma(x)$$

where $\sigma(t) = 1/(1 + e^{-t})$ is the sigmoid function.

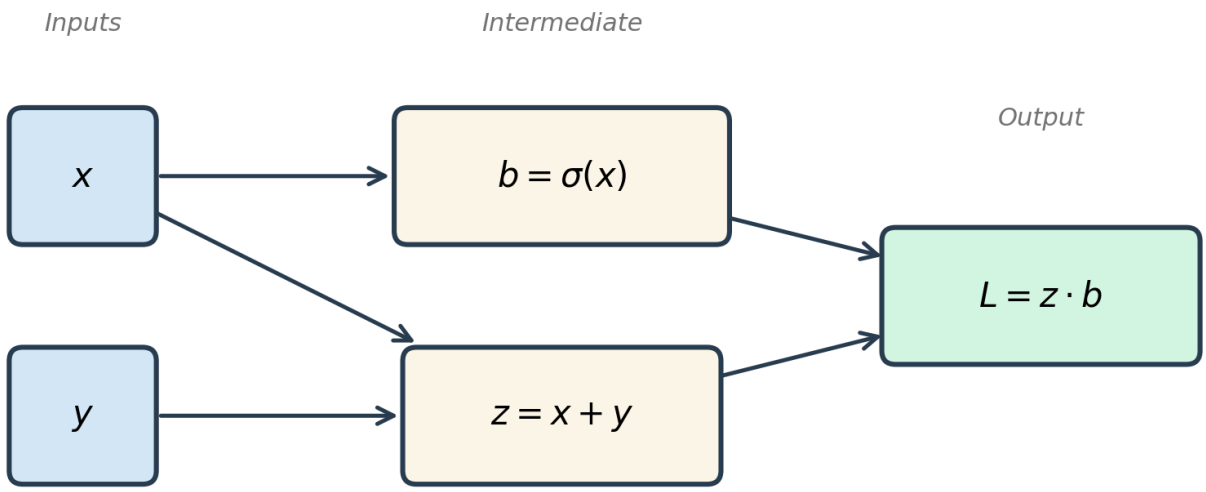


We introduce intermediate variables:

- $z = x + y$
- $b = \sigma(x)$

- $L = z \cdot b$

The resulting computational graph is:



Note that x has **two parents** z and b — it is used by both. This will be important during the backward pass: the gradient $\frac{\partial L}{\partial x}$ must **accumulate contributions from both paths**.

Nodes in topological order: x, y, z, b, L.

