Computation Graphs, Backpropagation and Autodiff

How gradients flow through neural networks

Prof. Nipun Batra

ES 667 · Deep Learning · IIT Gandhinagar

Backprop vocabulary

Lecture 2 built the forward map; today we reverse it

V VISUAL

Last time, an MLP turned an input into a prediction and then one scalar loss. Today we reuse that **same graph** in reverse:

Lecture 2 built the forward map; today we reverse it

V VISUAL

Last time, an MLP turned an input into a prediction and then one scalar loss. Today we reuse that **same graph** in reverse:

General pattern

Forward: compute values

$$\boldsymbol{x} \to f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \hat{\boldsymbol{y}} \to \mathcal{L}$$

Backward: compute sensitivities loss sensitivity for every parameter θ_j

Concrete scalar example

Forward: with $(w, x, b, y) = (2, 3, 1, 10)$

$$\hat{y} = 2 \cdot 3 + 1 = 7, \quad \mathcal{L} = (7 - 10)^2 = 9$$

Backward:

$$\left(\frac{\partial \mathcal{L}}{\partial w}, \frac{\partial \mathcal{L}}{\partial b}\right) = (-18, -6)$$

Lecture 2 built the forward map; today we reverse it

V VISUAL

Last time, an MLP turned an input into a prediction and then one scalar loss. Today we reuse that **same graph** in reverse:

General pattern

Forward: compute values

$$x \to f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \hat{\boldsymbol{y}} \to \mathcal{L}$$

Backward: compute sensitivities loss sensitivity for every parameter θ_j

Concrete scalar example

Forward: with $(w, x, b, y) = (2, 3, 1, 10)$

$$\hat{y} = 2 \cdot 3 + 1 = 7, \quad \mathcal{L} = (7 - 10)^2 = 9$$

Backward:

$$\left(\frac{\partial \mathcal{L}}{\partial w}, \frac{\partial \mathcal{L}}{\partial b} \right) = (-18, -6)$$

same executed graph · values flow forward, gradients flow backward

Read the notation before the algorithm

Symbol	**Meaning in this lecture**
u, v, z	scalars
$\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}$	vectors (bold lowercase)
$\boldsymbol{W}, \boldsymbol{X}$	matrices (bold uppercase)
$\partial\mathcal{L}/\partial u$	how the loss changes when scalar u changes
$\partial\mathcal{L}/\partial\boldsymbol{\theta}$	all parameter derivatives, arranged like $\boldsymbol{\theta}$

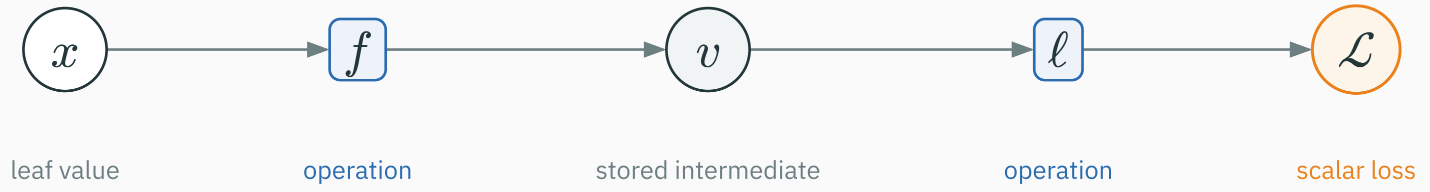
Read the notation before the algorithm

Symbol	Meaning in this lecture
u, v, z	scalars
$\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta}$	vectors (bold lowercase)
$\boldsymbol{W}, \boldsymbol{X}$	matrices (bold uppercase)
$\partial \mathcal{L} / \partial u$	how the loss changes when scalar u changes
$\partial \mathcal{L} / \partial \boldsymbol{\theta}$	all parameter derivatives, arranged like $\boldsymbol{\theta}$

> We keep the full $\partial \mathcal{L} / \partial(\cdot)$ notation visible throughout: the numerator is always the final loss; the denominator names the stored value whose derivative we want.

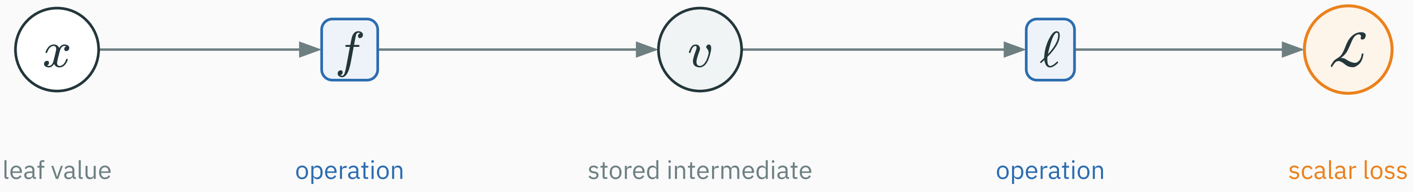
Our computation-graph convention

V VISUAL



Our computation-graph convention

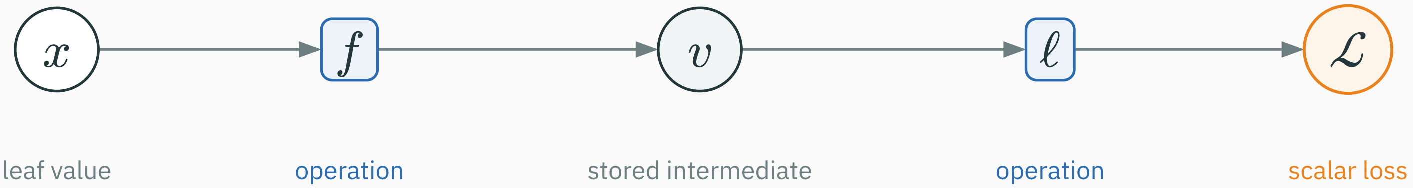
V VISUAL



white circle	given value: input, parameter, or target — a graph leaf
gray circle	computed intermediate value, saved for backward when needed
blue box	operation: consumes value(s) and produces a value
gray arrow	forward dependency from a value to the operation that uses it
orange circle	final scalar loss $\mathcal{L}$; seeds backward with $\partial\mathcal{L}/\partial\mathcal{L} = 1$

Our computation-graph convention

V VISUAL



white circle	given value: input, parameter, or target — a graph leaf
gray circle	computed intermediate value, saved for backward when needed
blue box	operation: consumes value(s) and produces a value
gray arrow	forward dependency from a value to the operation that uses it
orange circle	final scalar loss $\mathcal{L}$; seeds backward with $\partial\mathcal{L}/\partial\mathcal{L} = 1$

circles store values · boxes transform them · arrows show dependency

One derivative, three routes

D DERIVATION

Symbolic = formula $\to$ formula. A person or a computer-algebra system may do the algebra.

$$\mathcal{L} = (wx + b - y)^2, \quad (w, x, b, y) = (2, 3, 1, 10).$$

One derivative, three routes

D DERIVATION

Symbolic = formula $\to$ formula. A person or a computer-algebra system may do the algebra.

$$\mathcal{L} = (wx + b - y)^2, \quad (w, x, b, y) = (2, 3, 1, 10).$$

Route	What it does	What it returns here
symbolic	derive a formula	$\frac{\partial \mathcal{L}}{\partial w} = 2(wx + b - y)x$; substitute $\to -18$

One derivative, three routes

Symbolic = formula $\to$ formula. A person or a computer-algebra system may do the algebra.

$$\mathcal{L} = (wx + b - y)^2, \quad (w, x, b, y) = (2, 3, 1, 10).$$

Route	What it does	What it returns here
symbolic	derive a formula	$\frac{\partial \mathcal{L}}{\partial w} = 2(wx + b - y)x$; substitute $\to -18$
finite difference	evaluate $\mathcal{L}(w \pm \varepsilon)$	approximate number at $w = 2$: -18

One derivative, three routes

Symbolic = formula $\to$ formula. A person or a computer-algebra system may do the algebra.

$$\mathcal{L} = (wx + b - y)^2, \quad (w, x, b, y) = (2, 3, 1, 10).$$

Route	What it does	What it returns here
symbolic	derive a formula	$\frac{\partial \mathcal{L}}{\partial w} = 2(wx + b - y)x$; substitute $\to -18$
finite difference	evaluate $\mathcal{L}(w \pm \varepsilon)$	approximate number at $w = 2$: -18
reverse-mode AD	run local rules backward	number for this run: $(-6) \cdot 3 = -18$

One derivative, three routes

D DERIVATION

Symbolic = formula → formula. A person or a computer-algebra system may do the algebra.

$$\mathcal{L} = (wx + b - y)^2, \quad (w, x, b, y) = (2, 3, 1, 10).$$

Route	What it does	What it returns here
symbolic	derive a formula	$\frac{\partial \mathcal{L}}{\partial w} = 2(wx + b - y)x$; substitute $\to -18$
finite difference	evaluate $\mathcal{L}(w \pm \varepsilon)$	approximate number at $w = 2$: -18
reverse-mode AD	run local rules backward	number for this run: $(-6) \cdot 3 = -18$

formula · approximate value · reverse sweep through the executed graph

Gradient checking: compare one coordinate two ways

I INTERACTIVE

Reverse-mode AD computes the derivative from local rules, up to floating-point arithmetic. Central difference independently approximates the same value: $g_{\mathrm{FD}} = \frac{\mathcal{L}(\theta_j+\varepsilon)-\mathcal{L}(\theta_j-\varepsilon)}{2\varepsilon}$.

Gradient checking: compare one coordinate two ways

I INTERACTIVE

Reverse-mode AD computes the derivative from local rules, up to floating-point arithmetic. Central difference independently approximates the same value: $g_{\mathrm{FD}} = \frac{\mathcal{L}(\theta_j+\varepsilon)-\mathcal{L}(\theta_j-\varepsilon)}{2\varepsilon}$.

```
import torch
def loss(w): return (w*3.0 + 1.0 - 10.0)**2
w, eps = 2.0, 1e-5
g_fd = (loss(w + eps) - loss(w - eps)) / (2*eps)
wt = torch.tensor(w, dtype=torch.float64, requires_grad=True)
loss(wt).backward()
g_ad = wt.grad.item()
print(g_fd, g_ad)  # both approximately -18
```

Gradient checking: compare one coordinate two ways

I INTERACTIVE

Reverse-mode AD computes the derivative from local rules, up to floating-point arithmetic. Central difference independently approximates the same value: $g_{\mathrm{FD}} = \frac{\mathcal{L}(\theta_j+\varepsilon)-\mathcal{L}(\theta_j-\varepsilon)}{2\varepsilon}$.

```
import torch
def loss(w): return (w*3.0 + 1.0 - 10.0)**2
w, eps = 2.0, 1e-5
g_fd = (loss(w + eps) - loss(w - eps)) / (2*eps)
wt = torch.tensor(w, dtype=torch.float64, requires_grad=True)
loss(wt).backward()
g_ad = wt.grad.item()
print(g_fd, g_ad)  # both approximately -18
```

> Use float64 and check only a few coordinates. Finite differences debug gradients; they do not train models because every checked coordinate needs two extra forward passes.

Why train with reverse-mode autodiff?

D DERIVATION

Method	Exact?	Cost for P parameters	Use it for
symbolic	exact formula before numerical evaluation	expression-dependent; algebra can grow	deriving / reasoning

Why train with reverse-mode autodiff?

Method	Exact?	Cost for P parameters	Use it for
symbolic	exact formula before numerical evaluation	expression-dependent; algebra can grow	deriving / reasoning
finite difference	approximate	$2P$ loss evaluations	checking

Why train with reverse-mode autodiff?

DERIVATION

Method	**Exact?**	**Cost for P parameters**	**Use it for**
symbolic	exact formula before numerical evaluation	expression-dependent; algebra can grow	deriving / reasoning
finite difference	approximate	$2P$ loss evaluations	checking
reverse mode	exact local rules; floating-point values	one reverse sweep for scalar $\mathcal{L}$	training

Why train with reverse-mode autodiff?

D DERIVATION

Method	Exact?	Cost for P parameters	Use it for
symbolic	exact formula before numerical evaluation	expression-dependent; algebra can grow	deriving / reasoning
finite difference	approximate	$2P$ loss evaluations	checking
reverse mode	exact local rules; floating-point values	one reverse sweep for scalar $\mathcal{L}$	training

Symbolic to reason · finite differences to check · reverse mode to train.

Backprop returns every parameter sensitivity in one sweep

For each parameter coordinate, gradient descent uses

$$\theta_{j,t+1} = \theta_{j,t} - \eta \frac{\partial \mathcal{L}}{\partial \theta_j}.$$

Backprop returns every parameter sensitivity in one sweep

For each parameter coordinate, gradient descent uses

$$\theta_{j,t+1} = \theta_{j,t} - \eta \frac{\partial \mathcal{L}}{\partial \theta_j}.$$

η is the learning rate; the derivative supplies the direction and magnitude of the local change.

Backprop returns every parameter sensitivity in one sweep

For each parameter coordinate, gradient descent uses

$$\theta_{j,t+1} = \theta_{j,t} - \eta \frac{\partial \mathcal{L}}{\partial \theta_j}.$$

η is the learning rate; the derivative supplies the direction and magnitude of the local change. A modern network has $P = 10^7$ to 10^{11} parameters.

Backprop returns every parameter sensitivity in one sweep

For each parameter coordinate, gradient descent uses

$$\theta_{j,t+1} = \theta_{j,t} - \eta \frac{\partial \mathcal{L}}{\partial \theta_j}.$$

η is the learning rate; the derivative supplies the direction and magnitude of the local change. A modern network has $P = 10^7$ to 10^{11} parameters.

We need all P derivatives $\partial \mathcal{L} / \partial \theta_j$. Backprop computes them together in **one reverse sweep**, without perturbing parameters one at a time.

The route from one local operation to an entire network

V VISUAL

Stage	Question we answer
1 · local rule	How does one operation pass a gradient backward?

The route from one local operation to an entire network

V VISUAL

Stage	Question we answer
1 · local rule	How does one operation pass a gradient backward?
2 · branches	Why do gradient contributions add?

The route from one local operation to an entire network

V VISUAL

Stage	Question we answer
1 · local rule	How does one operation pass a gradient backward?
2 · branches	Why do gradient contributions add?
3 · worked graph	Can we reproduce every derivative numerically?

The route from one local operation to an entire network

V VISUAL

Stage	Question we answer
1 · local rule	How does one operation pass a gradient backward?
2 · branches	Why do gradient contributions add?
3 · worked graph	Can we reproduce every derivative numerically?
4 · layers + MLP	How do scalar rules become vector and matrix formulas?

The route from one local operation to an entire network

V VISUAL

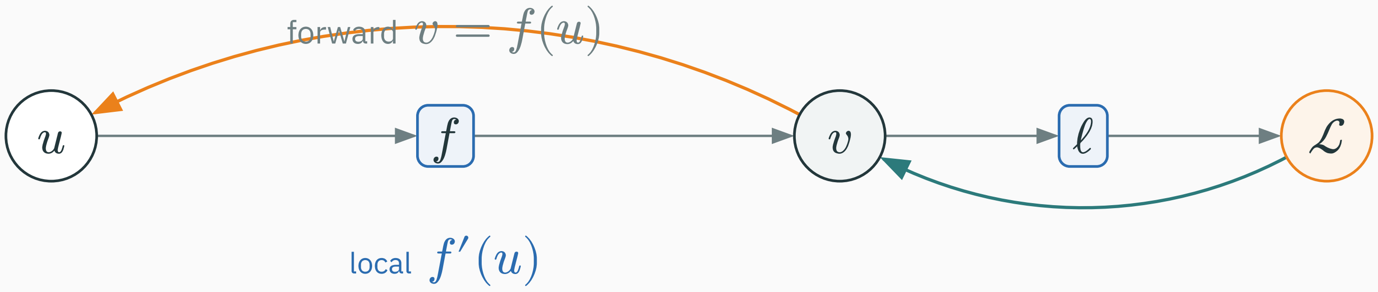
Stage	Question we answer
1 · local rule	How does one operation pass a gradient backward?
2 · branches	Why do gradient contributions add?
3 · worked graph	Can we reproduce every derivative numerically?
4 · layers + MLP	How do scalar rules become vector and matrix formulas?
5 · depth + gradient flow	What happens when many local gradients multiply?

One local operation

Zoom in on one operation. The value circle u enters the boxed operation f; its output is stored in the value circle v.

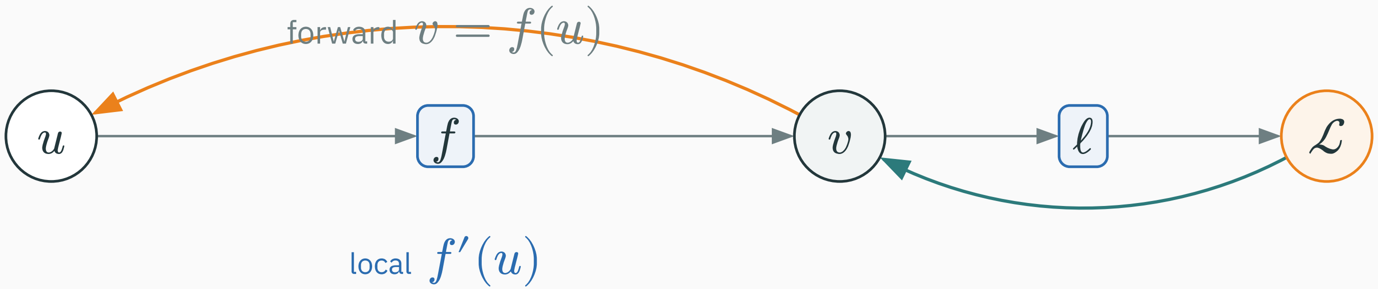
One local operation

Zoom in on one operation. The value circle u enters the boxed operation f; its output is stored in the value circle v.



One local operation

Zoom in on one operation. The value circle u enters the boxed operation f; its output is stored in the value circle v.

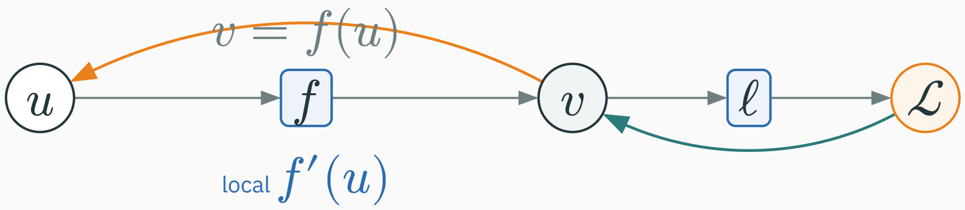


backward computes $\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot f'(u)$

Every operation multiplies the arriving gradient by its local gradient

D DERIVATION

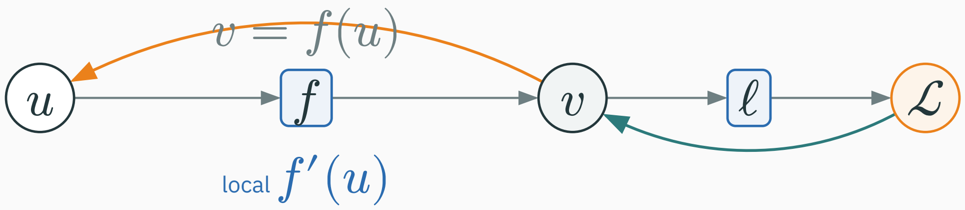
For the operation $v = f(u)$, the chain rule is



Every operation multiplies the arriving gradient by its local gradient

D DERIVATION

For the operation $v = f(u)$, the chain rule is

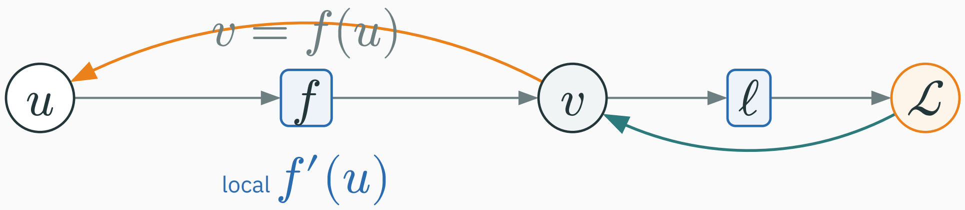


$$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot f'(u)$$

Every operation multiplies the arriving gradient by its local gradient

D DERIVATION

For the operation $v = f(u)$, the chain rule is



$$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot f'(u)$$

Downstream gradient

sent back to u

$\frac{\partial \mathcal{L}}{\partial u}$

Upstream gradient

arriving at v

$\frac{\partial \mathcal{L}}{\partial v}$

Local gradient

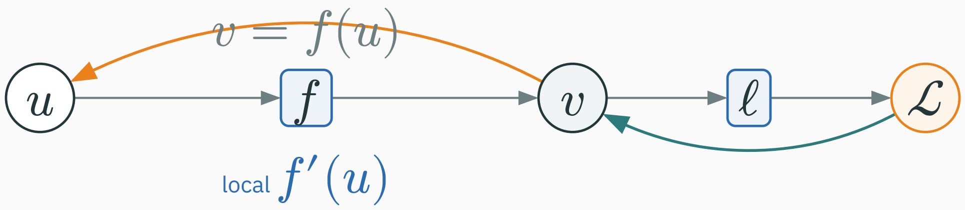
this operation's slope

$f'(u) = \frac{\partial v}{\partial u}$

The chain rule is one reusable local rule

D DERIVATION

For every scalar operation $v = f(u)$:

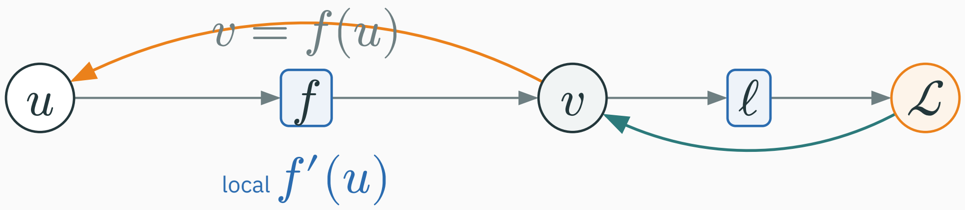


$$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot \frac{\partial v}{\partial u}.$$

The chain rule is one reusable local rule

D DERIVATION

For every scalar operation $v = f(u)$:



$$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot \frac{\partial v}{\partial u}.$$

> The operation multiplies its **upstream gradient** by its **local gradient** to produce the **downstream gradient**. It needs no global formula for the whole network.