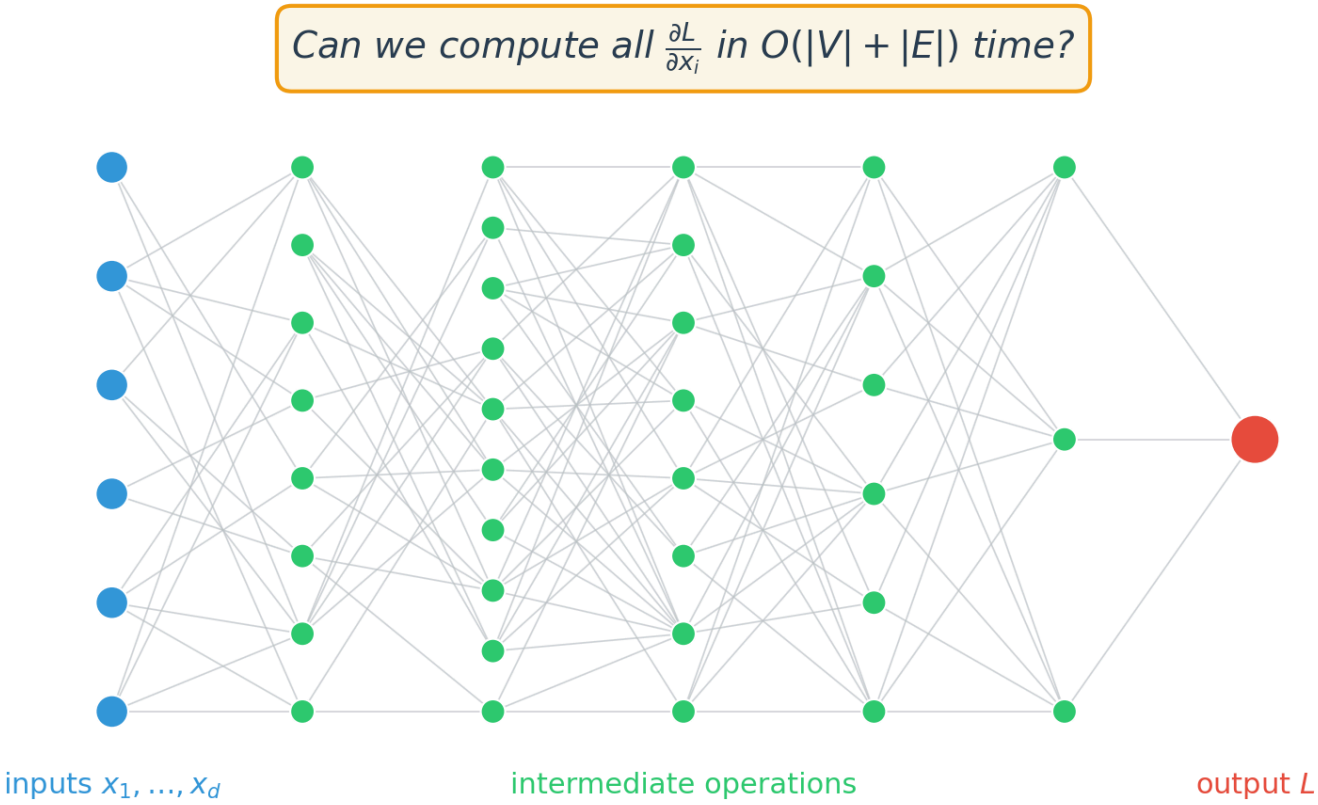
1.3 The Forward Pass

The **forward pass** is simply computing the value of each node from the inputs to the output, following topological order. Since the graph is a DAG, every node's children (inputs) are computed before the node itself.

Cost: $O(|V| + |E|)$ — each node is visited once, each edge is traversed once.

1.4 Key Question

Given the computational graph and the forward pass values, can we compute **all** partial derivatives $\frac{\partial L}{\partial x_i}$ efficiently?



Part 2: The Backpropagation Algorithm

2.1 Intuition

How does changing an input u_i affect the output L? Node u_i influences L only through its **parents** — the nodes that directly use u_i as an input. Each parent u_j contributes to $\frac{\partial L}{\partial u_i}$ by an amount equal to:

$$\underbrace{\frac{\partial L}{\partial u_j}}_{\text{how much } L \text{ changes per unit change in } u_j} \times \underbrace{\frac{\partial u_j}{\partial u_i}}_{\text{how much } u_j \text{ changes per unit change in } u_i}$$

Summing over all parents gives the total effect. This gives us a **recurrence**: to compute $\frac{\partial L}{\partial u_i}$, we need $\frac{\partial L}{\partial u_j}$ for all parents u_j of u_i — which are later in topological order. So we compute in **reverse topological order**, just like DP!

2.2 Algorithm

Let $u_1, u_2, \ldots, u_n$ be all the nodes of the computational graph, sorted in **topological order**. The first d nodes $u_1 = x_1, \ldots, u_d = x_d$ are the inputs, and the last node $u_n = L$ is the output.

Phase 1 — Forward Pass. Compute the value of each node in topological order:

$$\text{For } i = 1, 2, \ldots, n: \qquad u_i = \begin{cases} x_i & \text{if } i \leq d \text{ (input node)} \\ \text{op}_{u_i}(\text{children's values}) & \text{otherwise} \end{cases}$$

Phase 2 — Backward Pass. Compute the gradient $\frac{\partial L}{\partial u_i}$ for each node in **reverse** topological order:

Initialize: $\frac{\partial L}{\partial u_n} = 1$.

$$\text{For } i = n-1, n-2, \ldots, 1: \qquad \frac{\partial L}{\partial u_i} = \sum_{u_j \in \text{parents}(u_i)} \frac{\partial L}{\partial u_j} \cdot \frac{\partial u_j}{\partial u_i}$$

where $\text{parents}(u_i)$ denotes the set of nodes that use u_i as an input, and $\frac{\partial u_j}{\partial u_i}$ is the **local derivative** of the operation at node u_j with respect to its input u_i, evaluated using the forward pass values.

Result: After the backward pass, $\frac{\partial L}{\partial u_i}$ holds the correct partial derivative for every node — in particular, $\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_d}$ are the gradients with respect to all inputs.

2.3 Connection to Dynamic Programming

Backpropagation **is** dynamic programming on a DAG. The table below summarizes the structural parallel:

DP on DAGs	Backpropagation
Recurrence on subproblems	Recurrence on partial derivatives
Topological order (forward)	Reverse topological order (backward)
Base case at sources	Base case at root ($\frac{\partial L}{\partial L} = 1$)
Memoize subproblem solutions	Memoize (cache) gradient values
Avoids recomputing overlapping subproblems	Avoids enumerating exponentially many paths

Counting Paths in a DAG

Recall the classic DP problem: given a DAG G with source s and sink t, count the number of directed paths from s to t.

Recurrence: For each node v, let $\text{paths}(v)$ = number of directed paths from v to t:

$$\text{paths}(v) = \sum_{u \in \text{parents}(v)} \text{paths}(u)$$

Base case: $\text{paths}(t) = 1$. Compute in reverse topological order. Cost: $O(|V| + |E|)$.

Now look at the backpropagation recurrence:

$$\frac{\partial L}{\partial v} = \sum_{u \in \text{parents}(v)} \frac{\partial L}{\partial u} \cdot \frac{\partial u}{\partial v}$$

Base case: $\frac{\partial L}{\partial L} = 1$. Compute in reverse topological order. Cost: $O(|V| + |E|)$.

These are the same recurrence! Backpropagation is "weighted path counting" — instead of counting paths, we sum the **product of edge weights (local derivatives) along each path**. In fact, one can show:

> **Fact:** $\frac{\partial L}{\partial x_i} = \sum_{\text{path } P: x_i \to L} \prod_{(u,v) \in P} \frac{\partial v}{\partial u}$
>
> (Sum over all directed paths from x_i to L, of the product of local derivatives along the path.)

This "sum-over-paths" formula is the derivative analog of path counting. The naive approach would enumerate all paths and multiply out the local derivatives for each — but just like in DP, the number of paths can be **exponential**, while the DP/backprop approach computes the same quantity in **linear** time.

Part 3: Worked Example — A Two-Layer Linear Network

3.1 Setup

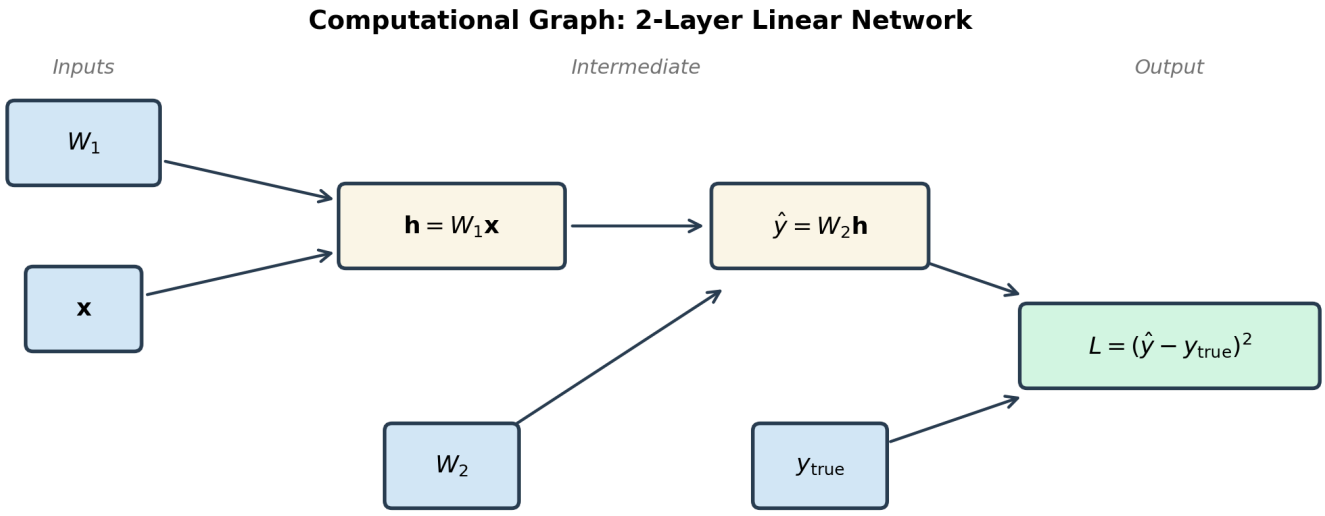
Consider a two-layer linear network with MSE loss:

$$\mathbf{h} = W_1\mathbf{x}, \qquad \hat{y} = W_2\mathbf{h}, \qquad L = (\hat{y} - y_{\text{true}})^2$$

with concrete values:

$$W_1 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \quad W_2 = \begin{pmatrix} 1 & -1 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad y_{\text{true}} = 3$$