Example: a square operation

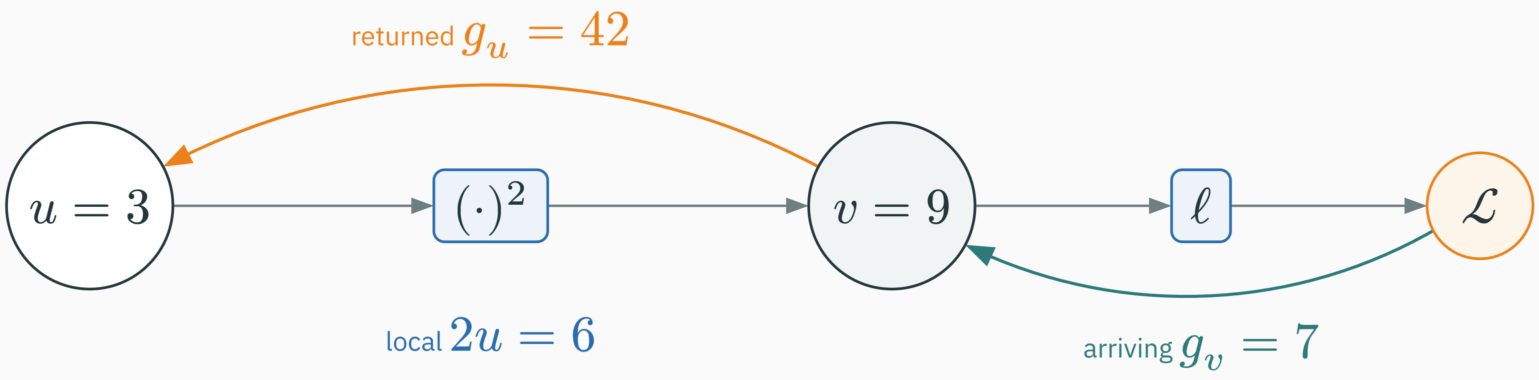
DERIVATION

For $v = u^2$ at $u = 3$, the forward value is $v = 9$ and the local slope is $\partial v/\partial u = 2u = 6$. Suppose the rest of the graph sends $g_v = \partial \mathcal{L}/\partial v = 7$ back to this square.

Example: a square operation

D DERIVATION

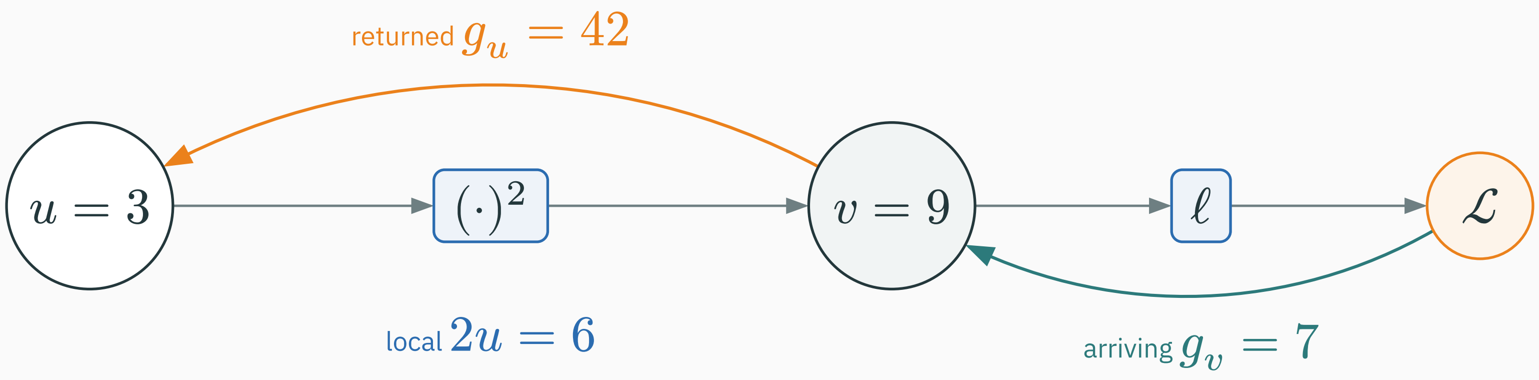
For $v = u^2$ at $u = 3$, the forward value is $v = 9$ and the local slope is $\partial v/\partial u = 2u = 6$. Suppose the rest of the graph sends $g_v = \partial \mathcal{L}/\partial v = 7$ back to this square.



Example: a square operation

D DERIVATION

For $v = u^2$ at $u = 3$, the forward value is $v = 9$ and the local slope is $\partial v/\partial u = 2u = 6$. Suppose the rest of the graph sends $g_v = \partial\mathcal{L}/\partial v = 7$ back to this square.

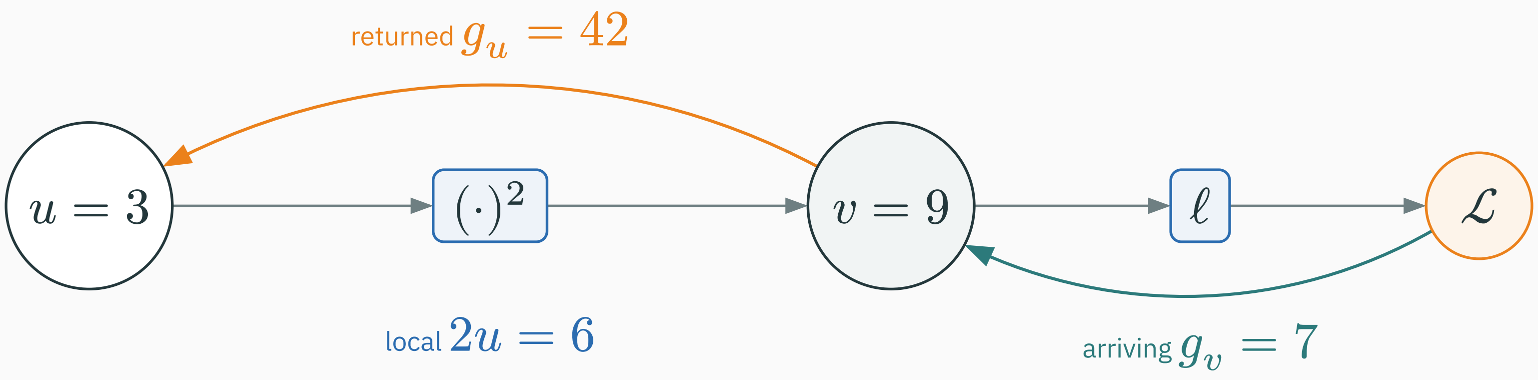


$$\frac{\partial\mathcal{L}}{\partial u} = \frac{\partial\mathcal{L}}{\partial v} \cdot 2u = 7 \cdot 6 = 42.$$

Example: a square operation

D DERIVATION

For $v = u^2$ at $u = 3$, the forward value is $v = 9$ and the local slope is $\partial v/\partial u = 2u = 6$. Suppose the rest of the graph sends $g_v = \partial\mathcal{L}/\partial v = 7$ back to this square.



$$\frac{\partial\mathcal{L}}{\partial u} = \frac{\partial\mathcal{L}}{\partial v} \cdot 2u = 7 \cdot 6 = 42.$$

> The square operation multiplies the arriving $\partial\mathcal{L}/\partial v = 7$ by its local gradient $2u = 6$ to produce $\partial\mathcal{L}/\partial u = 42$.

Addition copies the arriving gradient

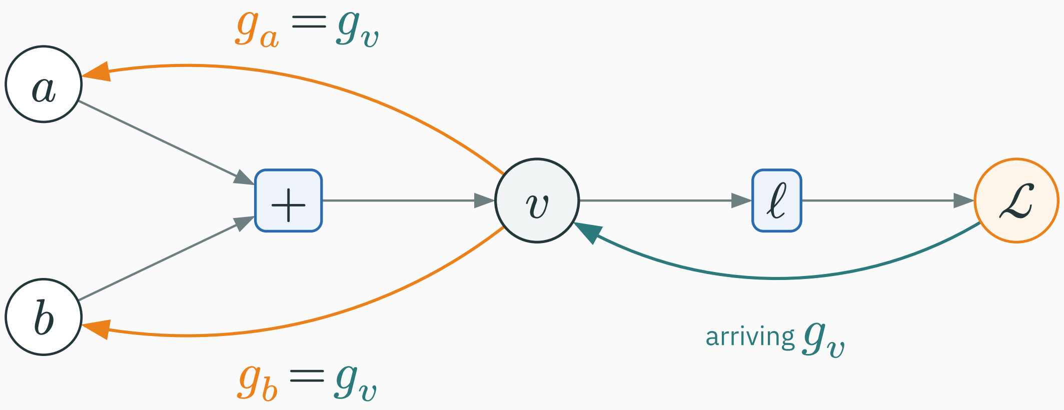
D DERIVATION

Operation $v = a + b$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = 1$, $\partial v/\partial b = 1$.

Addition copies the arriving gradient

D DERIVATION

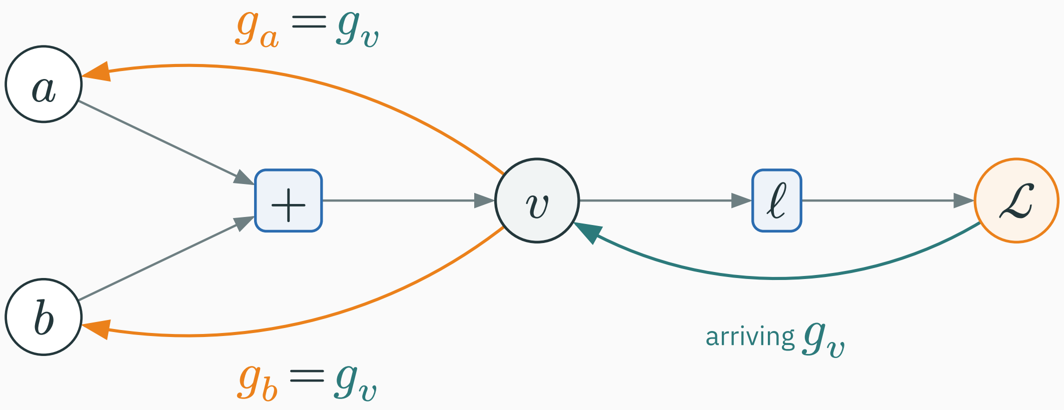
Operation $v = a + b$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = 1$, $\partial v/\partial b = 1$.



Addition copies the arriving gradient

D DERIVATION

Operation $v = a + b$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = 1$, $\partial v/\partial b = 1$.

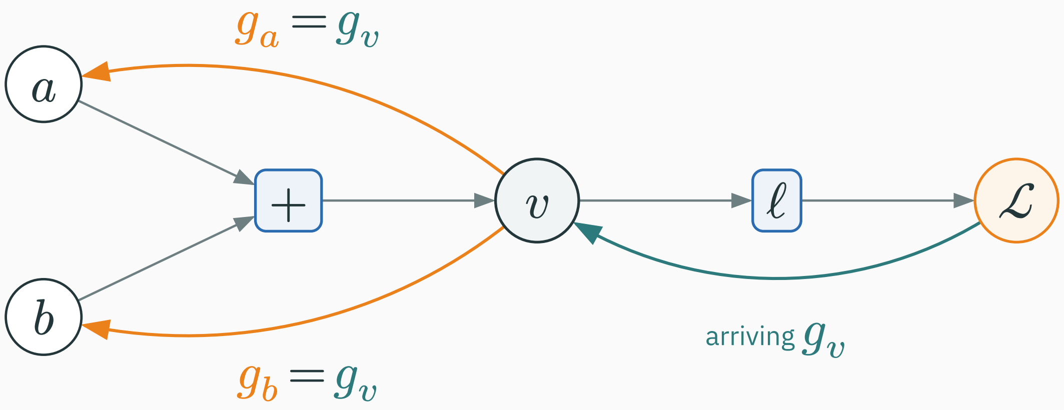


$$\frac{\partial\mathcal{L}}{\partial a} = \frac{\partial\mathcal{L}}{\partial v}\cdot 1, \qquad \frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial v}\cdot 1.$$

Addition copies the arriving gradient

D DERIVATION

Operation $v = a + b$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = 1$, $\partial v/\partial b = 1$.



$$\frac{\partial\mathcal{L}}{\partial a} = \frac{\partial\mathcal{L}}{\partial v}\cdot 1, \qquad \frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial v}\cdot 1.$$

$+$ copies the arriving derivative into a downstream derivative for every input.

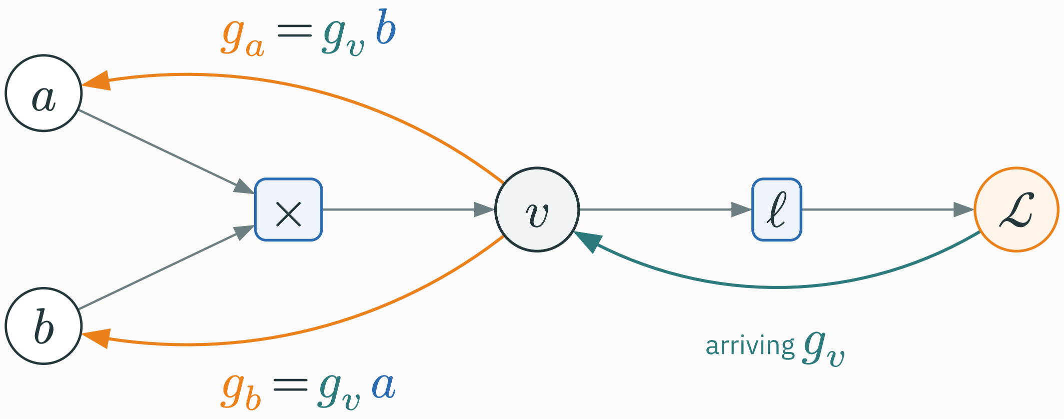
Multiplication scales by the other input

D DERIVATION

Operation $v = ab$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = b$, $\partial v/\partial b = a$.

Multiplication scales by the other input

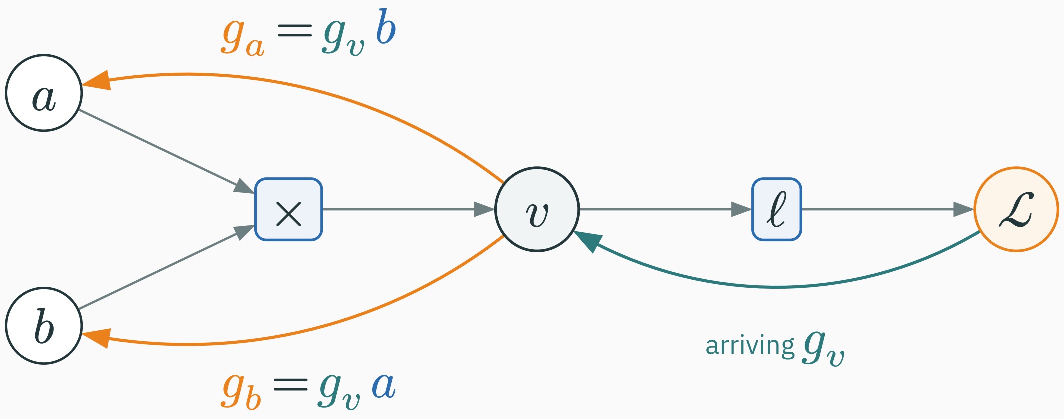
Operation $v = ab$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = b$, $\partial v/\partial b = a$.



Multiplication scales by the other input

D DERIVATION

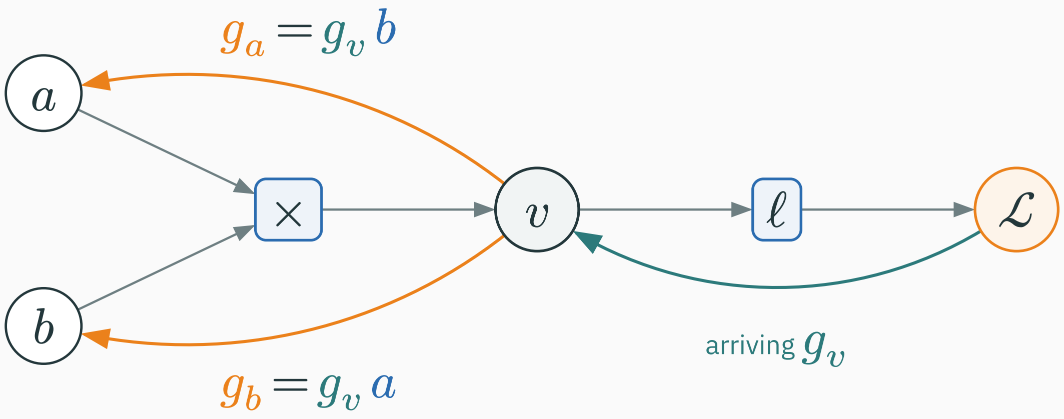
Operation $v = ab$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = b$, $\partial v/\partial b = a$.



$$\frac{\partial\mathcal{L}}{\partial a} = \frac{\partial\mathcal{L}}{\partial v}\cdot b, \qquad \frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial v}\cdot a.$$

Multiplication scales by the other input

Operation $v = ab$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = b$, $\partial v/\partial b = a$.



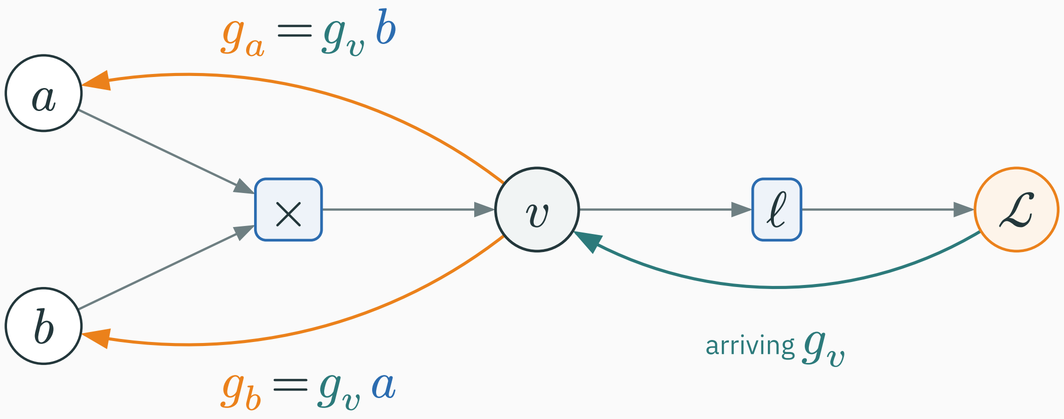
$$\frac{\partial\mathcal{L}}{\partial a} = \frac{\partial\mathcal{L}}{\partial v} \cdot b, \qquad \frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial v} \cdot a.$$

$\times$ **uses the other input as its local multiplier.**

Multiplication scales by the other input

D DERIVATION

Operation $v = ab$. Let $g_v = \partial\mathcal{L}/\partial v$ arrive. Local derivatives: $\partial v/\partial a = b$, $\partial v/\partial b = a$.



$$\frac{\partial\mathcal{L}}{\partial a} = \frac{\partial\mathcal{L}}{\partial v}\cdot b, \qquad \frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial v}\cdot a.$$

$\times$ uses the other input as its local multiplier.

If $a = 2$, $b = 5$, and $\partial\mathcal{L}/\partial v = 3$, then $\partial\mathcal{L}/\partial a = 15$ and $\partial\mathcal{L}/\partial b = 6$.

Activation operation

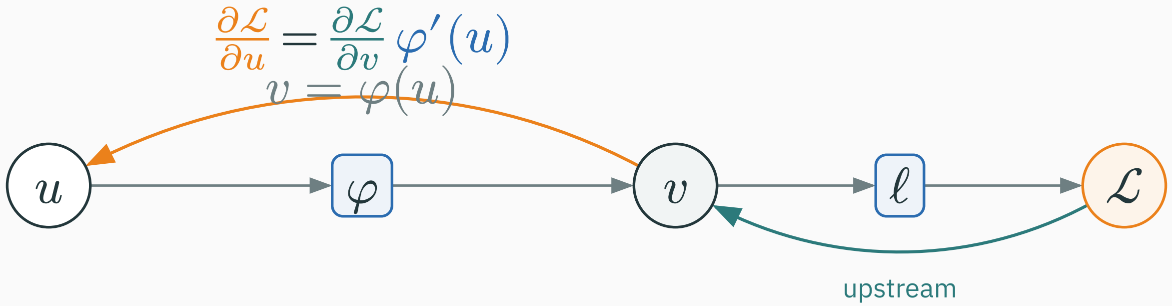
D DERIVATION

Operation $v = \varphi(u)$ (elementwise nonlinearity). Local derivative: $\partial v / \partial u = \varphi'(u)$.

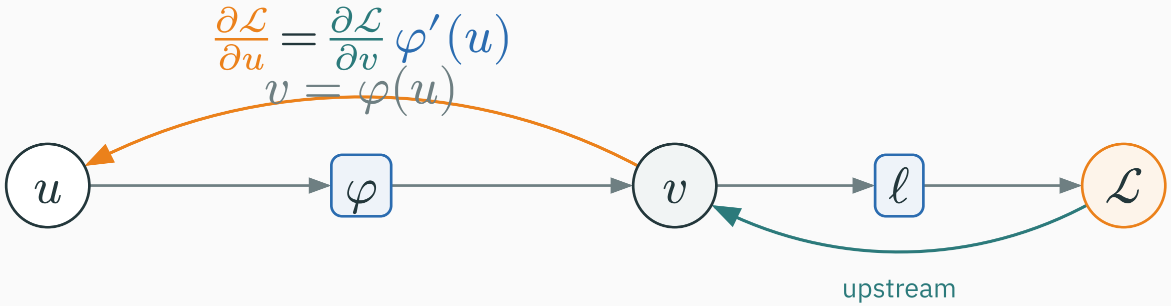
Activation operation

DERIVATION

Operation $v = \varphi(u)$ (elementwise nonlinearity). Local derivative: $\partial v/\partial u = \varphi'(u)$.



Operation $v = \varphi(u)$ (elementwise nonlinearity). Local derivative: $\partial v/\partial u = \varphi'(u)$.

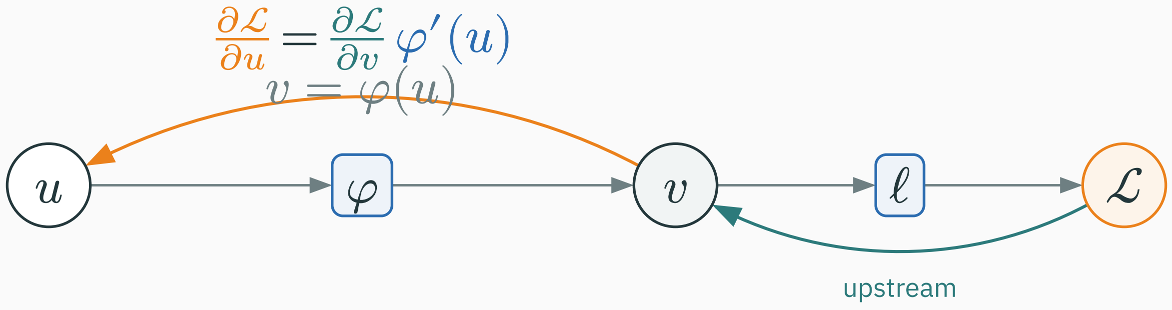


$$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot \varphi'(u).$$

Activation operation

D DERIVATION

Operation $v = \varphi(u)$ (elementwise nonlinearity). Local derivative: $\partial v / \partial u = \varphi'(u)$.



$$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot \varphi'(u).$$

sigmoid σ: $\varphi'(u) = \sigma(u)(1 - \sigma(u))$

ReLU: $\varphi'(u) = 1$ for $u > 0$, and 0 for $u < 0$

Pop quiz: what crosses an inactive ReLU?

Q QUESTION

Let $v = \mathrm{ReLU}(u)$. If $u = -2$ and the upstream gradient is $\partial\mathcal{L}/\partial v = 5$, what downstream gradient $\partial\mathcal{L}/\partial u$ is sent back?

A. 5

B. -10

C. 0

D. undefined because $u < 0$

Answer: an inactive ReLU blocks the gradient

A ANSWER

Correct: C — 0

Why: For $u < 0$, the local gradient is $\partial v / \partial u = 0$. Therefore downstream $=$ upstream $\times$ local $= 5 \times 0 = 0$.

Answer: an inactive ReLU blocks the gradient

A ANSWER

Correct: C — 0

Why: For $u < 0$, the local gradient is $\partial v/\partial u = 0$. Therefore downstream $=$ upstream $\times$ local $= 5 \times 0 = 0$.

Forward: $u = -2 \to v = 0$

Backward: $5 \times 0 \to \partial\mathcal{L}/\partial u = 0$

Answer: an inactive ReLU blocks the gradient

A ANSWER

Correct: C — 0

Why: For $u < 0$, the local gradient is $\partial v / \partial u = 0$. Therefore downstream $=$ upstream $\times$ local $= 5 \times 0 = 0$.

Forward: $u = -2 \to v = 0$

Backward: $5 \times 0 \to \partial \mathcal{L} / \partial u = 0$

At exactly $u = 0$, ReLU's ordinary derivative is undefined. Autodiff software must choose a convention; commonly it returns 0.

The rule table — memorize this

VISUAL

Forward operation	Contribution sent backward
$v = a + b$	$\frac{\partial \mathcal{L}}{\partial a}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot 1$ $\frac{\partial \mathcal{L}}{\partial b}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot 1$
$v = ab$	$\frac{\partial \mathcal{L}}{\partial a}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot b$ $\frac{\partial \mathcal{L}}{\partial b}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot a$
$v = u^2$	$\frac{\partial \mathcal{L}}{\partial u}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot 2u$
$v = \varphi(u)$	$\frac{\partial \mathcal{L}}{\partial u}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot \varphi'(u)$

The rule table — memorize this

V VISUAL

Forward operation	Contribution sent backward
$v = a + b$	$\frac{\partial \mathcal{L}}{\partial a}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot 1$ $\frac{\partial \mathcal{L}}{\partial b}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot 1$
$v = ab$	$\frac{\partial \mathcal{L}}{\partial a}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot b$ $\frac{\partial \mathcal{L}}{\partial b}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot a$
$v = u^2$	$\frac{\partial \mathcal{L}}{\partial u}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot 2u$
$v = \varphi(u)$	$\frac{\partial \mathcal{L}}{\partial u}$ += $\frac{\partial \mathcal{L}}{\partial v} \cdot \varphi'(u)$

Use `+=`, not `=` — a value consumed by several operations accumulates gradient from **all** of them.

Gradients accumulate at branches

Why gradients add at a branch

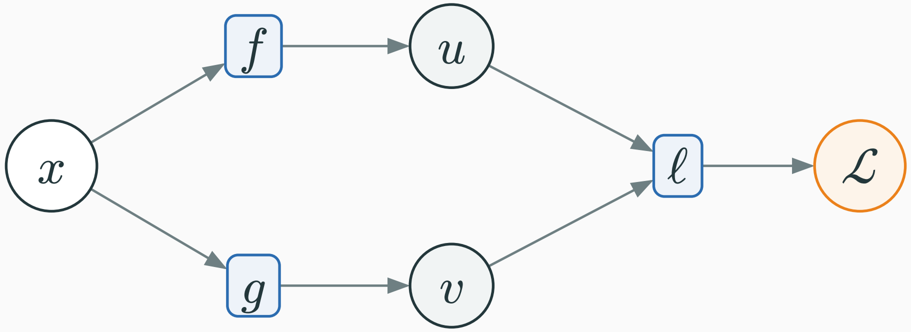
V VISUAL

Imagine nudging x from 4 to 4.01. The **same nudge** travels down both routes to the loss:

Why gradients add at a branch

V VISUAL

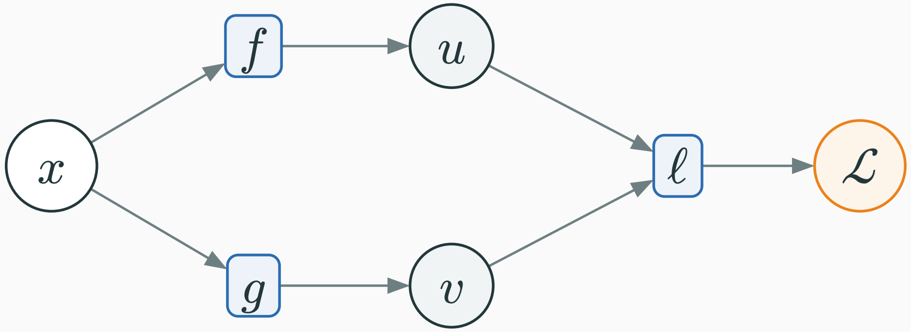
Imagine nudging x from 4 to 4.01. The **same nudge** travels down both routes to the loss:



Why gradients add at a branch

V VISUAL

Imagine nudging x from 4 to 4.01. The **same nudge** travels down both routes to the loss:



Via $u = x^2$: the local slope is $2x = 8$, so $\Delta x = 0.01$ changes the loss by about $8 \cdot 0.01 = 0.08$.

Via $v = 3x$: the local slope is 3, so the same nudge changes the loss by about $3 \cdot 0.01 = 0.03$.

$$\textbf{total slope} = \textbf{total loss change} \div \textbf{nudge} = \frac{0.08+0.03}{0.01} = 8 + 3 = 11$$

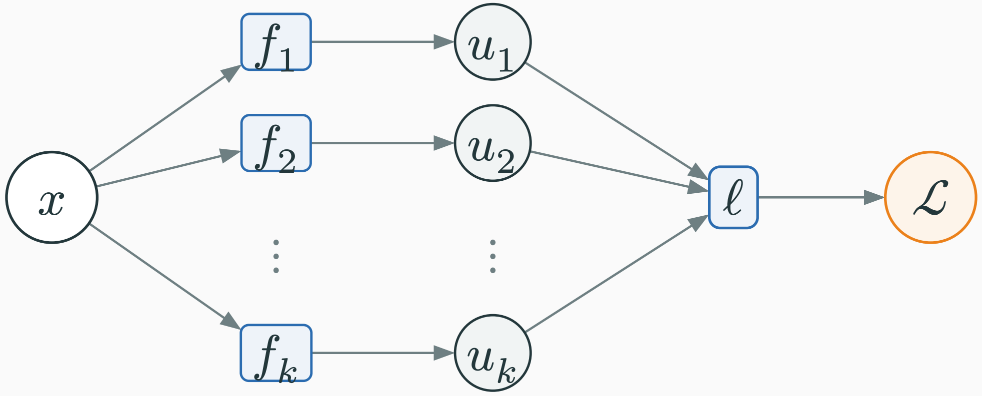
The multivariable chain rule adds every path

D DERIVATION

The nudge experiment generalizes. If x influences $\mathcal{L}$ through $u_1, \ldots, u_k$:

The multivariable chain rule adds every path

The nudge experiment generalizes. If x influences $\mathcal{L}$ through $u_1, \ldots, u_k$:

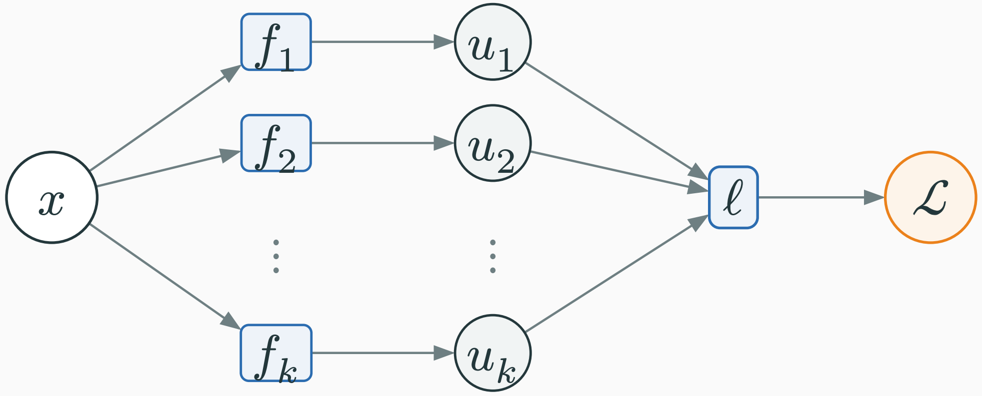


branch i contributes upstream $\partial\mathcal{L}/\partial u_i$ × local $\partial u_i/\partial x$

The multivariable chain rule adds every path

D DERIVATION

The nudge experiment generalizes. If x influences $\mathcal{L}$ through $u_1, \ldots, u_k$:



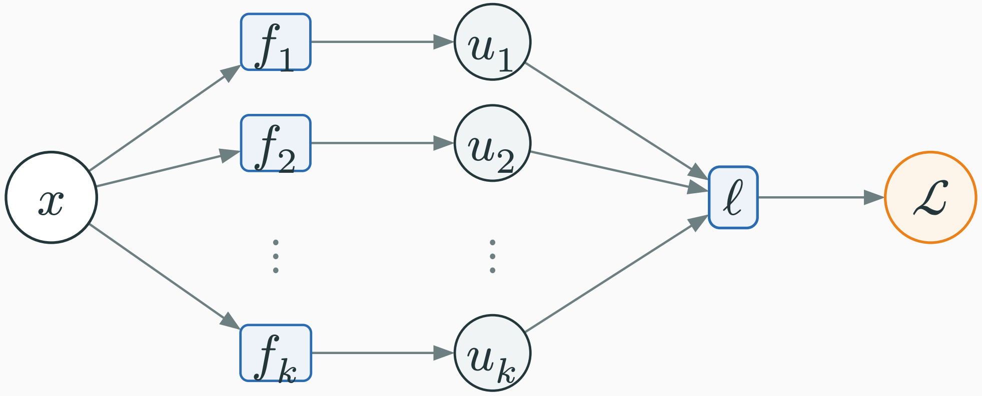
branch i contributes upstream $\partial\mathcal{L}/\partial u_i$ $\times$ local $\partial u_i/\partial x$

$$\frac{\partial \mathcal{L}}{\partial x} = \sum_{i=1}^{k} \frac{\partial \mathcal{L}}{\partial u_i} \frac{\partial u_i}{\partial x}.$$

The multivariable chain rule adds every path

D DERIVATION

The nudge experiment generalizes. If x influences $\mathcal{L}$ through $u_1, \ldots, u_k$:



branch i contributes upstream $\partial\mathcal{L}/\partial u_i$ × local $\partial u_i/\partial x$

$$\frac{\partial \mathcal{L}}{\partial x} = \sum_{i=1}^{k} \frac{\partial \mathcal{L}}{\partial u_i} \frac{\partial u_i}{\partial x}.$$

each path contributes one chain-rule product; the products add

Worked branch example

Let $u = x^2, \;\; v = 3x, \;\; \mathcal{L} = u + v = x^2 + 3x$.

Worked branch example

Let $u = x^2,\ \ v = 3x,\ \ \mathcal{L} = u + v = x^2 + 3x$. By symbolic differentiation — derive the formula first:

$$\frac{\partial \mathcal{L}}{\partial x} = 2x + 3, \quad \text{at } x = 4: \quad 2(4) + 3 = 11.$$

Worked branch example

Let $u = x^2, \;\; v = 3x, \;\; \mathcal{L} = u + v = x^2 + 3x$. By symbolic differentiation — derive the formula first:

$$\frac{\partial \mathcal{L}}{\partial x} = 2x + 3, \quad \text{at } x = 4 : \quad 2(4) + 3 = 11.$$

Now let us get the **same** answer by backprop.

Same answer, by backprop

D DERIVATION

Forward at $x = 4$: $u = 16, \; v = 12, \; \mathcal{L} = 28.$

Same answer, by backprop

D DERIVATION

Forward at $x = 4$: $\;u = 16,\; v = 12,\; \mathcal{L} = 28$. **Backward:** seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$; the add operation gives $\partial\mathcal{L}/\partial u = 1$ and $\partial\mathcal{L}/\partial v = 1$.

Same answer, by backprop

D DERIVATION

Forward at $x = 4$: $u = 16,\ v = 12,\ \mathcal{L} = 28$. **Backward:** seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$; the add operation gives $\partial\mathcal{L}/\partial u = 1$ and $\partial\mathcal{L}/\partial v = 1$.

$$\left(\frac{\partial\mathcal{L}}{\partial x}\right)_{\text{via } u} = 1 \cdot 2x = 8, \qquad \left(\frac{\partial\mathcal{L}}{\partial x}\right)_{\text{via } v} = 1 \cdot 3 = 3.$$

Same answer, by backprop

D DERIVATION

Forward at $x = 4$: $u = 16,\ v = 12,\ \mathcal{L} = 28$. **Backward:** seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$; the add operation gives $\partial\mathcal{L}/\partial u = 1$ and $\partial\mathcal{L}/\partial v = 1$.

$$\left(\frac{\partial\mathcal{L}}{\partial x}\right)_{\text{via } u} = 1 \cdot 2x = 8, \qquad \left(\frac{\partial\mathcal{L}}{\partial x}\right)_{\text{via } v} = 1 \cdot 3 = 3.$$

$$\frac{\partial\mathcal{L}}{\partial x} = 8 + 3 = 11. \quad \checkmark$$

PyTorch adds the same two paths

INTERACTIVE

```python
import torch

x = torch.tensor(4.0, requires_grad=True)
u = x**2                        # path 1
v = 3*x                         # path 2
L = u + v
L.backward()

print(x.grad)                   # tensor(11.)
```

PyTorch adds the same two paths

I INTERACTIVE

```python
import torch

x = torch.tensor(4.0, requires_grad=True)
u = x**2                        # path 1
v = 3*x                         # path 2
L = u + v
L.backward()

print(x.grad)                   # tensor(11.)
```

via u: $\partial\frac{u}{\partial}x = 2x = 8$

via v: $\partial\frac{v}{\partial}x = 3$

PyTorch adds the same two paths

INTERACTIVE

```
import torch

x = torch.tensor(4.0, requires_grad=True)
u = x**2                    # path 1
v = 3*x                     # path 2
L = u + v
L.backward()

print(x.grad)               # tensor(11.)
```

via u: $\partial\frac{u}{\partial}x = 2x = 8$

via v: $\partial\frac{v}{\partial}x = 3$

autograd accumulates both downstream contributions: $8 + 3 = 11$

The central worked example

A scalar linear model

D DERIVATION

Predict $\hat{y} = wx + b$, squared-error loss $\mathcal{L} = (\hat{y} - y)^2$.

A scalar linear model

D DERIVATION

Predict $\hat{y} = wx + b$, squared-error loss $\mathcal{L} = (\hat{y} - y)^2$. Concrete numbers:

$$x = 3, \quad w = 2, \quad b = 1, \quad y = 10.$$

A scalar linear model

D DERIVATION

Predict $\hat{y} = wx + b$, squared-error loss $\mathcal{L} = (\hat{y} - y)^2$. Concrete numbers:

$$x = 3, \quad w = 2, \quad b = 1, \quad y = 10.$$

> Compute $\partial\mathcal{L}/\partial w$, $\partial\mathcal{L}/\partial x$, $\partial\mathcal{L}/\partial b$, and $\partial\mathcal{L}/\partial y$ by backpropagation; verify the four values with PyTorch.