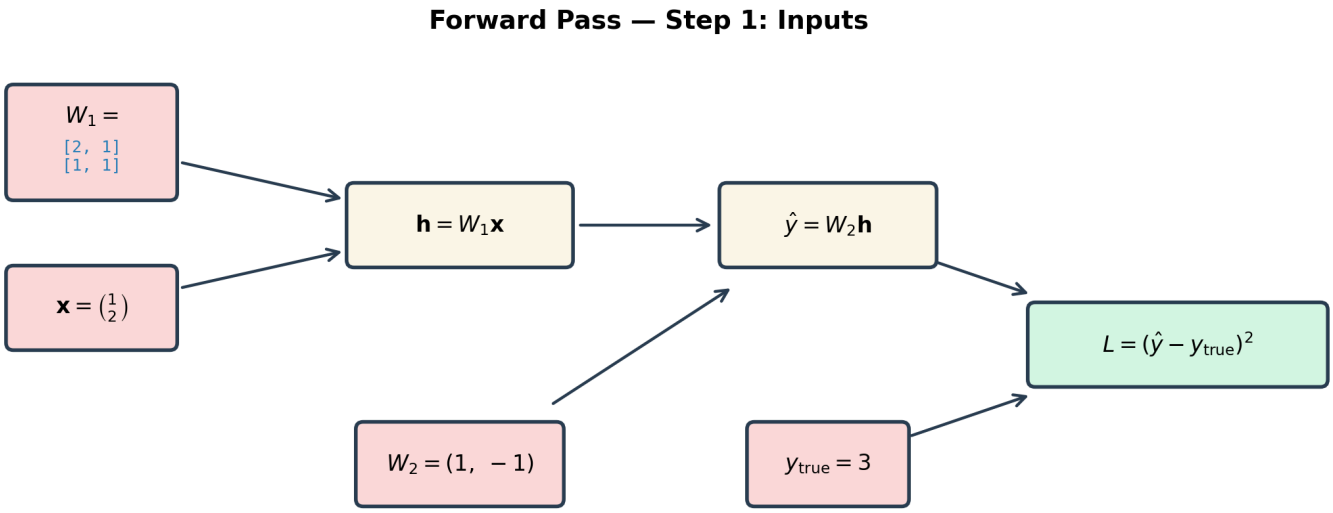
The inputs to this computational graph are W_1 (4 variables), W_2 (2 variables), $\mathbf{x}$ (2 variables), and y_{true} (1 variable) — 9 inputs in total. For simplicity, we draw each of these as a single "big node":



3.2 Forward Pass

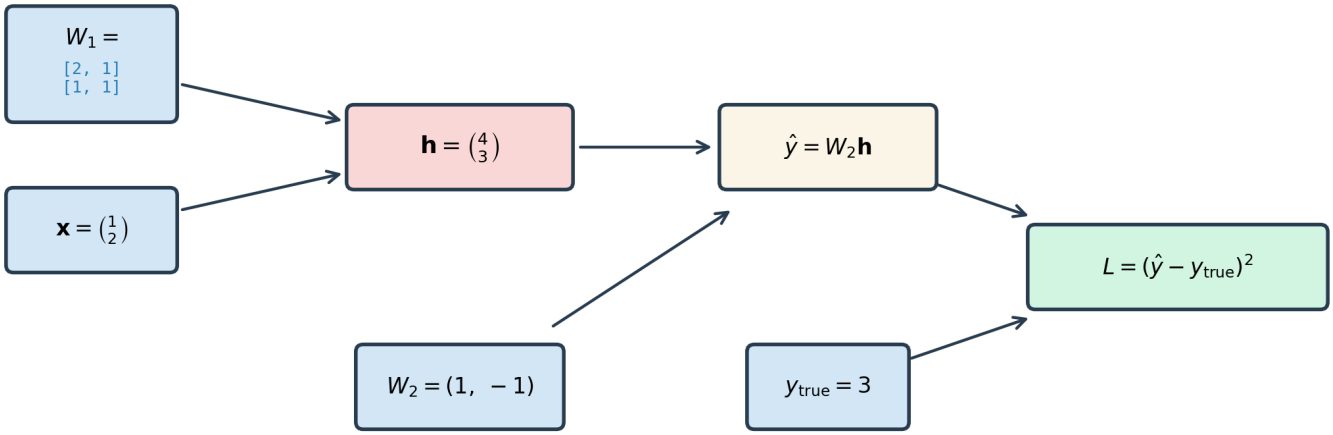
We compute the value of each node layer by layer, following topological order.