Break into primitives

Decompose the loss into elementary operations and name each stored result:

$$m = wx, \quad a = m + b, \quad e = a - y, \quad \mathcal{L} = e^2.$$

Break into primitives

Decompose the loss into elementary operations and name each stored result:

$$m = wx, \quad a = m + b, \quad e = a - y, \quad \mathcal{L} = e^2.$$

value m **after** $\times$ **· value** a **after** $+$ **· value** e **after** $-$ **· value** $\mathcal{L}$ **after** $(\cdot)^2$

Break into primitives

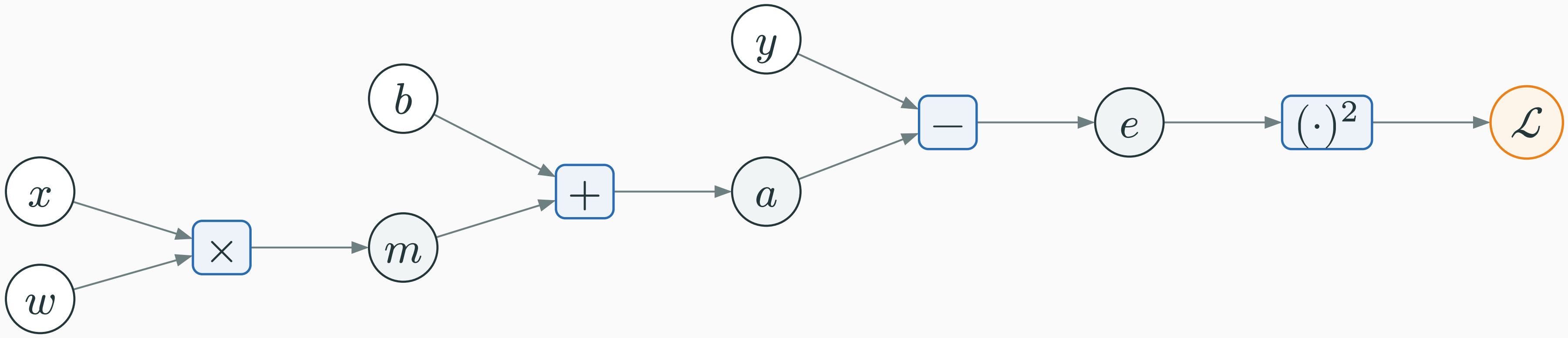
Decompose the loss into elementary operations and name each stored result:

$$m = wx, \quad a = m + b, \quad e = a - y, \quad \mathcal{L} = e^2.$$

value m **after** $\times$ · **value** a **after** $+$ · **value** e **after** $-$ · **value** $\mathcal{L}$ **after** $(\cdot)^2$

The symbols m, a, and e name stored intermediate values. They are circles in the graph; the four operations are boxes.

The computation graph



Given values w, x, b, y	**Stored values** m, a, e
Operations $\times, +, -, (\cdot)^2$	**Scalar loss** $\mathcal{L}$

Forward pass

D DERIVATION

Stored value	Operation that produces it	Numeric value
m	$w \times x$	$2 \cdot 3 = 6$
a	$m + b$	$6 + 1 = 7$
e	$a - y$	$7 - 10 = -3$
$\mathcal{L}$	e^2	$(-3)^2 = 9$

Forward pass

D DERIVATION

Stored value	Operation that produces it	Numeric value
m	$w \times x$	$2 \cdot 3 = 6$
a	$m + b$	$6 + 1 = 7$
e	$a - y$	$7 - 10 = -3$
$\mathcal{L}$	e^2	$(-3)^2 = 9$

$\mathcal{L} = 9$ **— now run the graph backward**

Backward: seed and the square operation

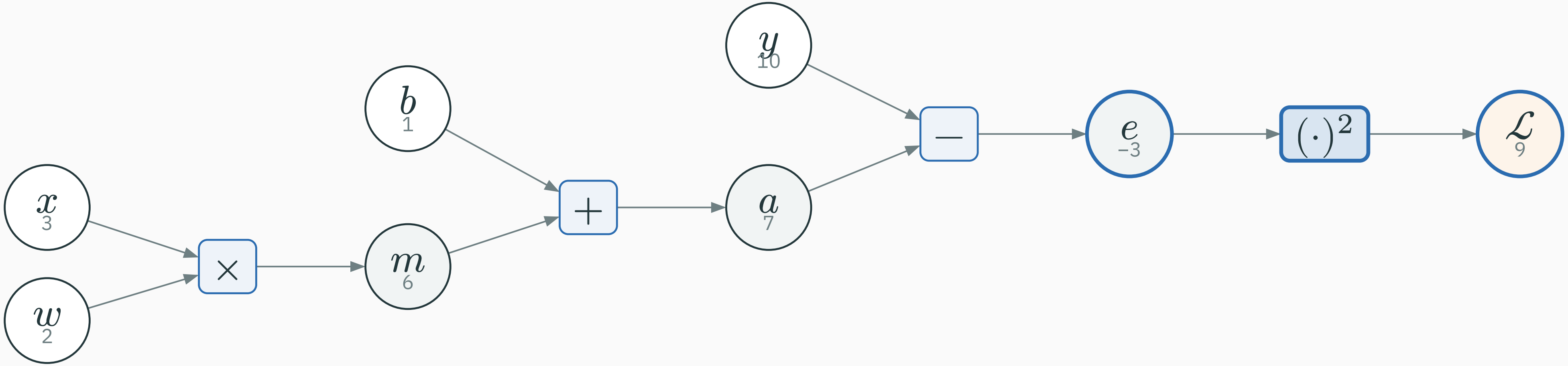
D DERIVATION

Seed the output: $\partial\mathcal{L}/\partial\mathcal{L} = 1$.

Backward: seed and the square operation

D DERIVATION

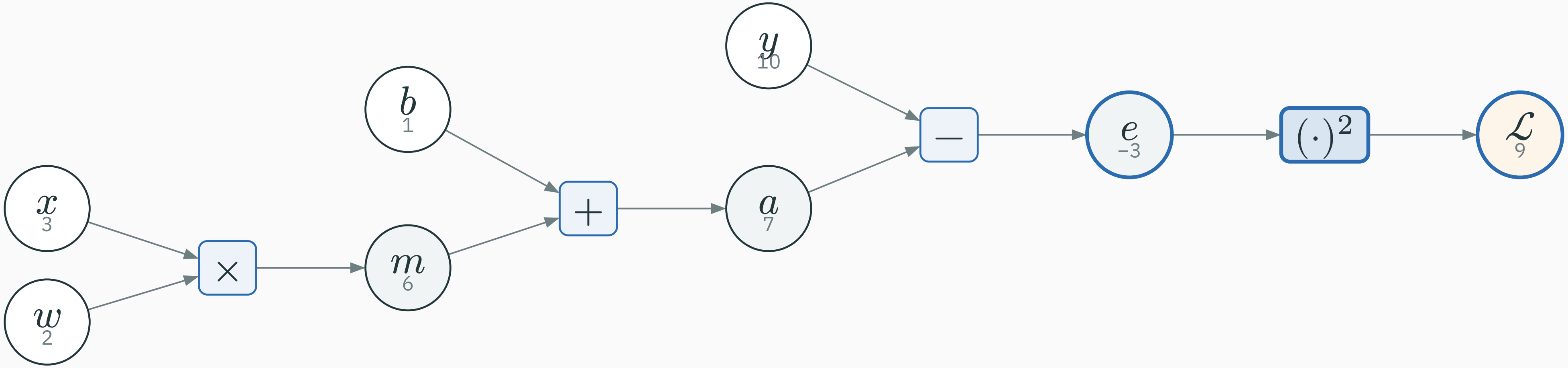
Seed the output: $\partial\mathcal{L}/\partial\mathcal{L} = 1$.



Backward: seed and the square operation

D DERIVATION

Seed the output: $\partial\mathcal{L}/\partial\mathcal{L} = 1$.



Square operation $\mathcal{L} = e^2$, local $\partial\mathcal{L}/\partial e = 2e$:

$$\frac{\partial\mathcal{L}}{\partial e} = 1 \cdot 2e = 1 \cdot 2(-3) = -6.$$

Backward: the subtraction operation

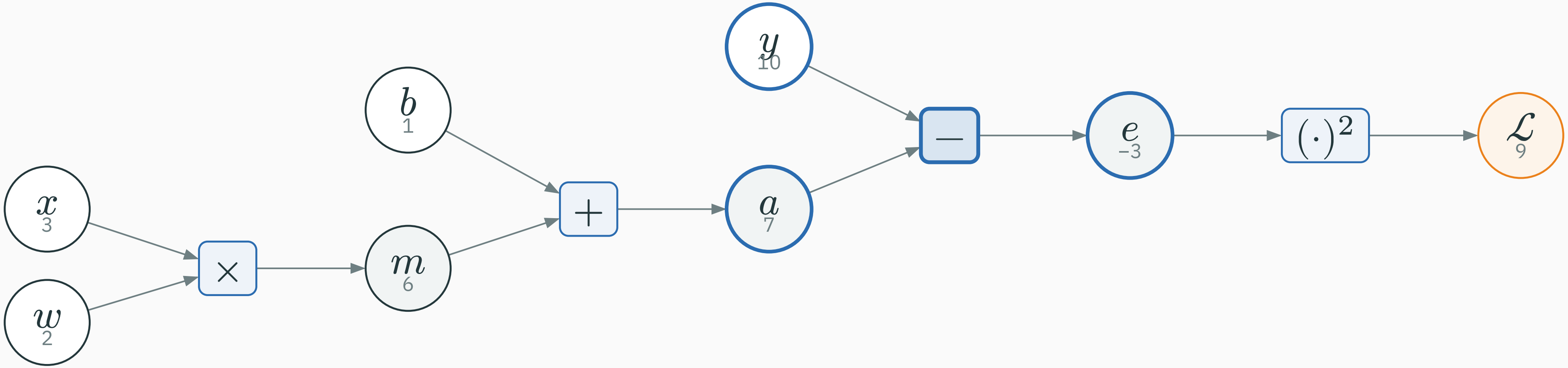
D DERIVATION

Subtraction operation $e = a - y$, locals $\partial e/\partial a = 1$, $\partial e/\partial y = -1$:

Backward: the subtraction operation

D DERIVATION

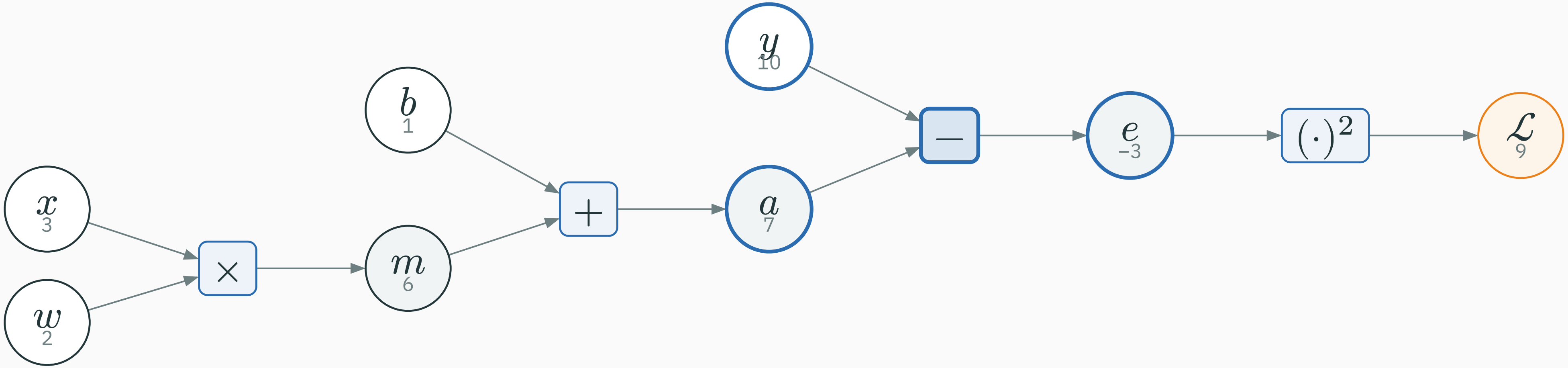
Subtraction operation $e = a - y$, locals $\partial e/\partial a = 1$, $\partial e/\partial y = -1$:



Backward: the subtraction operation

D DERIVATION

Subtraction operation $e = a - y$, locals $\partial e / \partial a = 1$, $\partial e / \partial y = -1$:

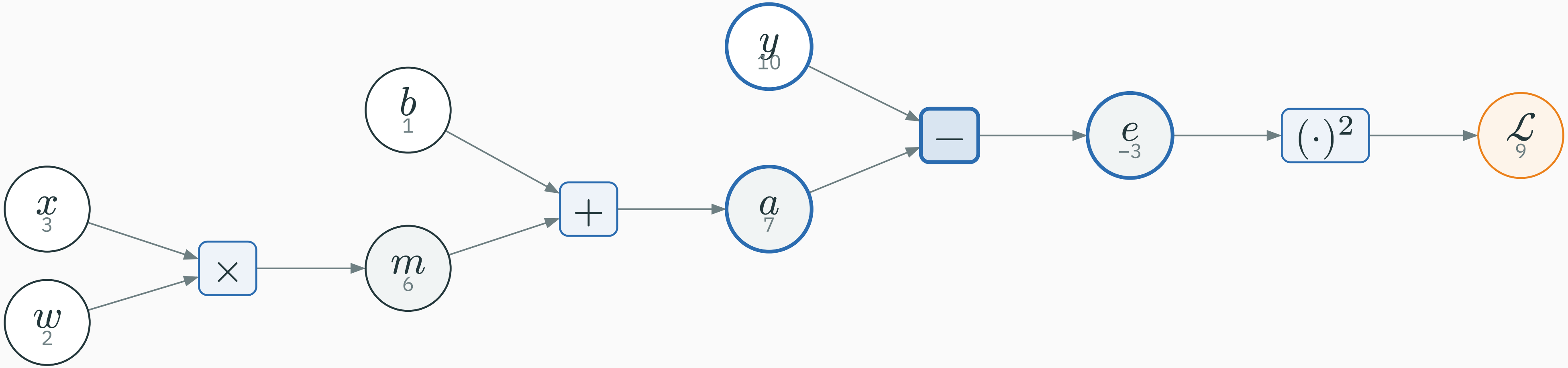


$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial e} \cdot 1 = -6$$

Backward: the subtraction operation

D DERIVATION

Subtraction operation $e = a - y$, locals $\partial e/\partial a = 1$, $\partial e/\partial y = -1$:

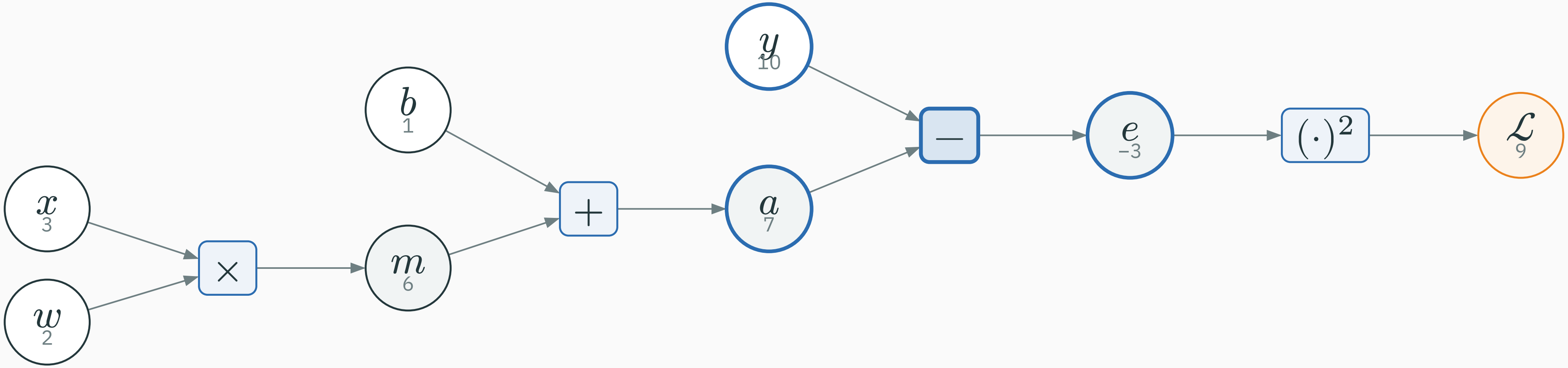


$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial e} \cdot 1 = -6 \qquad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial e} \cdot (-1) = 6$$

Backward: the subtraction operation

D DERIVATION

Subtraction operation $e = a - y$, locals $\partial e/\partial a = 1$, $\partial e/\partial y = -1$:



$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial e} \cdot 1 = -6 \qquad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial e} \cdot (-1) = 6$$

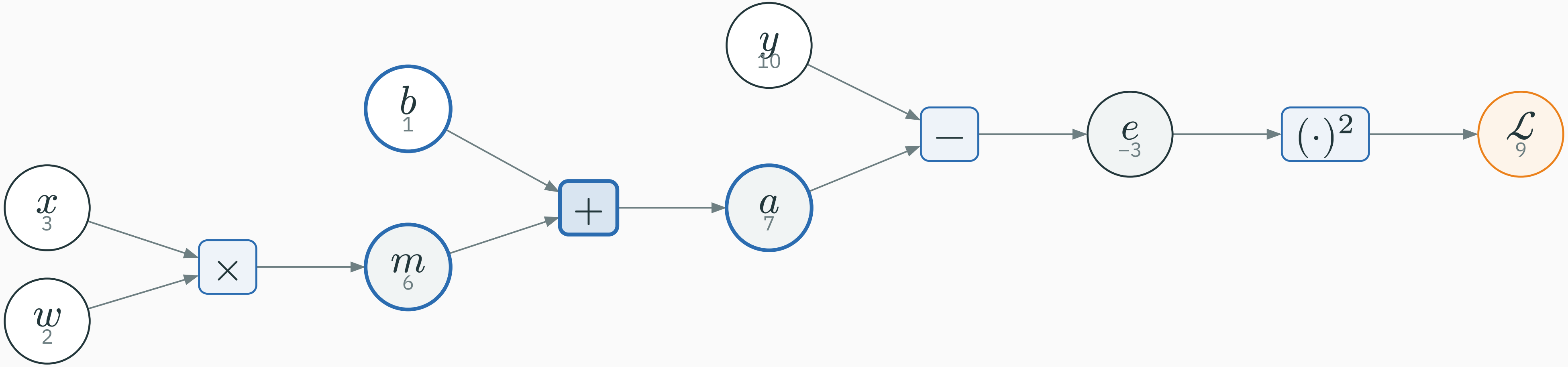
> The target y also receives a derivative. We would use $\partial \mathcal{L}/\partial y$ if y were itself a learned quantity.

Backward: the addition operation

D DERIVATION

Addition operation $a = m + b$ **copies** its gradient:

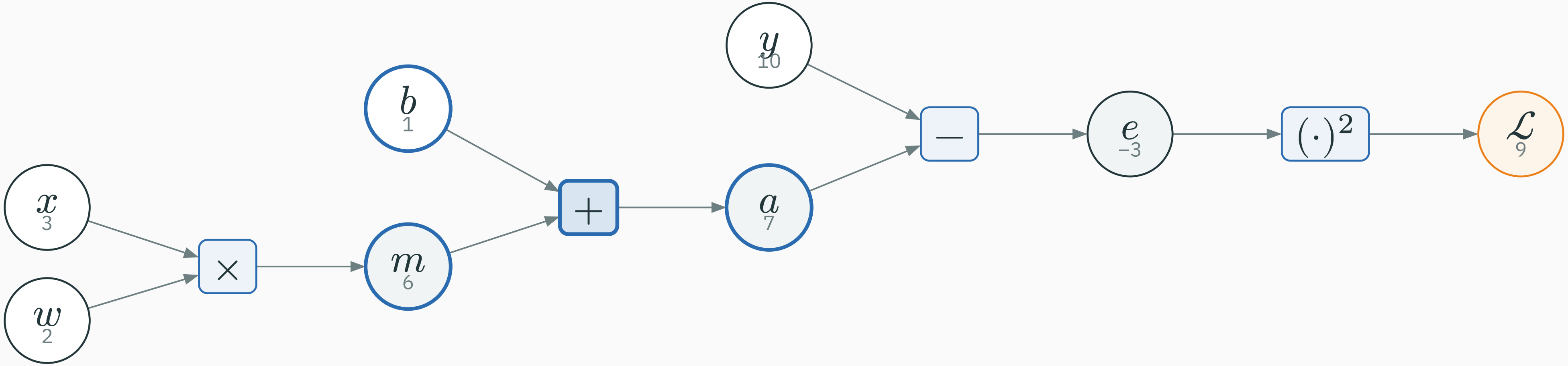
Addition operation $a = m + b$ **copies** its gradient:



Backward: the addition operation

D DERIVATION

Addition operation $a = m + b$ **copies** its gradient:

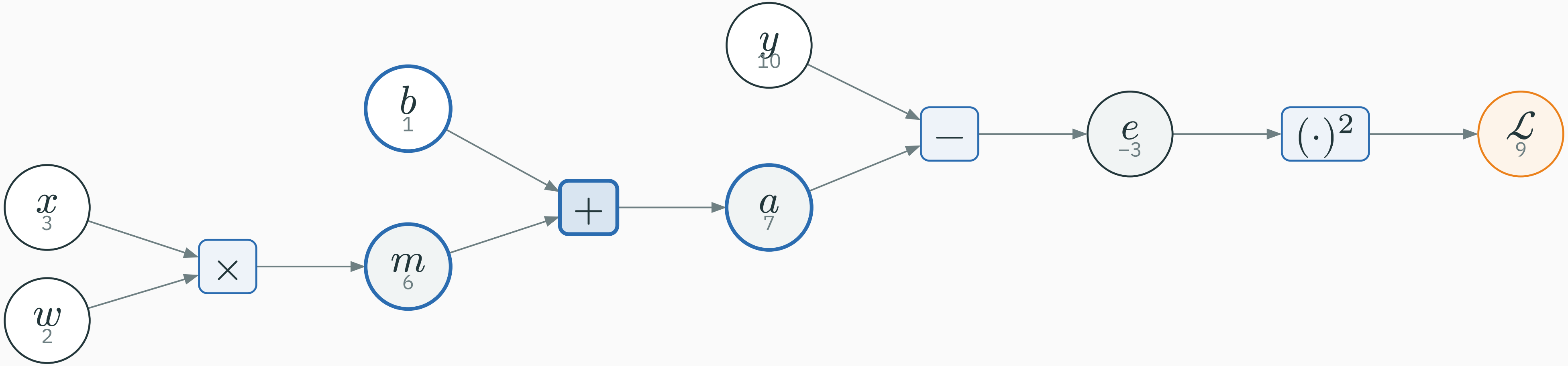


$$\frac{\partial \mathcal{L}}{\partial m} = \frac{\partial \mathcal{L}}{\partial a} = -6, \quad \frac{\partial \mathcal{L}}{\partial b} = -6.$$

Backward: the addition operation

D DERIVATION

Addition operation $a = m + b$ **copies** its gradient:



$$\frac{\partial \mathcal{L}}{\partial m} = \frac{\partial \mathcal{L}}{\partial a} = -6, \quad \frac{\partial \mathcal{L}}{\partial b} = -6.$$

So $\partial \mathcal{L}/\partial b = -6$ is one of our final answers.

Backward: the multiplication operation

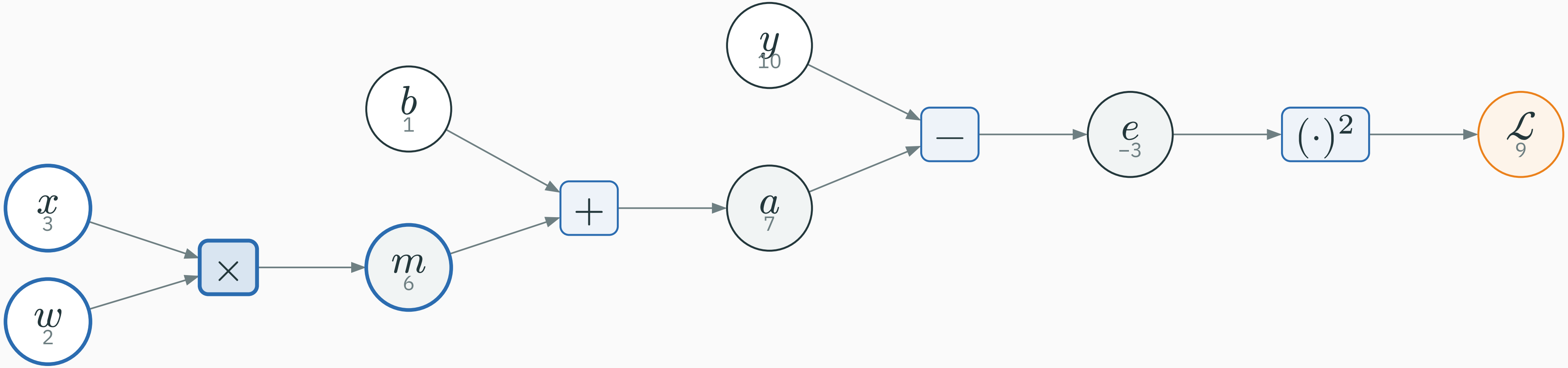
D DERIVATION

Multiplication operation $m = wx$ **sends the other input**:

Backward: the multiplication operation

D DERIVATION

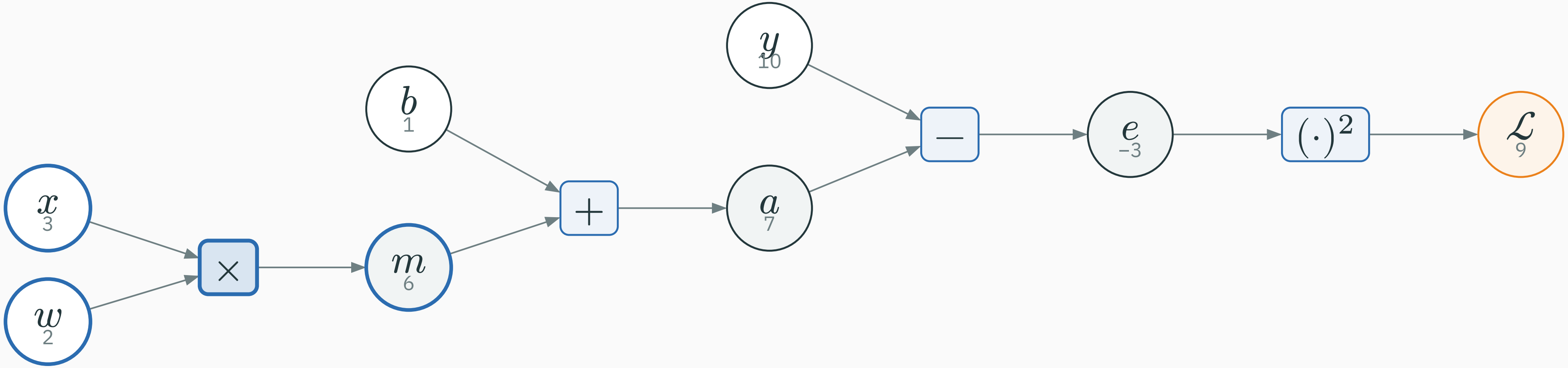
Multiplication operation $m = wx$ **sends the other input**:



Backward: the multiplication operation

D DERIVATION

Multiplication operation $m = wx$ **sends the other input**:

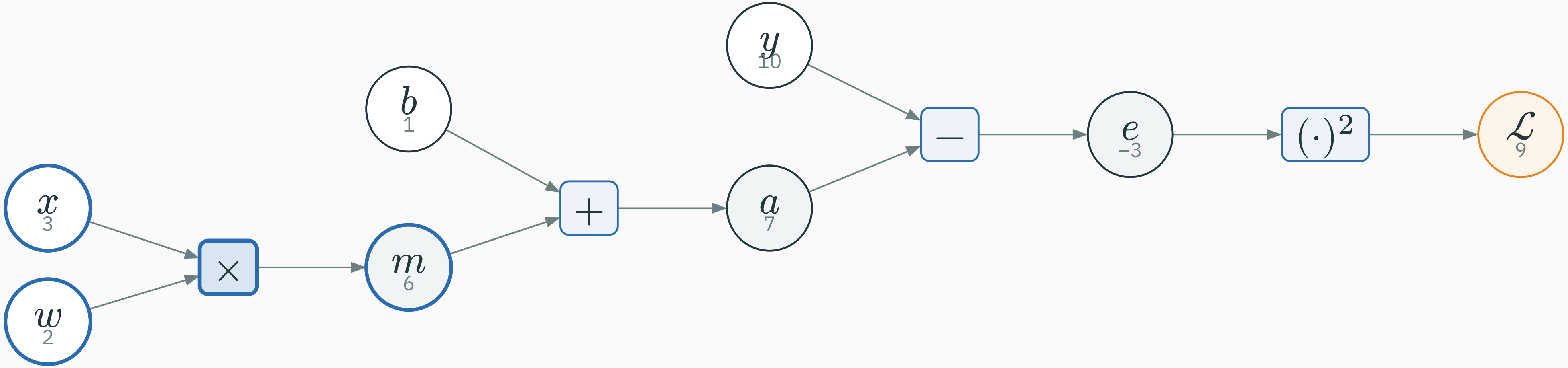


$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial m} \cdot x = -6 \cdot 3 = -18$$

Backward: the multiplication operation

D DERIVATION

Multiplication operation $m = wx$ **sends the other input**:

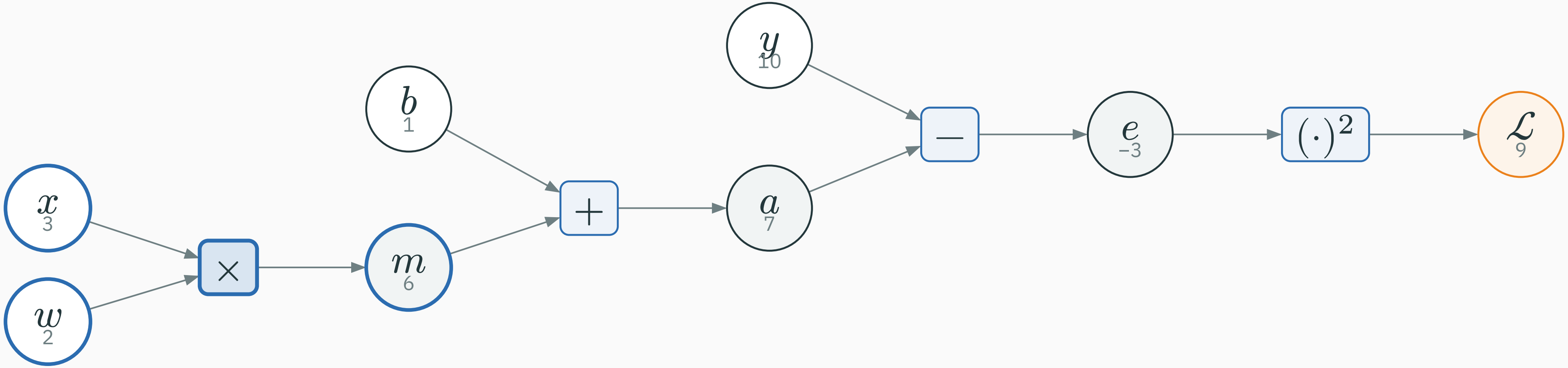


$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial m} \cdot x = -6 \cdot 3 = -18 \qquad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial m} \cdot w = -6 \cdot 2 = -12$$

Backward: the multiplication operation

D DERIVATION

Multiplication operation $m = wx$ **sends the other input**:



$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial m} \cdot x = -6 \cdot 3 = -18 \qquad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial m} \cdot w = -6 \cdot 2 = -12$$

$$\partial \mathcal{L}/\partial w = -18, \quad \partial \mathcal{L}/\partial x = -12$$

Final gradients

V VISUAL

$\partial\mathcal{L}/\partial w$	$\partial\mathcal{L}/\partial b$	$\partial\mathcal{L}/\partial x$	$\partial\mathcal{L}/\partial y$
-18	-6	-12	6

Final gradients

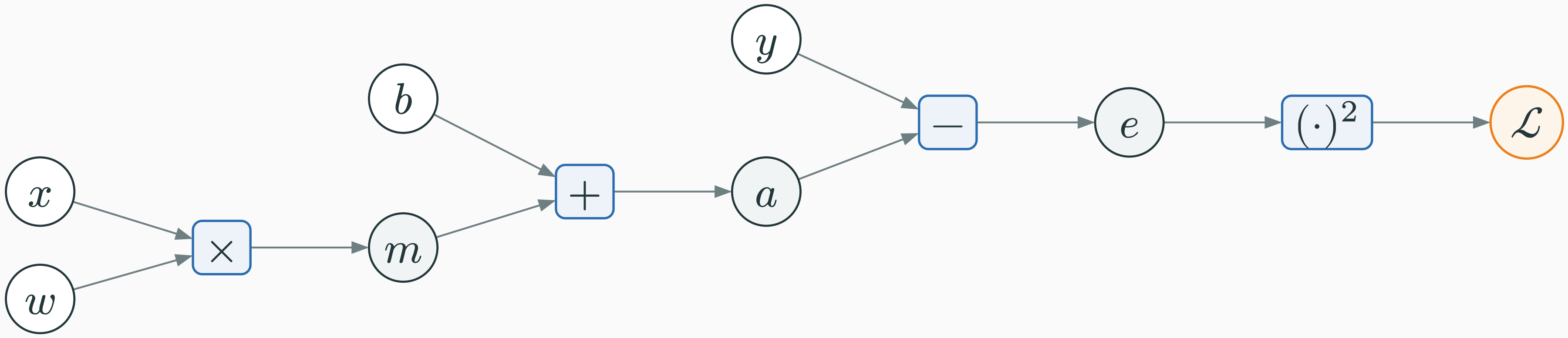
V VISUAL

$\partial \mathcal{L}/\partial w$	$\partial \mathcal{L}/\partial b$	$\partial \mathcal{L}/\partial x$	$\partial \mathcal{L}/\partial y$
-18	-6	-12	6

> One backward sweep produced **all four** partial derivatives — no formula re-derivation per parameter.

All stored values and their gradients

V VISUAL



first half	w	x	m	b
forward value	2	3	6	1
gradient	$g_w = -18$	$g_x = -12$	$g_m = -6$	$g_b = -6$

second half	a	y	e	$\mathcal{L}$
forward value	7	10	-3	9
gradient	$g_a = -6$	$g_y = 6$	$g_e = -6$	$g_{\mathcal{L}} = 1$

Check with symbolic differentiation

D DERIVATION

Keep w, x, b, y symbolic, derive the formulas, and only then substitute $wx + b - y = -3$:

Check with symbolic differentiation

D DERIVATION

Keep w, x, b, y symbolic, derive the formulas, and only then substitute $wx + b - y = -3$:

$$\frac{\partial \mathcal{L}}{\partial w} = 2(wx + b - y)x = 2(-3)(3) = -18 \quad \checkmark$$

Check with symbolic differentiation

D DERIVATION

Keep w, x, b, y symbolic, derive the formulas, and only then substitute $wx + b - y = -3$:

$$\frac{\partial \mathcal{L}}{\partial w} = 2(wx + b - y)x = 2(-3)(3) = -18 \quad \checkmark$$

$$\frac{\partial \mathcal{L}}{\partial b} = 2(wx + b - y) = 2(-3) = -6 \quad \checkmark$$

Check with symbolic differentiation

D DERIVATION

Keep w, x, b, y symbolic, derive the formulas, and only then substitute $wx + b - y = -3$:

$$\frac{\partial \mathcal{L}}{\partial w} = 2(wx + b - y)x = 2(-3)(3) = -18 \quad \checkmark$$

$$\frac{\partial \mathcal{L}}{\partial b} = 2(wx + b - y) = 2(-3) = -6 \quad \checkmark$$

Backprop and the symbolic formulas agree at this point.

PyTorch executes the same graph

I INTERACTIVE

```
import torch

w = torch.tensor(2.0, requires_grad=True)
x = torch.tensor(3.0, requires_grad=True)
b = torch.tensor(1.0, requires_grad=True)
# Targets normally do not track gradients; enable it here to inspect dL/dy.
y = torch.tensor(10.0, requires_grad=True)

m = w * x
a = m + b
e = a - y
L = e**2
for value in (m, a, e, L):
    value.retain_grad()      # keep intermediate grads for inspection

L.backward()
print([t.item() for t in (m, a, e, L)])
# [6.0, 7.0, -3.0, 9.0]
print([t.grad.item() for t in (L, e, a, m, w, x, b, y)])
# [1.0, -6.0, -6.0, -6.0, -18.0, -12.0, -6.0, 6.0]
```

Hand computation → PyTorch autograd

V VISUAL

In our graph	In PyTorch	Meaning
stored scalar value	`Tensor`	one stored forward value
local backward rule	`grad_fn`	operation that produced a non-leaf tensor
saved forward value	saved by the operation	operand needed by a local derivative
reverse sweep	`L.backward()`	seed at $\mathcal{L}$; visit operations in reverse order
leaf derivative	`p.grad`	$\partial\mathcal{L}/\partial p$, accumulated with `+=`
intermediate derivative	`retain_grad()`	keep non-leaf `.grad` for inspection

Hand computation → PyTorch autograd

V VISUAL

In our graph	In PyTorch	Meaning
stored scalar value	`Tensor`	one stored forward value
local backward rule	`grad_fn`	operation that produced a non-leaf tensor
saved forward value	saved by the operation	operand needed by a local derivative
reverse sweep	`L.backward()`	seed at $\mathcal{L}$; visit operations in reverse order
leaf derivative	`p.grad`	$\partial\mathcal{L}/\partial p$, accumulated with `+=`
intermediate derivative	`retain_grad()`	keep non-leaf `.grad` for inspection

reverse mode = direction · backprop = algorithm · autograd = engine: record → save → replay

Checkpoint: read the sign of a gradient

Q QUESTION

We found $\partial\mathcal{L}/\partial w = -18$. For a small positive learning rate, which way does gradient descent move w?

A. Decrease w

B. Increase w

C. Leave w unchanged

D. The sign tells us nothing

Answer: a negative gradient means increase the parameter

A ANSWER

Correct: B — Increase w

Why: Gradient descent subtracts the gradient: $w \leftarrow w - \eta(-18) = w + 18\eta$.

A small gradient step reduces this loss

D DERIVATION

Take $\eta = 0.01$ and update:

$$w \leftarrow 2 - 0.01(-18) = 2.18, \qquad b \leftarrow 1 - 0.01(-6) = 1.06.$$

A small gradient step reduces this loss

D DERIVATION

Take $\eta = 0.01$ and update:

$$w \leftarrow 2 - 0.01(-18) = 2.18, \qquad b \leftarrow 1 - 0.01(-6) = 1.06.$$

New prediction: $\hat{y} = 2.18 \cdot 3 + 1.06 = 7.60$.

A small gradient step reduces this loss

D DERIVATION

Take $\eta = 0.01$ and update:

$$w \leftarrow 2 - 0.01(-18) = 2.18, \qquad b \leftarrow 1 - 0.01(-6) = 1.06.$$

New prediction: $\hat{y} = 2.18 \cdot 3 + 1.06 = 7.60$.

loss falls from 9 **to** $(7.60 - 10)^2 = 5.76$

A small gradient step reduces this loss

D DERIVATION

Take $\eta = 0.01$ and update:

$$w \leftarrow 2 - 0.01(-18) = 2.18, \qquad b \leftarrow 1 - 0.01(-6) = 1.06.$$

New prediction: $\hat{y} = 2.18 \cdot 3 + 1.06 = 7.60$.

loss falls from 9 **to** $(7.60 - 10)^2 = 5.76$

> Backprop supplied the direction. The learning rate controls how far we move; a later optimization lecture studies that choice.

Interactive: the computation graph

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/autograd

Build a scalar computation graph, run the forward pass, then click **backward** to watch each $\partial\mathcal{L}/\partial u$ fill in — exactly the $(wx + b - y)^2$ example above.

Interactive: the computation graph

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/autograd

Build a scalar computation graph, run the forward pass, then click **backward** to watch each $\partial\mathcal{L}/\partial u$ fill in — exactly the $(wx+b-y)^2$ example above.

Trace which local rule changes when the final squared-error operation is replaced with $|e|$.

Practice the scalar graph three ways

I INTERACTIVE

See it move

Step through values and loss derivatives in the interactive graph ↗.

Run one complete Colab

Open the lecture notebook ↓: manual graph, PyTorch, dense layer, batch, and MLP.

Follow the checkpoints

Predict first; then execute one checkpoint and compare. The same numbers recur throughout the lecture.

Practice the scalar graph three ways

I INTERACTIVE

See it move

Step through values and loss derivatives in the interactive graph ↗.

Run one complete Colab

Open the lecture notebook ↓: manual graph, PyTorch, dense layer, batch, and MLP.

Follow the checkpoints

Predict first; then execute one checkpoint and compare. The same numbers recur throughout the lecture.