Step 1 — Inputs. Set the values of all input nodes:



Step 2 — Hidden layer. Compute $\mathbf{h} = W_1\mathbf{x}$:

$$\mathbf{h} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \cdot 1 + 1 \cdot 2 \\ 1 \cdot 1 + 1 \cdot 2 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \end{pmatrix}$$

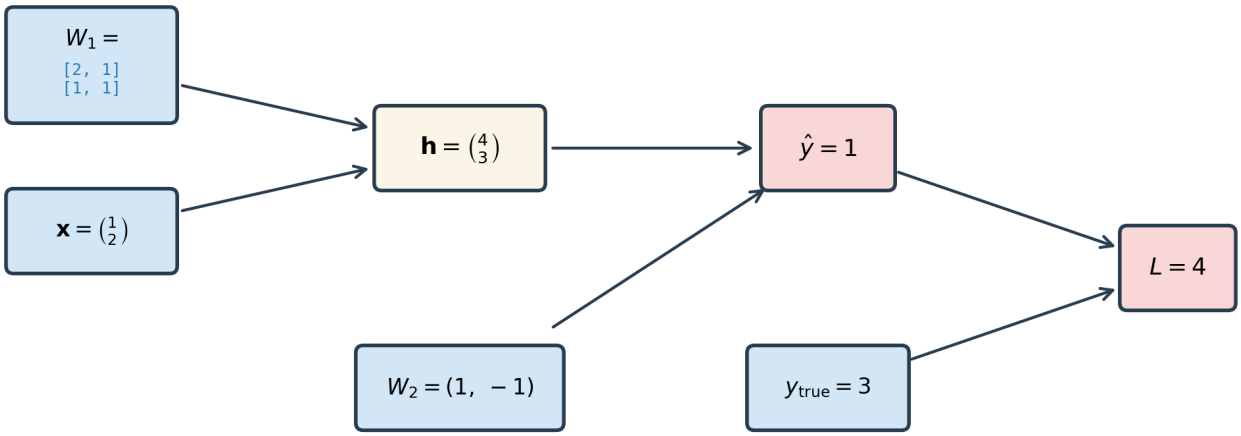


Step 3 — Output and loss. Compute $\hat{y} = W_2\mathbf{h}$ and $L = (\hat{y} - y_{\text{true}})^2$:

$$\hat{y} = \begin{pmatrix} 1 & -1 \end{pmatrix} \begin{pmatrix} 4 \\ 3 \end{pmatrix} = 4 - 3 = 1$$

$$L = (1 - 3)^2 = 4$$

Forward Pass — Step 3: $\hat{y} = W_2\mathbf{h},\ L = (\hat{y} - y_{\text{true}})^2$