Easy	change one input and recompute the forward pass
Medium	derive all four gradients before calling `backward()`
Hard	replace the square by $\lvert e \rvert$; state what happens at $e = 0$

From scalars to vectors

Derivative, gradient, Jacobian: the shape tells us

V VISUAL

We store gradients as column vectors. The name of the local object depends on the input and output shapes:

Derivative, gradient, Jacobian: the shape tells us

V VISUAL

We store gradients as column vectors. The name of the local object depends on the input and output shapes:

Operation shape	Local object	Name
$u \in \mathbb{R} \to v \in \mathbb{R}$	$\partial v / \partial u \in \mathbb{R}$	derivative
$\boldsymbol{x} \in \mathbb{R}^d \to z \in \mathbb{R}$	$\partial z / \partial \boldsymbol{x} \in \mathbb{R}^d$	gradient
$\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m$	$J_{ij} = \partial z_i / \partial x_j$	Jacobian $\boldsymbol{J} \in \mathbb{R}^{m \times d}$

Derivative, gradient, Jacobian: the shape tells us

V VISUAL

We store gradients as column vectors. The name of the local object depends on the input and output shapes:

Operation shape	Local object	Name
$u \in \mathbb{R} \to v \in \mathbb{R}$	$\partial v / \partial u \in \mathbb{R}$	derivative
$\boldsymbol{x} \in \mathbb{R}^d \to z \in \mathbb{R}$	$\partial z / \partial \boldsymbol{x} \in \mathbb{R}^d$	gradient
$\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m$	$J_{ij} = \partial z_i / \partial x_j$	Jacobian $\boldsymbol{J} \in \mathbb{R}^{m \times d}$

a Jacobian contains one local derivative for every output–input pair

A 2×2 Jacobian is four familiar derivatives

V VISUAL

Take a vector operation with two inputs and two outputs:

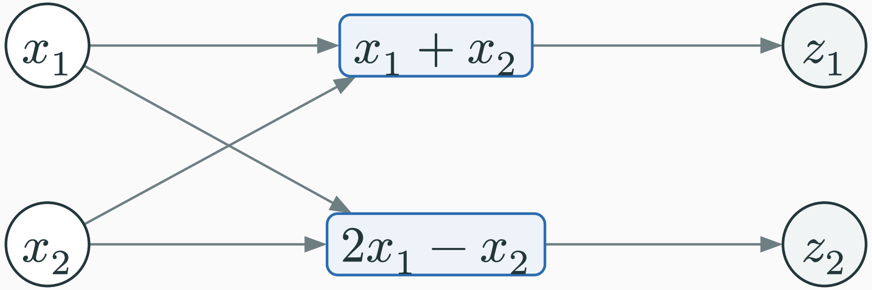
$$z_1 = x_1 + x_2, \quad z_2 = 2x_1 - x_2.$$

A 2×2 Jacobian is four familiar derivatives

V VISUAL

Take a vector operation with two inputs and two outputs:

$$z_1 = x_1 + x_2, \quad z_2 = 2x_1 - x_2.$$

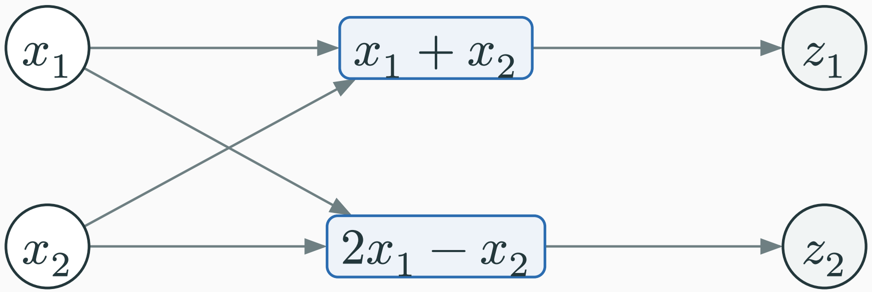


A 2×2 Jacobian is four familiar derivatives

VISUAL

Take a vector operation with two inputs and two outputs:

$$z_1 = x_1 + x_2, \quad z_2 = 2x_1 - x_2.$$

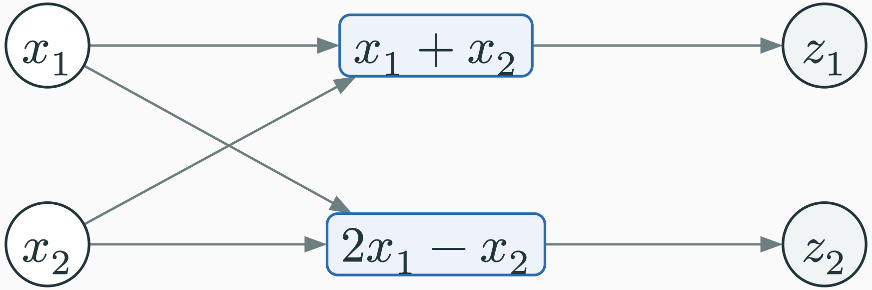


Output	Input x_1	Input x_2
z_1	$\partial z_1/\partial x_1 = 1$	$\partial z_1/\partial x_2 = 1$
z_2	$\partial z_2/\partial x_1 = 2$	$\partial z_2/\partial x_2 = -1$

A 2×2 Jacobian is four familiar derivatives

Take a vector operation with two inputs and two outputs:

$$z_1 = x_1 + x_2, \quad z_2 = 2x_1 - x_2.$$



Output	Input x_1	Input x_2
z_1	$\partial z_1/\partial x_1 = 1$	$\partial z_1/\partial x_2 = 1$
z_2	$\partial z_2/\partial x_1 = 2$	$\partial z_2/\partial x_2 = -1$

$J = \begin{pmatrix} 1 & 1 \\ 2 & -1 \end{pmatrix}$ · **rows are outputs · columns are inputs**

Vector chain rule: make the shapes fit

D DERIVATION

For $\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m$, let the local Jacobian be

$$\boldsymbol{J}_{ij} = \partial z_i / \partial x_j, \quad \boldsymbol{J} \in \mathbb{R}^{m \times d}.$$

Vector chain rule: make the shapes fit

D DERIVATION

For $\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m$, let the local Jacobian be

$$\boldsymbol{J}_{ij} = \partial z_i / \partial x_j, \quad \boldsymbol{J} \in \mathbb{R}^{m \times d}.$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \boldsymbol{J}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$$

Vector chain rule: make the shapes fit

D DERIVATION

For $\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m$, let the local Jacobian be

$$\boldsymbol{J}_{ij} = \partial z_i / \partial x_j, \quad \boldsymbol{J} \in \mathbb{R}^{m \times d}.$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \boldsymbol{J}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$$

$$(d \times 1) \quad = \quad (d \times m) \quad (m \times 1)$$

Vector chain rule: make the shapes fit

D DERIVATION

For $\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m$, let the local Jacobian be

$$\boldsymbol{J}_{ij} = \partial z_i / \partial x_j, \quad \boldsymbol{J} \in \mathbb{R}^{m \times d}.$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \boldsymbol{J}^\top \, \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$$

$$(d \times 1) \quad = \quad (d \times m) \quad (m \times 1)$$

downstream gradient = local Jacobian transpose × upstream gradient

Vector chain rule: make the shapes fit

D DERIVATION

For $\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m$, let the local Jacobian be

$$\boldsymbol{J}_{ij} = \partial z_i / \partial x_j, \quad \boldsymbol{J} \in \mathbb{R}^{m \times d}.$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \boldsymbol{J}^\top \ \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$$

$$(d \times 1) \quad = \quad (d \times m) \quad (m \times 1)$$

downstream gradient = local Jacobian transpose × upstream gradient

> When $d = m = 1$, $\boldsymbol{J}$ is one number and this is the scalar chain rule from earlier.

The transpose adds every output's contribution

D DERIVATION

For the previous operation, suppose the arriving gradient is

$$\boldsymbol{g}_z = \partial \mathcal{L} / \partial \boldsymbol{z} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}.$$

The transpose adds every output's contribution

D DERIVATION

For the previous operation, suppose the arriving gradient is

$$\boldsymbol{g}_z = \partial\mathcal{L}/\partial\boldsymbol{z} = \begin{pmatrix}3\\4\end{pmatrix}.$$

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}} = \boldsymbol{J}^\top\,\boldsymbol{g}_z = \begin{pmatrix}1 & 2\\1 & -1\end{pmatrix}\begin{pmatrix}3\\4\end{pmatrix} = \begin{pmatrix}11\\-1\end{pmatrix}$$

The transpose adds every output's contribution

D DERIVATION

For the previous operation, suppose the arriving gradient is

$$\boldsymbol{g}_z = \partial\mathcal{L}/\partial\boldsymbol{z} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}.$$

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}} = \boldsymbol{J}^\top \boldsymbol{g}_z = \begin{pmatrix} 1 & 2 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 11 \\ -1 \end{pmatrix}$$

$\partial\mathcal{L}/\partial x_1$	$3(1) + 4(2) = 11$
$\partial\mathcal{L}/\partial x_2$	$3(1) + 4(-1) = -1$

The transpose adds every output's contribution

D DERIVATION

For the previous operation, suppose the arriving gradient is

$$\boldsymbol{g}_z = \partial\mathcal{L}/\partial\boldsymbol{z} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}.$$

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}} = \boldsymbol{J}^\top \boldsymbol{g}_z = \begin{pmatrix} 1 & 2 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 11 \\ -1 \end{pmatrix}$$

$\partial\mathcal{L}/\partial x_1$	$3(1) + 4(2) = 11$
$\partial\mathcal{L}/\partial x_2$	$3(1) + 4(-1) = -1$

the vector rule is the scalar branch rule applied to every input

An affine operation adds scalar products and a bias

V VISUAL

Affine means a weighted sum plus a bias:

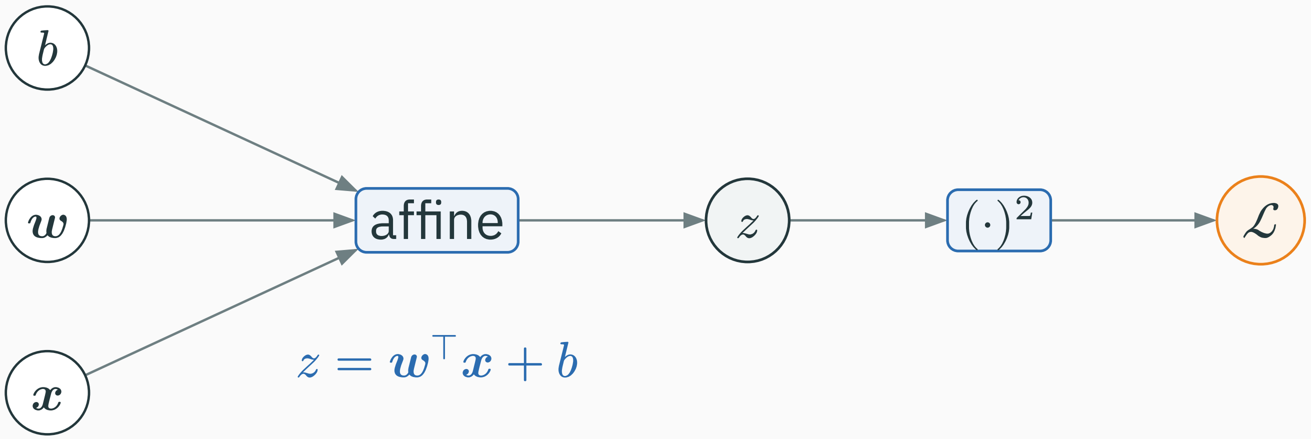
$$z = \boldsymbol{w}^\top \boldsymbol{x} + b \quad = \quad \sum_{j=1}^{d} w_j x_j + b.$$

An affine operation adds scalar products and a bias

V VISUAL

Affine means a weighted sum plus a bias:

$$z = \boldsymbol{w}^\top \boldsymbol{x} + b \quad = \quad \sum_{j=1}^{d} w_j x_j + b.$$



Change one scalar while holding every other value fixed

D DERIVATION

Start from $z = \sum_j w_j x_j + b$. Change only one input by a small amount Δ:

Change one scalar while holding every other value fixed

D DERIVATION

Start from $z = \sum_j w_j x_j + b$. Change only one input by a small amount Δ:

Change	Resulting change in z	Divide by Δ
$w_i \to w_i + \Delta$	$((w_i + \Delta)x_i) - w_i x_i = x_i \Delta$	$\partial z / \partial w_i = x_i$
$x_i \to x_i + \Delta$	$w_i(x_i + \Delta) - w_i x_i = w_i \Delta$	$\partial z / \partial x_i = w_i$
$b \to b + \Delta$	$(b + \Delta) - b = \Delta$	$\partial z / \partial b = 1$

Change one scalar while holding every other value fixed

D DERIVATION

Start from $z = \sum_j w_j x_j + b$. Change only one input by a small amount Δ:

Change	Resulting change in z	Divide by Δ
$w_i \to w_i + \Delta$	$((w_i + \Delta)x_i) - w_i x_i = x_i \Delta$	$\partial z / \partial w_i = x_i$
$x_i \to x_i + \Delta$	$w_i(x_i + \Delta) - w_i x_i = w_i \Delta$	$\partial z / \partial x_i = w_i$
$b \to b + \Delta$	$(b + \Delta) - b = \Delta$	$\partial z / \partial b = 1$

local derivative = change in the operation's output ÷ change in one input

Stack the coordinate derivatives into gradient vectors

D DERIVATION

The derivatives with respect to the coordinates of a vector are stored in one column:

Stack the coordinate derivatives into gradient vectors

D DERIVATION

The derivatives with respect to the coordinates of a vector are stored in one column:

$$\frac{\partial z}{\partial \boldsymbol{w}} := \begin{pmatrix} \partial z / \partial w_1 \\ \vdots \\ \partial z / \partial w_d \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} = \boldsymbol{x}$$

Stack the coordinate derivatives into gradient vectors

D DERIVATION

The derivatives with respect to the coordinates of a vector are stored in one column:

$$\frac{\partial z}{\partial \boldsymbol{w}} := \begin{pmatrix} \partial z / \partial w_1 \\ \vdots \\ \partial z / \partial w_d \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} = \boldsymbol{x}$$

$$\frac{\partial z}{\partial \boldsymbol{x}} := \begin{pmatrix} \partial z / \partial x_1 \\ \vdots \\ \partial z / \partial x_d \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix} = \boldsymbol{w}$$

Stack the coordinate derivatives into gradient vectors

D DERIVATION

The derivatives with respect to the coordinates of a vector are stored in one column:

$$\frac{\partial z}{\partial \boldsymbol{w}} := \begin{pmatrix} \partial z/\partial w_1 \\ \vdots \\ \partial z/\partial w_d \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} = \boldsymbol{x}$$

$$\frac{\partial z}{\partial \boldsymbol{x}} := \begin{pmatrix} \partial z/\partial x_1 \\ \vdots \\ \partial z/\partial x_d \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix} = \boldsymbol{w}$$

With $\boldsymbol{x} = \binom{2}{-1}$ and $\boldsymbol{w} = \binom{1}{3}$:

$$\frac{\partial z}{\partial \boldsymbol{w}} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \quad \frac{\partial z}{\partial \boldsymbol{x}} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \quad \frac{\partial z}{\partial b} = 1.$$

Stack the coordinate derivatives into gradient vectors

D DERIVATION

The derivatives with respect to the coordinates of a vector are stored in one column:

$$\frac{\partial z}{\partial \boldsymbol{w}} := \begin{pmatrix} \partial z/\partial w_1 \\ \vdots \\ \partial z/\partial w_d \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} = \boldsymbol{x}$$

$$\frac{\partial z}{\partial \boldsymbol{x}} := \begin{pmatrix} \partial z/\partial x_1 \\ \vdots \\ \partial z/\partial x_d \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix} = \boldsymbol{w}$$

With $\boldsymbol{x} = \binom{2}{-1}$ and $\boldsymbol{w} = \binom{1}{3}$:

$$\frac{\partial z}{\partial \boldsymbol{w}} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \quad \frac{\partial z}{\partial \boldsymbol{x}} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \quad \frac{\partial z}{\partial b} = 1.$$

the vector formulas are just the scalar coordinate derivatives stacked together

Affine forward: multiply componentwise, then add

D DERIVATION

Use $x = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$, $w = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$, and $b = 0$.

Affine forward: multiply componentwise, then add

D DERIVATION

Use $x = \binom{2}{-1}$, $w = \binom{1}{3}$, and $b = 0$.

Coordinate	x_i	w_i	$w_i x_i$
1	2	1	$1 \cdot 2 = 2$
2	-1	3	$3 \cdot (-1) = -3$

Affine forward: multiply componentwise, then add

D DERIVATION

Use $\boldsymbol{x} = \binom{2}{-1}$, $\boldsymbol{w} = \binom{1}{3}$, and $b = 0$.

Coordinate	x_i	w_i	$w_i x_i$
1	2	1	$1 \cdot 2 = 2$
2	-1	3	$3 \cdot (-1) = -3$

$$\boldsymbol{w}^\top \boldsymbol{x} = 2 + (-3) = -1, \quad z = -1 + 0 = -1.$$

Affine forward: multiply componentwise, then add

D DERIVATION

Use $\boldsymbol{x} = \binom{2}{-1}$, $\boldsymbol{w} = \binom{1}{3}$, and $b = 0$.

Coordinate	x_i	w_i	$w_i x_i$
1	2	1	$1 \cdot 2 = 2$
2	-1	3	$3 \cdot (-1) = -3$

$$\boldsymbol{w}^\top \boldsymbol{x} = 2 + (-3) = -1, \quad z = -1 + 0 = -1.$$

$$\mathcal{L} = z^2 = (-1)^2 = 1.$$

Affine forward: multiply componentwise, then add

D DERIVATION

Use $\boldsymbol{x} = \binom{2}{-1}$, $\boldsymbol{w} = \binom{1}{3}$, and $b = 0$.

Coordinate	x_i	w_i	$w_i x_i$
1	2	1	$1 \cdot 2 = 2$
2	-1	3	$3 \cdot (-1) = -3$

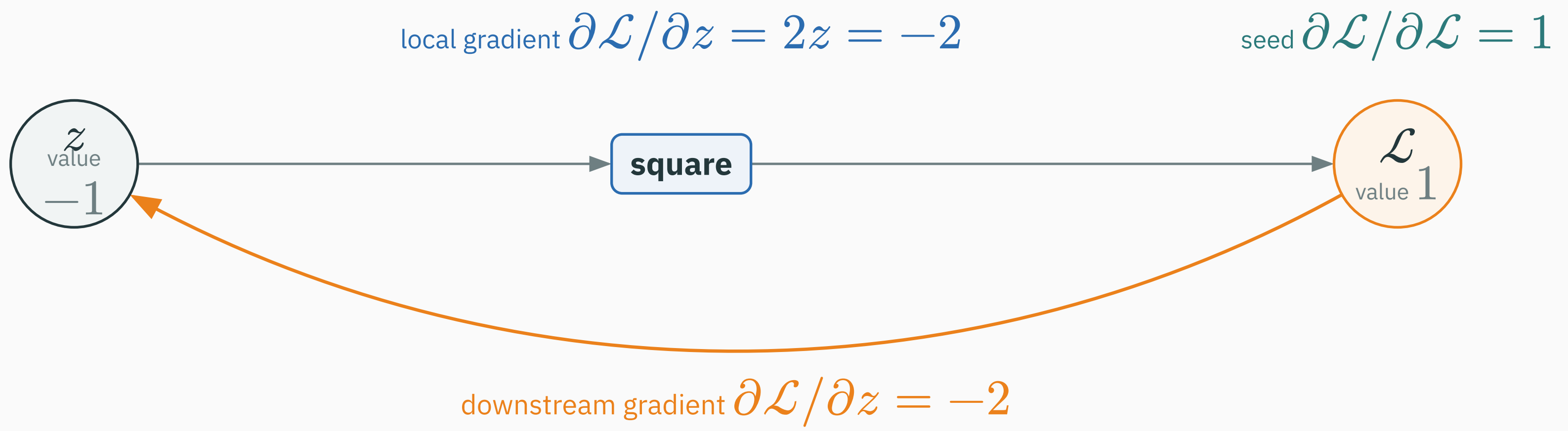
$$\boldsymbol{w}^\top \boldsymbol{x} = 2 + (-3) = -1, \quad z = -1 + 0 = -1.$$

$$\mathcal{L} = z^2 = (-1)^2 = 1.$$

dot product = multiply matching entries, then sum

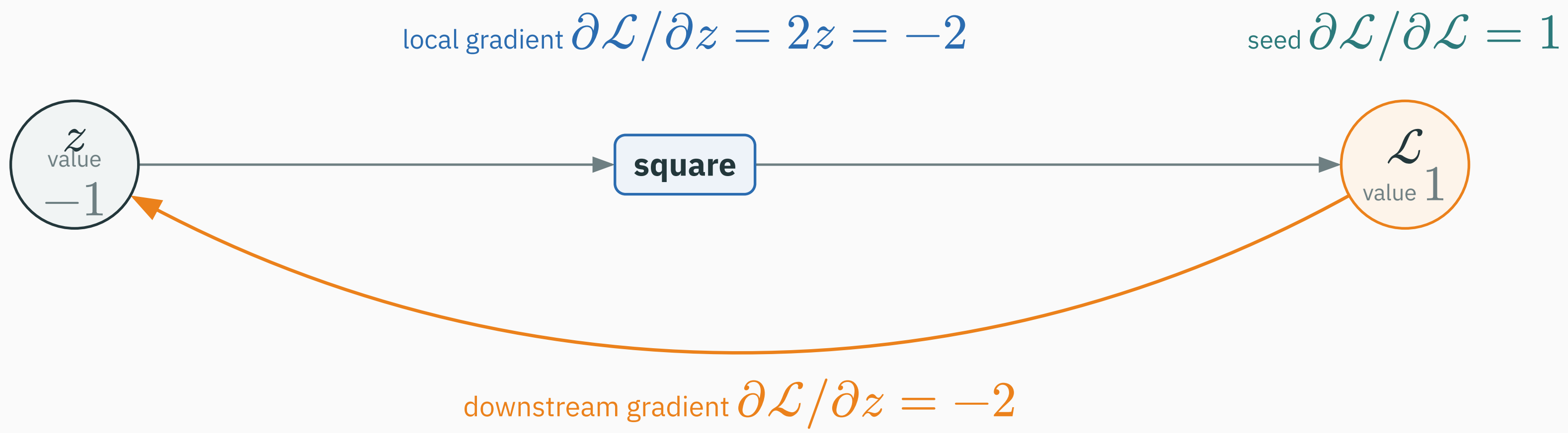
First backward step: the square operation

D DERIVATION



First backward step: the square operation

D DERIVATION

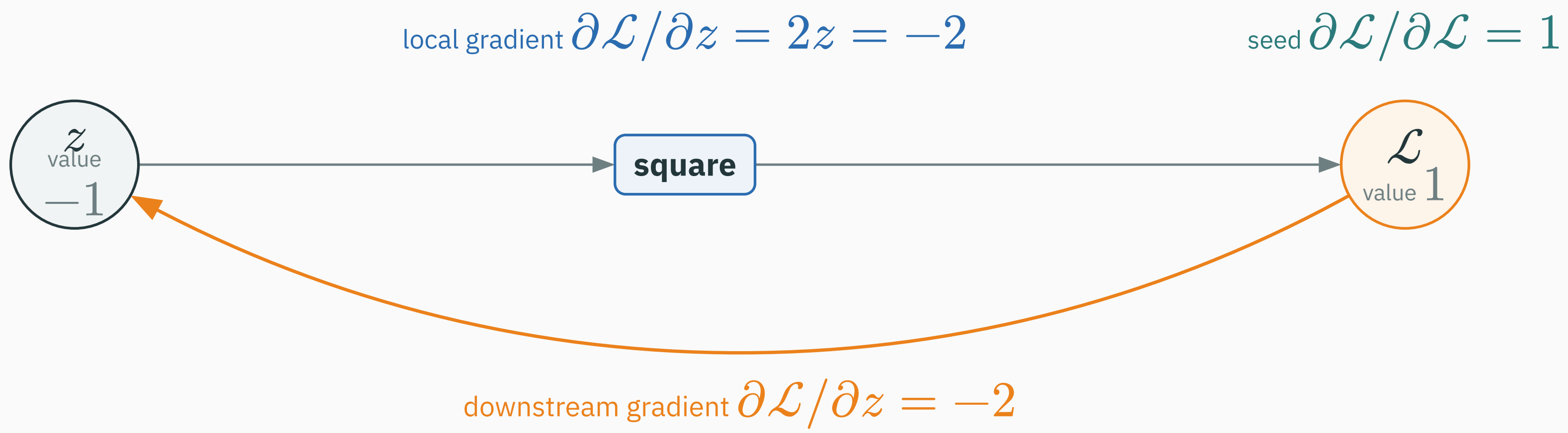


Seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$. The square has local derivative $2z = -2$, so

$$\frac{\partial\mathcal{L}}{\partial z} = 1 \cdot 2z = -2.$$

First backward step: the square operation

D DERIVATION



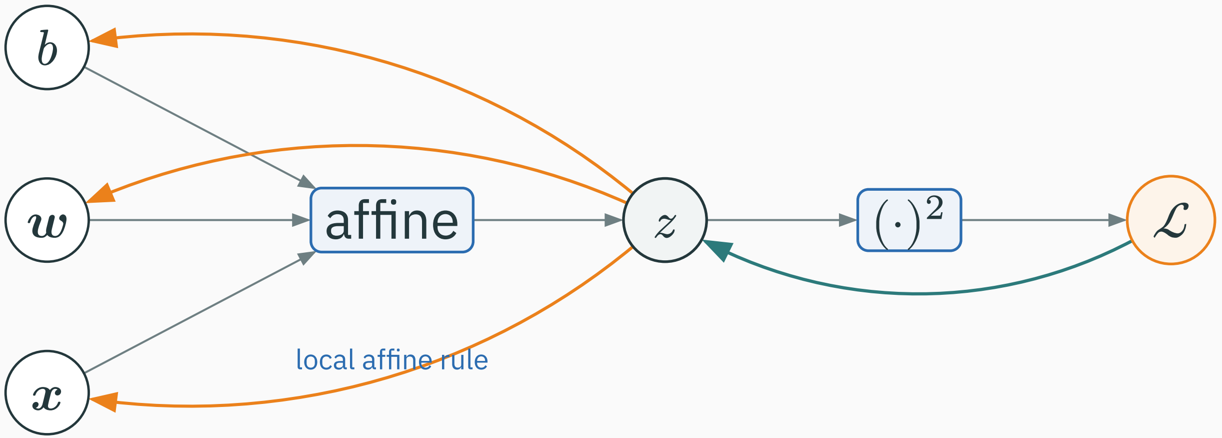
Seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$. The square has local derivative $2z = -2$, so

$$\frac{\partial\mathcal{L}}{\partial z} = 1 \cdot 2z = -2.$$

the square returns -2; it now arrives at the affine operation

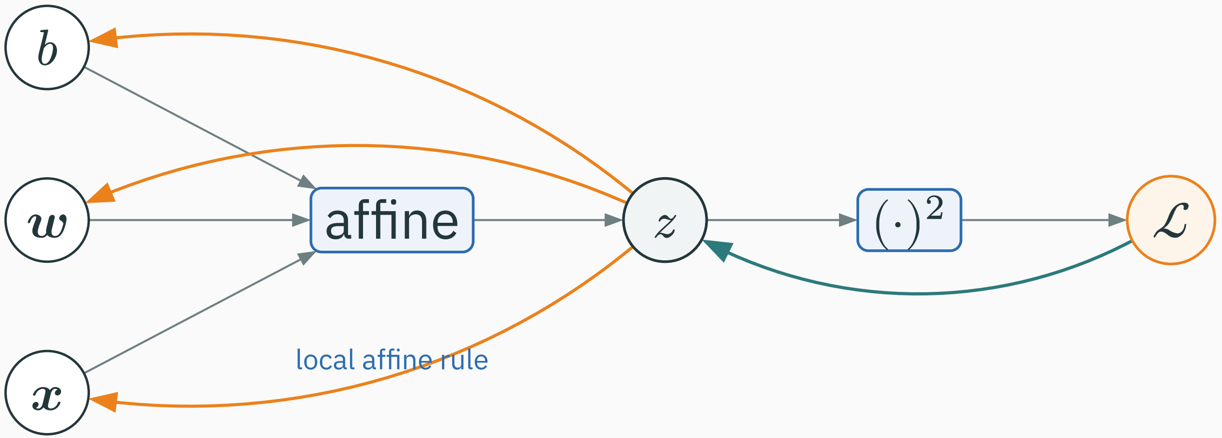
Affine backward: apply the three local gradients

D DERIVATION



Affine backward: apply the three local gradients

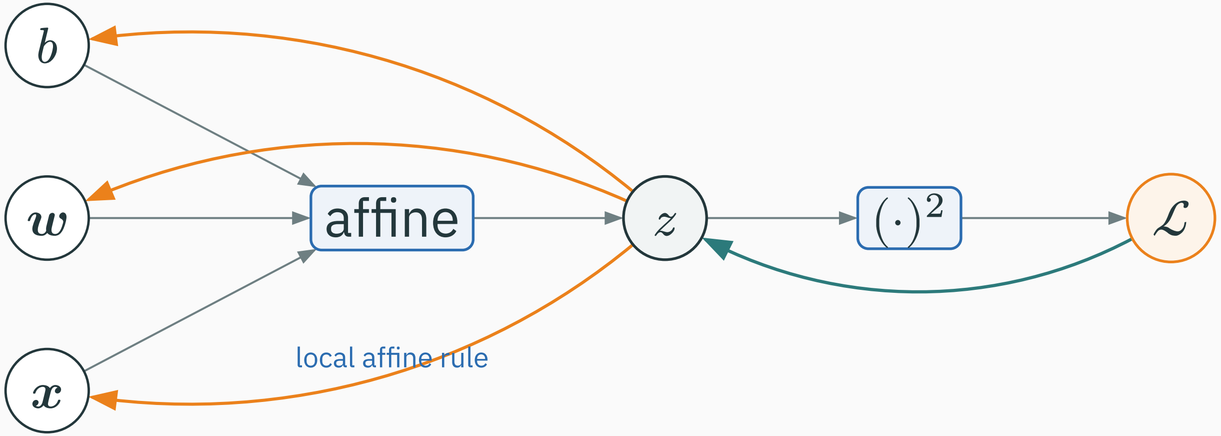
D DERIVATION



Let $g_z = \partial\mathcal{L}/\partial z = -2$ arrive at the affine operation. Its local gradients send back:

Affine backward: apply the three local gradients

D DERIVATION

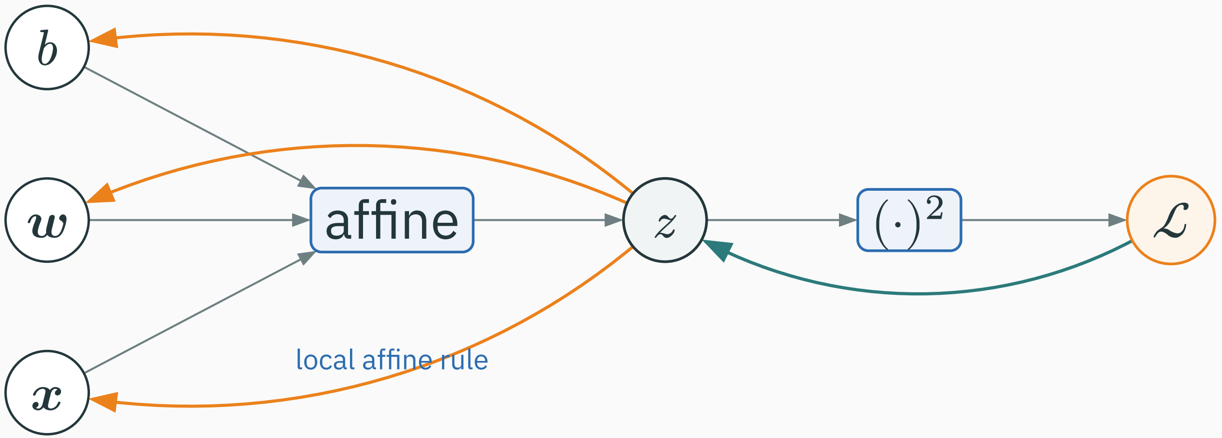


Let $g_z = \partial\mathcal{L}/\partial z = -2$ arrive at the affine operation. Its local gradients send back:

Input	**Local gradient**	**Downstream gradient**
$\boldsymbol{w}$	$\partial z/\partial \boldsymbol{w} = \boldsymbol{x}$	$\partial\mathcal{L}/\partial \boldsymbol{w} = (-2)\boldsymbol{x} = (-4, 2)^\top$
$\boldsymbol{x}$	$\partial z/\partial \boldsymbol{x} = \boldsymbol{w}$	$\partial\mathcal{L}/\partial \boldsymbol{x} = (-2)\boldsymbol{w} = (-2, -6)^\top$
b	$\partial z/\partial b = 1$	$\partial\mathcal{L}/\partial b = -2$

Affine backward: apply the three local gradients

D DERIVATION



Let $g_z = \partial\mathcal{L}/\partial z = -2$ arrive at the affine operation. Its local gradients send back:

Input	Local gradient	Downstream gradient
$\boldsymbol{w}$	$\partial z/\partial \boldsymbol{w} = \boldsymbol{x}$	$\partial\mathcal{L}/\partial \boldsymbol{w} = (-2)\boldsymbol{x} = (-4, 2)^\top$
$\boldsymbol{x}$	$\partial z/\partial \boldsymbol{x} = \boldsymbol{w}$	$\partial\mathcal{L}/\partial \boldsymbol{x} = (-2)\boldsymbol{w} = (-2, -6)^\top$
b	$\partial z/\partial b = 1$	$\partial\mathcal{L}/\partial b = -2$

one arriving gradient g_z produces three separate downstream gradients

PyTorch gives the same vector gradients

I INTERACTIVE

```
import torch

x = torch.tensor([2., -1.], requires_grad=True)
w = torch.tensor([1.,  3.], requires_grad=True)
b = torch.tensor(0., requires_grad=True)

z = w @ x + b
L = z**2
L.backward()

print(z.item(), L.item())  # -1.0, 1.0
print(w.grad)              # tensor([-4.,  2.])
print(x.grad, b.grad)      # tensor([-2., -6.]), tensor(-2.)
```

PyTorch gives the same vector gradients

INTERACTIVE

```
import torch

x = torch.tensor([2., -1.], requires_grad=True)
w = torch.tensor([1.,  3.], requires_grad=True)
b = torch.tensor(0., requires_grad=True)

z = w @ x + b
L = z**2
L.backward()

print(z.item(), L.item())  # -1.0, 1.0
print(w.grad)              # tensor([-4.,  2.])
print(x.grad, b.grad)      # tensor([-2., -6.]), tensor(-2.)
```

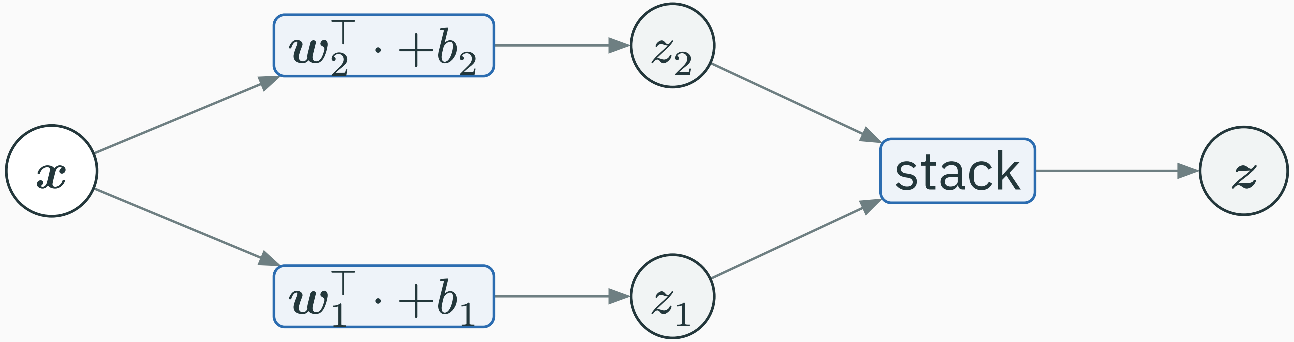
`@` performs the dot product; `backward()` still applies the same local rules

From one scalar output to a vector output

A dense layer repeats one familiar neuron

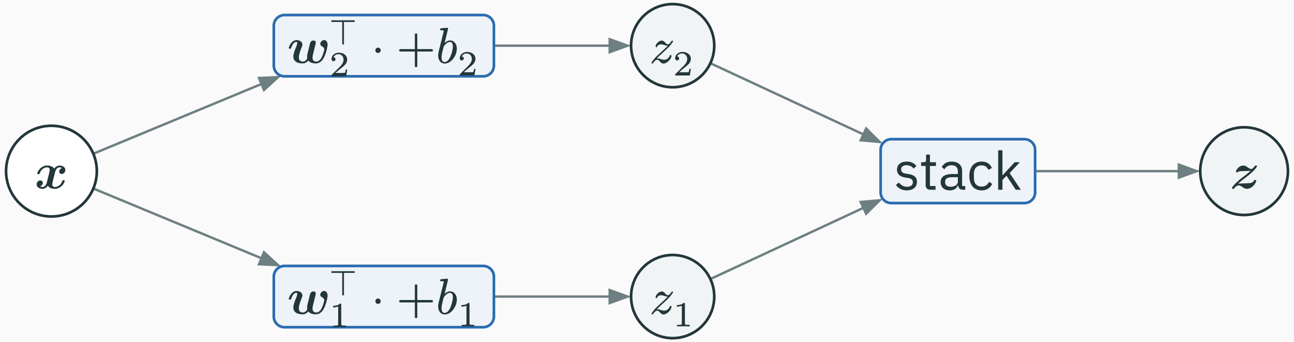
V VISUAL

Every row of $\boldsymbol{W}$ is one affine neuron. All rows read the same input $\boldsymbol{x}$.



A dense layer repeats one familiar neuron

Every row of $\boldsymbol{W}$ is one affine neuron. All rows read the same input $\boldsymbol{x}$.



ROW 1 · neuron 1

$$z_1 = \boldsymbol{w}_1^\top \boldsymbol{x} + b_1$$

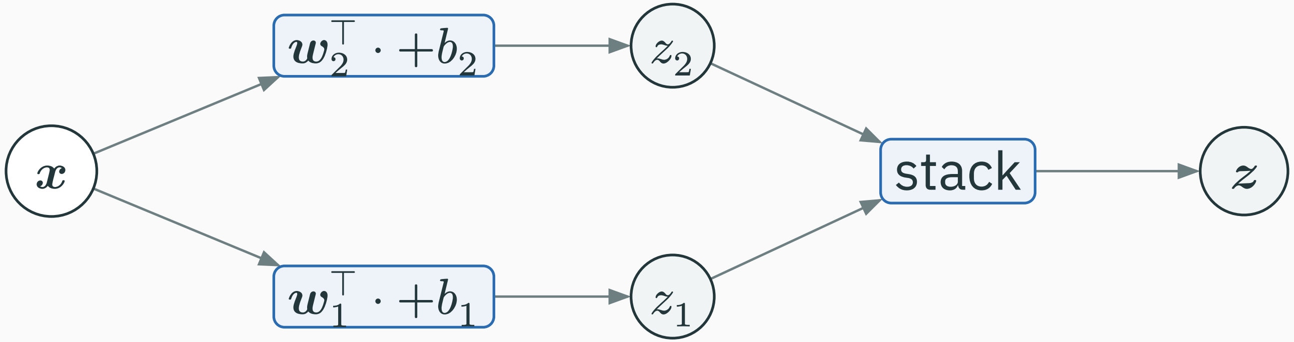
ROW 2 · neuron 2

$$z_2 = \boldsymbol{w}_2^\top \boldsymbol{x} + b_2$$

A dense layer repeats one familiar neuron

V VISUAL

Every row of $\boldsymbol{W}$ is one affine neuron. All rows read the same input $\boldsymbol{x}$.



ROW 1 · neuron 1

$$z_1 = \boldsymbol{w}_1^\top \boldsymbol{x} + b_1$$

ROW 2 · neuron 2

$$z_2 = \boldsymbol{w}_2^\top \boldsymbol{x} + b_2$$

stack the outputs: $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$

Forward, row 1: do one dot product

D DERIVATION

$$\boldsymbol{x} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \quad \boldsymbol{w}_1^\top = (1, 3), \quad b_1 = 0$$

Forward, row 1: do one dot product

D DERIVATION

$$\boldsymbol{x} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \quad \boldsymbol{w}_1^\top = (1, 3), \quad b_1 = 0$$

$$z_1 = \underbrace{1(2)}_{\text{first input}} + \underbrace{3(-1)}_{\text{second input}} + 0 = -1$$

Forward, row 1: do one dot product

D DERIVATION

$$\boldsymbol{x} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \quad \boldsymbol{w}_1^\top = (1, 3), \quad b_1 = 0$$

$$z_1 = \underbrace{1(2)}_{\text{first input}} + \underbrace{3(-1)}_{\text{second input}} + 0 = -1$$

> Nothing new happened: multiply matching entries, add them, then add the bias.

Forward, row 2: repeat, then stack

D DERIVATION

$$\boldsymbol{w}_2^\top = (-2, 1), \quad b_2 = 1$$

Forward, row 2: repeat, then stack

DERIVATION

$$\boldsymbol{w}_2^\top = (-2, 1), \quad b_2 = 1$$

$$z_2 = (-2)(2) + 1(-1) + 1 = -4$$

Forward, row 2: repeat, then stack

D DERIVATION

$$\boldsymbol{w}_2^\top = (-2, 1), \quad b_2 = 1$$

$$z_2 = (-2)(2) + 1(-1) + 1 = -4$$

$$\boldsymbol{z} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} -1 \\ -4 \end{pmatrix}$$

Forward, row 2: repeat, then stack

D DERIVATION

$$\boldsymbol{w}_2^\top = (-2, 1), \quad b_2 = 1$$

$$z_2 = (-2)(2) + 1(-1) + 1 = -4$$

$$\boldsymbol{z} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} -1 \\ -4 \end{pmatrix}$$

$\boldsymbol{W} = \begin{pmatrix} 1 & 3 \\ -2 & 1 \end{pmatrix}$ **simply stores the two neuron rows together**

Backward: reuse the scalar affine rule for each row

D DERIVATION

Use column vectors: $\boldsymbol{x} \in \mathbb{R}^{d\times 1}$, $\boldsymbol{W} \in \mathbb{R}^{m\times d}$, and $\boldsymbol{z} \in \mathbb{R}^{m\times 1}$. Suppose the later loss sends $\boldsymbol{g}_z = \partial\mathcal{L}/\partial \boldsymbol{z} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$ back.