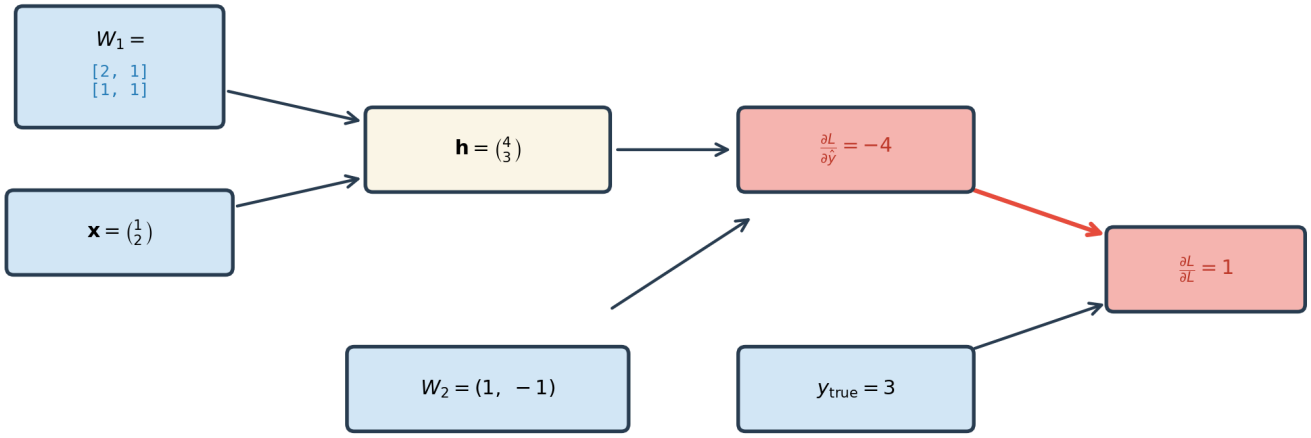
3.3 Backward Pass

Now we compute $\frac{\partial L}{\partial u_i}$ for each node in **reverse** topological order.

Step 1 — From L to $\hat{y}$. The base case is $\frac{\partial L}{\partial L} = 1$. Then:

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y_{\text{true}}) = 2(1 - 3) = -4$$

Backward Pass — Step 1: $\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y_{\text{true}}) = -4$



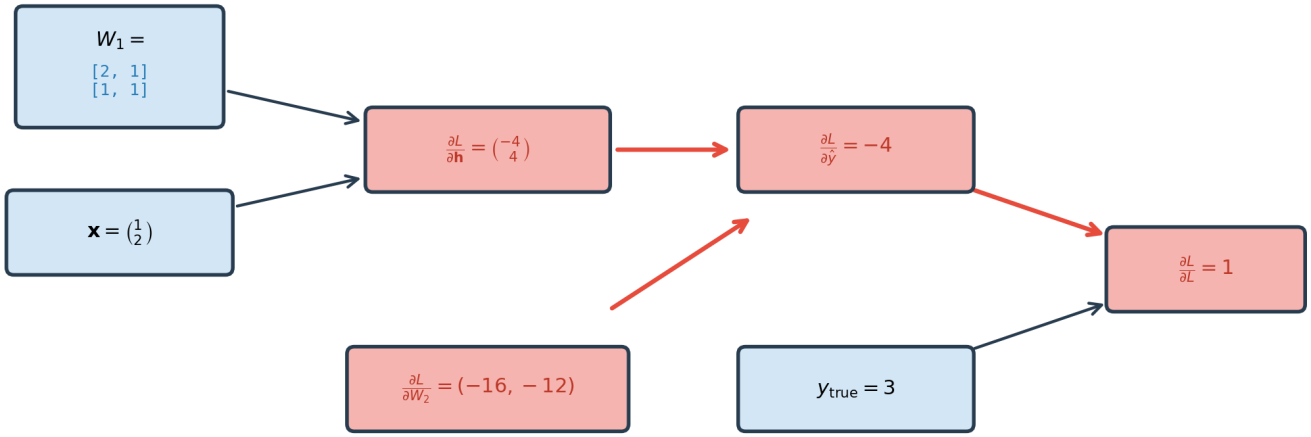
Step 2 — From $\hat{y}$ to $\mathbf{h}$ and W_2. Since $\hat{y} = W_2\mathbf{h} = w_{21}h_1 + w_{22}h_2$:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial \hat{y}} \cdot w_{21} = (-4) \cdot 1 = -4$$

$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial \hat{y}} \cdot w_{22} = (-4) \cdot (-1) = 4$$

In vector form: $\frac{\partial L}{\partial \mathbf{h}} = W_2^T \frac{\partial L}{\partial \hat{y}} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \cdot (-4) = \begin{pmatrix} -4 \\ 4 \end{pmatrix}$.

Backward Pass — Step 2: $\frac{\partial L}{\partial \mathbf{h}} = W_2^T \cdot \frac{\partial L}{\partial \hat{y}}$

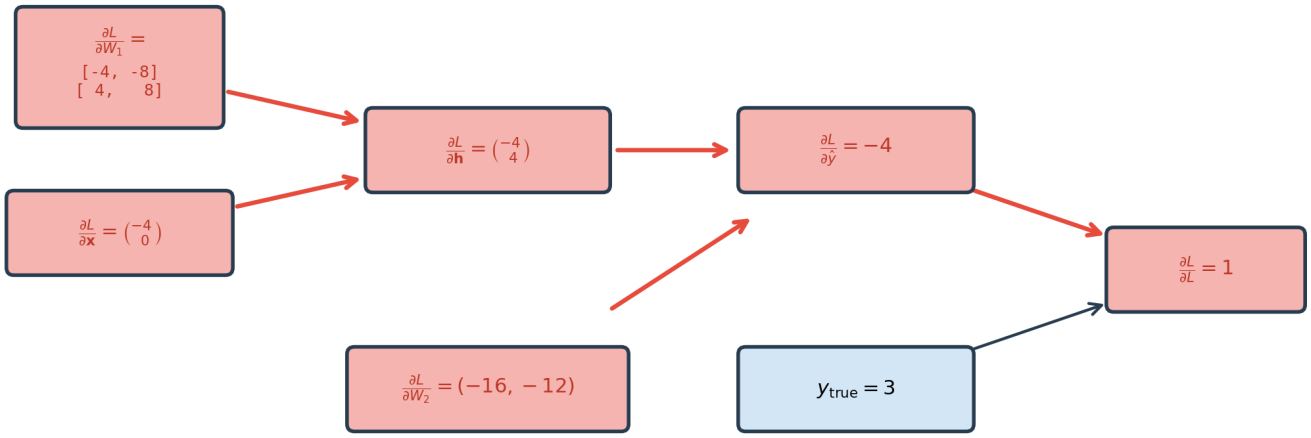


Step 3 — From $\mathbf{h}$ to $\mathbf{x}$ and W_1. Since $\mathbf{h} = W_1\mathbf{x}$:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial x_1} + \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial x_1} = (-4) \cdot 2 + 4 \cdot 1 = -4$$

$$\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial x_2} + \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial x_2} = (-4) \cdot 1 + 4 \cdot 1 = 0$$

Backward Pass — Step 3: all gradients computed



Part 4: Why It Works — The Chain Rule

We have presented the backpropagation algorithm and seen it in action. But why does the recurrence $\frac{\partial L}{\partial u_i} = \sum_{u_j \in \text{parents}(u_i)} \frac{\partial L}{\partial u_j} \cdot \frac{\partial u_j}{\partial u_i}$ actually compute the correct partial derivatives? The answer is the

chain rule from calculus.

4.1 Single-Variable Chain Rule

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

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx} = f'(y) \cdot g'(x)$$

The key idea: we compute $\frac{dL}{dx}$ by going through the intermediate variable y — first computing $\frac{dy}{dx}$, then multiplying by $\frac{dL}{dy}$.

4.2 Multivariable Chain Rule

If L is a function of $x_1, x_2, \ldots, x_k$, and each x_i is a function of t, then:

$$\frac{dL}{dt} = \sum_{i=1}^{k} \frac{\partial L}{\partial x_i} \cdot \frac{dx_i}{dt}$$

In words: when t influences L through **multiple intermediate variables** $x_1, \ldots, x_k$, the total derivative is the **sum** of the contributions along each path.

4.3 The Chain Rule on a Computational Graph

For any node v in the computational graph, let $u_1, u_2, \ldots, u_m$ be the **parents** of v (nodes that take v as input). Applying the multivariable chain rule:

$$\boxed{\frac{\partial L}{\partial v} = \sum_{j=1}^{m} \frac{\partial L}{\partial u_j} \cdot \frac{\partial u_j}{\partial v}}$$

This is exactly the recurrence used in the backward pass of backpropagation. The chain rule guarantees that this recurrence computes the correct partial derivatives, and the reverse topological order ensures that $\frac{\partial L}{\partial u_j}$ is available when we need it to compute $\frac{\partial L}{\partial v}$.

Part 5: Application — Deep Learning

The backpropagation algorithm is the engine behind modern deep learning. Here is a simple example to illustrate how it fits into the bigger picture.

5.1 Example: Cat vs. Dog Classifier

Suppose we want to build a **binary image classifier** that takes an image as input and predicts whether it contains a cat (0) or a dog (1).

The computational graph:

1. **Input:** A grayscale image of size 28×28, flattened into a vector $\mathbf{x} \in \mathbb{R}^{784}$.
2. **Hidden layers:** Apply a sequence of linear transformations and nonlinear activations:

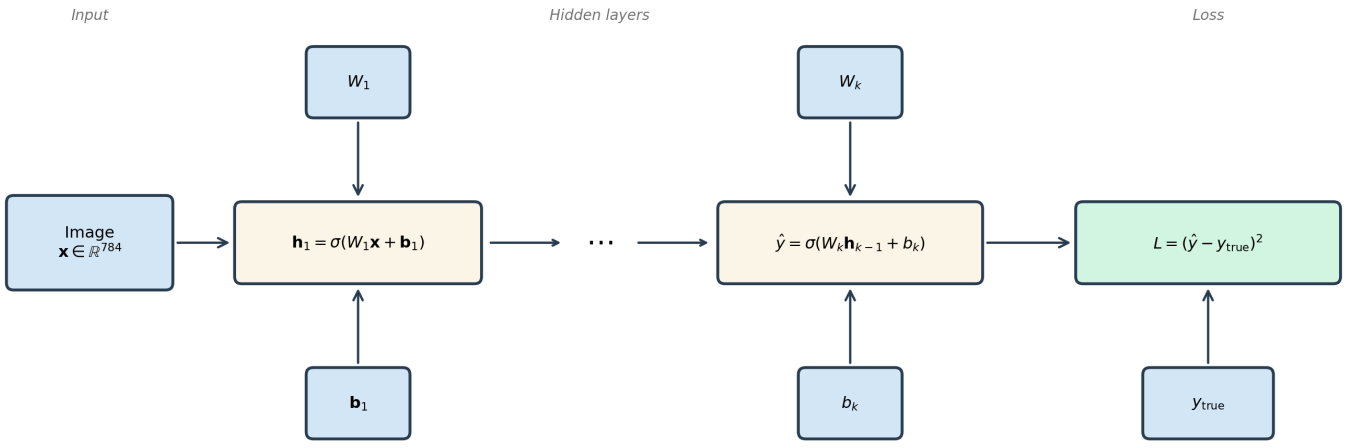
$$\mathbf{h}_1 = \sigma(W_1\mathbf{x} + \mathbf{b}_1), \quad \mathbf{h}_2 = \sigma(W_2\mathbf{h}_1 + \mathbf{b}_2), \quad \dots$$

   where σ is a nonlinear function like sigmoid.
3. **Output:** A final linear layer followed by sigmoid gives a probability:

$$\hat{y} = \sigma(W_k\mathbf{h}_{k-1} + b_k) \in [0, 1]$$

   where $\hat{y} \approx 0$ means "cat" and $\hat{y} \approx 1$ means "dog."
4. **Loss:** We use the MSE loss $L = (\hat{y} - y_{\text{true}})^2$, where $y_{\text{true}} \in \{0, 1\}$ is the correct label.

This entire pipeline — from pixels to loss — is one large computational graph. The inputs include all the weight matrices $W_1, W_2, \dots$ and bias vectors $\mathbf{b}_1, \mathbf{b}_2, \dots$, which can total millions of parameters.



5.2 Training with Backpropagation

To train the classifier, we repeat:

1. **Forward pass:** Feed a training image through the network, compute $\hat{y}$ and L.

2. **Backward pass:** Run backpropagation to compute $\frac{\partial L}{\partial W_i}$ and $\frac{\partial L}{\partial \mathbf{b}_i}$ for every weight and bias.
3. **Gradient step:** Update every parameter in the direction that decreases the loss:

$$W_i \leftarrow W_i - \eta \frac{\partial L}{\partial W_i}, \qquad \mathbf{b}_i \leftarrow \mathbf{b}_i - \eta \frac{\partial L}{\partial \mathbf{b}_i}$$

   where $\eta > 0$ is a small step size (the **learning rate**).

Repeat over many images until the network learns to classify cats and dogs accurately.

The key point: Step 2 is where backpropagation shines. Without it, computing the gradient with respect to millions of parameters would require millions of separate forward passes (one per parameter). Backpropagation computes **all** gradients in a single backward pass — at roughly the same cost as the forward pass itself.