Backward: reuse the scalar affine rule for each row

D DERIVATION

Use column vectors: $\boldsymbol{x} \in \mathbb{R}^{d\times 1}$, $\boldsymbol{W} \in \mathbb{R}^{m\times d}$, and $\boldsymbol{z} \in \mathbb{R}^{m\times 1}$. Suppose the later loss sends $\boldsymbol{g}_z = \partial\mathcal{L}/\partial\boldsymbol{z} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$ back. For output row i,

$$z_i = \sum_j W_{i,j} x_j + b_i, \quad g_i := \partial\mathcal{L}/\partial z_i.$$

Backward: reuse the scalar affine rule for each row

D DERIVATION

Use column vectors: $\boldsymbol{x} \in \mathbb{R}^{d\times 1}$, $\boldsymbol{W} \in \mathbb{R}^{m\times d}$, and $\boldsymbol{z} \in \mathbb{R}^{m\times 1}$. Suppose the later loss sends $\boldsymbol{g}_z = \partial\mathcal{L}/\partial\boldsymbol{z} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$ back. For output row i,

$$z_i = \sum_j W_{i,j}x_j + b_i, \quad g_i := \partial\mathcal{L}/\partial z_i.$$

TO x_j

$$\partial z_i/\partial x_j = W_{i,j}$$

contribution $g_i W_{i,j}$

TO $W_{i,j}$

$$\partial z_i/\partial W_{i,j} = x_j$$

contribution $g_i x_j$

TO b_i

$$\partial z_i/\partial b_i = 1$$

contribution g_i

Backward: reuse the scalar affine rule for each row

D DERIVATION

Use column vectors: $\boldsymbol{x} \in \mathbb{R}^{d\times 1}$, $\boldsymbol{W} \in \mathbb{R}^{m\times d}$, and $\boldsymbol{z} \in \mathbb{R}^{m\times 1}$. Suppose the later loss sends $\boldsymbol{g}_z = \partial\mathcal{L}/\partial\boldsymbol{z} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$ back. For output row i,

$$z_i = \sum_j W_{i,j} x_j + b_i, \quad g_i := \partial\mathcal{L}/\partial z_i.$$

TO x_j	TO $W_{i,j}$	TO b_i
$\partial z_i/\partial x_j = W_{i,j}$ contribution $g_i W_{i,j}$	$\partial z_i/\partial W_{i,j} = x_j$ contribution $g_i x_j$	$\partial z_i/\partial b_i = 1$ contribution g_i

$\boldsymbol{g}_z = (4, -2)^\top$ **means two scalar arrivals:** $g_1 = 4$ **and** $g_2 = -2$

Input gradient: shared paths add

D DERIVATION

The same x enters both output rows, so both rows return a contribution.

Input gradient: shared paths add

D DERIVATION

The same x enters both output rows, so both rows return a contribution.

ROW 1 RETURNS

$$g_1 \boldsymbol{w}_1 = 4\binom{1}{3} = \binom{4}{12}$$

ROW 2 RETURNS

$$g_2 \boldsymbol{w}_2 = (-2)\binom{-2}{1} = \binom{4}{-2}$$

Input gradient: shared paths add

D DERIVATION

The same x enters both output rows, so both rows return a contribution.

ROW 1 RETURNS

$$g_1 \boldsymbol{w}_1 = 4\begin{pmatrix}1\\3\end{pmatrix} = \begin{pmatrix}4\\12\end{pmatrix}$$

ROW 2 RETURNS

$$g_2 \boldsymbol{w}_2 = (-2)\begin{pmatrix}-2\\1\end{pmatrix} = \begin{pmatrix}4\\-2\end{pmatrix}$$

$$\boldsymbol{g}_x = \begin{pmatrix}4\\12\end{pmatrix} + \begin{pmatrix}4\\-2\end{pmatrix} = \begin{pmatrix}8\\10\end{pmatrix}$$

Input gradient: shared paths add

D DERIVATION

The same x enters both output rows, so both rows return a contribution.

ROW 1 RETURNS

$$g_1 \boldsymbol{w}_1 = 4\binom{1}{3} = \binom{4}{12}$$

ROW 2 RETURNS

$$g_2 \boldsymbol{w}_2 = (-2)\binom{-2}{1} = \binom{4}{-2}$$

$$\boldsymbol{g}_x = \binom{4}{12} + \binom{4}{-2} = \binom{8}{10}$$

$$(g_x)_j = \sum_i g_i W_{i,j} \quad \to \quad \boldsymbol{g}_x = \boldsymbol{W}^\top \boldsymbol{g}_z$$

Input gradient: shared paths add

D DERIVATION

The same x enters both output rows, so both rows return a contribution.

ROW 1 RETURNS

$$g_1 \boldsymbol{w}_1 = 4\binom{1}{3} = \binom{4}{12}$$

ROW 2 RETURNS

$$g_2 \boldsymbol{w}_2 = (-2)\binom{-2}{1} = \binom{4}{-2}$$

$$\boldsymbol{g}_x = \binom{4}{12} + \binom{4}{-2} = \binom{8}{10}$$

$$(g_x)_j = \sum_i g_i W_{i,j} \quad \to \quad \boldsymbol{g}_x = \boldsymbol{W}^\top \boldsymbol{g}_z$$

> Shape check: $(d \times 1) = (d \times m)(m \times 1)$. The transpose arranges the row contributions so shared input paths add.

Weight and bias gradients: separate rows stack

D DERIVATION

Output i owns row i of $\boldsymbol{W}$, so that gradient row is $g_i \boldsymbol{x}^\top$.

Weight and bias gradients: separate rows stack

D DERIVATION

Output i owns row i of $\boldsymbol{W}$, so that gradient row is $g_i\boldsymbol{x}^\top$.

ROW 1

$$g_1\boldsymbol{x}^\top = 4(2,-1) = (8,-4)$$

ROW 2

$$g_2\boldsymbol{x}^\top = (-2)(2,-1) = (-4,2)$$

Weight and bias gradients: separate rows stack

D DERIVATION

Output i owns row i of $\boldsymbol{W}$, so that gradient row is $g_i \boldsymbol{x}^\top$.

ROW 1

$$g_1 \boldsymbol{x}^\top = 4(2, -1) = (8, -4)$$

ROW 2

$$g_2 \boldsymbol{x}^\top = (-2)(2, -1) = (-4, 2)$$

$$\boldsymbol{g}_W = \begin{pmatrix} 8 & -4 \\ -4 & 2 \end{pmatrix} = \boldsymbol{g}_z \, \boldsymbol{x}^\top$$

Weight and bias gradients: separate rows stack

D DERIVATION

Output i owns row i of $\boldsymbol{W}$, so that gradient row is $g_i\boldsymbol{x}^\top$.

ROW 1

$$g_1\boldsymbol{x}^\top = 4(2,-1) = (8,-4)$$

ROW 2

$$g_2\boldsymbol{x}^\top = (-2)(2,-1) = (-4,2)$$

$$\boldsymbol{g}_W = \begin{pmatrix} 8 & -4 \\ -4 & 2 \end{pmatrix} = \boldsymbol{g}_z\,\boldsymbol{x}^\top$$

Each bias has local derivative 1, so

$$\boldsymbol{g}_b = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix} = \boldsymbol{g}_z = \begin{pmatrix} 4 \\ -2 \end{pmatrix}.$$

Weight and bias gradients: separate rows stack

D DERIVATION

Output i owns row i of $\boldsymbol{W}$, so that gradient row is $g_i\boldsymbol{x}^\top$.

ROW 1

$$g_1\boldsymbol{x}^\top = 4(2,-1) = (8,-4)$$

ROW 2

$$g_2\boldsymbol{x}^\top = (-2)(2,-1) = (-4,2)$$

$$\boldsymbol{g}_W = \begin{pmatrix} 8 & -4 \\ -4 & 2 \end{pmatrix} = \boldsymbol{g}_z\,\boldsymbol{x}^\top$$

Each bias has local derivative 1, so

$$\boldsymbol{g}_b = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix} = \boldsymbol{g}_z = \begin{pmatrix} 4 \\ -2 \end{pmatrix}.$$

returns to a shared input add · returns to separate parameter rows stack

Dense backward: derive first, then check shapes

D DERIVATION

Recipient	Coordinate rule	Stacked rule	Shape check
$\boldsymbol{x}$	$(g_x)_j = \sum_i g_i W_{i,j}$	$\boldsymbol{g}_x = \boldsymbol{W}^\top \boldsymbol{g}_z$	$(d \times 1) = (d \times m)(m \times 1)$
$\boldsymbol{W}$	$(g_W)_{i,j} = g_i x_j$	$\boldsymbol{g}_W = \boldsymbol{g}_z \boldsymbol{x}^\top$	$(m \times d) = (m \times 1)(1 \times d)$
$\boldsymbol{b}$	$(g_b)_i = g_i$	$\boldsymbol{g}_b = \boldsymbol{g}_z$	$(m \times 1) = (m \times 1)$

Dense backward: derive first, then check shapes

D DERIVATION

Recipient	Coordinate rule	Stacked rule	Shape check
$\boldsymbol{x}$	$(g_x)_j = \sum_i g_i W_{i,j}$	$g_x = \boldsymbol{W}^\top \boldsymbol{g}_z$	$(d \times 1) = (d \times m)(m \times 1)$
$\boldsymbol{W}$	$(g_W)_{i,j} = g_i x_j$	$g_W = \boldsymbol{g}_z \, \boldsymbol{x}^\top$	$(m \times d) = (m \times 1)(1 \times d)$
$\boldsymbol{b}$	$(g_b)_i = g_i$	$g_b = \boldsymbol{g}_z$	$(m \times 1) = (m \times 1)$

The coordinate rules **derive** the gradients. Shape matching then verifies their orientation: $\boldsymbol{g}_z \boldsymbol{x}^\top$ has the same row-by-column shape as $\boldsymbol{W}$.

From one example to a batch

A batch adds an axis—not new parameters

V VISUAL

One example remains a column $\boldsymbol{x}^n \in \mathbb{R}^{d\times 1}$. A framework stores their transposes as rows of

$$\boldsymbol{X} = \begin{pmatrix} 2 & -1 \\ -1 & 2 \\ 2 & 2 \end{pmatrix} \in \mathbb{R}^{3\times 2}.$$

A batch adds an axis—not new parameters

V VISUAL

One example remains a column $\boldsymbol{x}^n \in \mathbb{R}^{d\times 1}$. A framework stores their transposes as rows of

$$\boldsymbol{X} = \begin{pmatrix} 2 & -1 \\ -1 & 2 \\ 2 & 2 \end{pmatrix} \in \mathbb{R}^{3\times 2}.$$

SHARED PARAMETERS

$$\boldsymbol{W} \in \mathbb{R}^{m\times d}$$
$$\boldsymbol{b} \in \mathbb{R}^{m\times 1}$$

the same values serve every row

BATCH FORWARD

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W}^\top + \mathbf{1}_B \boldsymbol{b}^\top$$
$$(B \times m) = (B \times d)(d \times m)$$

PyTorch: `X @ W.T + b`

A batch adds an axis—not new parameters

V VISUAL

One example remains a column $\boldsymbol{x}^n \in \mathbb{R}^{d\times 1}$. A framework stores their transposes as rows of

$$\boldsymbol{X} = \begin{pmatrix} 2 & -1 \\ -1 & 2 \\ 2 & 2 \end{pmatrix} \in \mathbb{R}^{3\times 2}.$$

SHARED PARAMETERS

$$\boldsymbol{W} \in \mathbb{R}^{m\times d}$$
$$\boldsymbol{b} \in \mathbb{R}^{m\times 1}$$

the same values serve every row

BATCH FORWARD

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W}^\top + \mathbf{1}_B \boldsymbol{b}^\top$$
$$(B \times m) = (B \times d)(d \times m)$$

PyTorch: `X @ W.T + b`

batching adds an example axis; every row still uses the same W, b

Three examples, three residuals

V VISUAL

For row n, first compute $\boldsymbol{z}^n$, then $\boldsymbol{r}^n = \boldsymbol{z}^n - \boldsymbol{y}^n$.

Three examples, three residuals

V VISUAL

For row n, first compute $\boldsymbol{z}^n$, then $\boldsymbol{r}^n = \boldsymbol{z}^n - \boldsymbol{y}^n$.

n	${\boldsymbol{x}^{(n)}}^\top$	${\boldsymbol{z}^{(n)}}^\top$	${\boldsymbol{y}^{(n)}}^\top$	${\boldsymbol{r}^{(n)}}^\top$	ℓ_n
1	$(2, -1)$	$(-1, -4)$	$(-5, -2)$	$(4, -2)$	10
2	$(-1, 2)$	$(5, 5)$	$(7, 1)$	$(-2, 4)$	10
3	$(2, 2)$	$(8, -1)$	$(7, -2)$	$(1, 1)$	1

Three examples, three residuals

V VISUAL

For row n, first compute $\boldsymbol{z}^n$, then $\boldsymbol{r}^n = \boldsymbol{z}^n - \boldsymbol{y}^n$.

n	$\boldsymbol{x}^{(n)\top}$	$\boldsymbol{z}^{(n)\top}$	$\boldsymbol{y}^{(n)\top}$	$\boldsymbol{r}^{(n)\top}$	ℓ_n
1	$(2, -1)$	$(-1, -4)$	$(-5, -2)$	$(4, -2)$	10
2	$(-1, 2)$	$(5, 5)$	$(7, 1)$	$(-2, 4)$	10
3	$(2, 2)$	$(8, -1)$	$(7, -2)$	$(1, 1)$	1

row 1 reproduces the earlier single example: its residual $(4, -2)^\top$ is the arriving g_z

Why the loss sends residuals backward

D DERIVATION

For one example, use half-squared error:

$$\ell_n = \tfrac{1}{2} \sum_k \left(z_k^n - y_k^n\right)^2.$$

Why the loss sends residuals backward

D DERIVATION

For one example, use half-squared error:

$$\ell_n = \tfrac{1}{2} \textstyle\sum_k \left(z_k^n - y_k^n\right)^2.$$

$$\partial \ell_n / \partial z_k^n = z_k^n - y_k^n = r_k^n$$

Why the loss sends residuals backward

D DERIVATION

For one example, use half-squared error:

$$\ell_n = \frac{1}{2}\sum_k \left(z_k^n - y_k^n\right)^2.$$

$$\partial \ell_n / \partial z_k^n = z_k^n - y_k^n = r_k^n$$

THREE LOSSES

$$\ell_1 = 10, \quad \ell_2 = 10, \quad \ell_3 = 1$$

ONE SCALAR MEAN

$$\mathcal{L} = \frac{1}{3}(10 + 10 + 1) = 7$$

Why the loss sends residuals backward

D DERIVATION

For one example, use half-squared error:

$$\ell_n = \tfrac{1}{2} \sum_k \left(z_k^n - y_k^n\right)^2.$$

$$\partial \ell_n / \partial z_k^n = z_k^n - y_k^n = r_k^n$$

THREE LOSSES

$$\ell_1 = 10, \quad \ell_2 = 10, \quad \ell_3 = 1$$

ONE SCALAR MEAN

$$\mathcal{L} = \tfrac{1}{3}(10 + 10 + 1) = 7$$

$$\boldsymbol{G}_Z = \partial \mathcal{L} / \partial \boldsymbol{Z} = \tfrac{1}{3} \boldsymbol{R}$$

Why the loss sends residuals backward

D DERIVATION

For one example, use half-squared error:

$$\ell_n = \tfrac{1}{2}\sum_k \left(z_k^n - y_k^n\right)^2.$$

$$\partial \ell_n / \partial z_k^n = z_k^n - y_k^n = r_k^n$$

THREE LOSSES

$$\ell_1 = 10, \quad \ell_2 = 10, \quad \ell_3 = 1$$

ONE SCALAR MEAN

$$\mathcal{L} = \tfrac{1}{3}(10 + 10 + 1) = 7$$

$$\boldsymbol{G}_Z = \partial \mathcal{L} / \partial \boldsymbol{Z} = \tfrac{1}{3}\boldsymbol{R}$$

> The residual is the upstream gradient from each example's loss. The mean contributes the factor $\frac{1}{B} = \frac{1}{3}$.

One example proposes one dense-layer gradient

D DERIVATION

Reuse the single-example rules on example 1, before doing any batch algebra.

One example proposes one dense-layer gradient

D DERIVATION

Reuse the single-example rules on example 1, before doing any batch algebra.

TO $\boldsymbol{W}$

$$\boldsymbol{G}_W^1 = \boldsymbol{r}^1(\boldsymbol{x}^1)^\top = \begin{pmatrix} 8 & -4 \\ -4 & 2 \end{pmatrix}$$

TO $\boldsymbol{b}$

$$\boldsymbol{g}_b^1 = \boldsymbol{r}^1 = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

TO $\boldsymbol{x}^1$

$$\boldsymbol{g}_x^1 = \boldsymbol{W}^\top \boldsymbol{r}^1 = \begin{pmatrix} 8 \\ 10 \end{pmatrix}$$

One example proposes one dense-layer gradient

D DERIVATION

Reuse the single-example rules on example 1, before doing any batch algebra.

TO $\boldsymbol{W}$

$$\boldsymbol{G}_W^1 = \boldsymbol{r}^1 (\boldsymbol{x}^1)^\top = \begin{pmatrix} 8 & -4 \\ -4 & 2 \end{pmatrix}$$

TO $\boldsymbol{b}$

$$\boldsymbol{g}_b^1 = \boldsymbol{r}^1 = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

TO $\boldsymbol{x}^1$

$$\boldsymbol{g}_x^1 = \boldsymbol{W}^\top \boldsymbol{r}^1 = \begin{pmatrix} 8 \\ 10 \end{pmatrix}$$

these are unscaled example-1 returns; its contribution to the mean loss is each value divided by 3

Shared parameters add, then the mean scales

D DERIVATION

Each example proposes one outer product for the **same** W. Shared paths add; the mean loss scales the final sum.

Shared parameters add, then the mean scales

D DERIVATION

Each example proposes one outer product for the **same** $\boldsymbol{W}$. Shared paths add; the mean loss scales the final sum.

EXAMPLE 1

$$\boldsymbol{G}_W^1 = \begin{pmatrix} 8 & -4 \\ -4 & 2 \end{pmatrix}$$

EXAMPLE 2

$$\boldsymbol{G}_W^2 = \begin{pmatrix} 2 & -4 \\ -4 & 8 \end{pmatrix}$$

EXAMPLE 3

$$\boldsymbol{G}_W^3 = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$$

Shared parameters add, then the mean scales

D DERIVATION

Each example proposes one outer product for the **same** W. Shared paths add; the mean loss scales the final sum.

EXAMPLE 1

$$G_W^1 = \begin{pmatrix} 8 & -4 \\ -4 & 2 \end{pmatrix}$$

EXAMPLE 2

$$G_W^2 = \begin{pmatrix} 2 & -4 \\ -4 & 8 \end{pmatrix}$$

EXAMPLE 3

$$G_W^3 = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$$

$$G_W^1 + G_W^2 + G_W^3 = \begin{pmatrix} 12 & -6 \\ -6 & 12 \end{pmatrix}$$

Shared parameters add, then the mean scales

D DERIVATION

Each example proposes one outer product for the **same** W. Shared paths add; the mean loss scales the final sum.

EXAMPLE 1

$$G_W^1 = \begin{pmatrix} 8 & -4 \\ -4 & 2 \end{pmatrix}$$

EXAMPLE 2

$$G_W^2 = \begin{pmatrix} 2 & -4 \\ -4 & 8 \end{pmatrix}$$

EXAMPLE 3

$$G_W^3 = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$$

$$G_W^1 + G_W^2 + G_W^3 = \begin{pmatrix} 12 & -6 \\ -6 & 12 \end{pmatrix}$$

MEAN WEIGHT GRADIENT

$$g_W = \frac{1}{3}\begin{pmatrix} 12 & -6 \\ -6 & 12 \end{pmatrix} = \begin{pmatrix} 4 & -2 \\ -2 & 4 \end{pmatrix}$$

MEAN BIAS GRADIENT

$$g_b = \frac{1}{3}\left(\begin{pmatrix} 4 \\ -2 \end{pmatrix} + \begin{pmatrix} -2 \\ 4 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Batch matrix rules stack the same arithmetic

D DERIVATION

Let $\boldsymbol{G}_Z = \frac{\boldsymbol{R}}{B} \in \mathbb{R}^{B\times m}$. The formulas below package the rowwise returns; they do not replace the derivation.

Batch matrix rules stack the same arithmetic

D DERIVATION

Let $\boldsymbol{G}_Z = \frac{\boldsymbol{R}}{B} \in \mathbb{R}^{B\times m}$. The formulas below package the rowwise returns; they do not replace the derivation.

Recipient	Stacked rule	Operand shapes
$\boldsymbol{W}$	$\boldsymbol{g}_W = \boldsymbol{G}_Z^\top \boldsymbol{X}$	$(m \times d) = (m \times B)(B \times d)$
$\boldsymbol{b}$	$\boldsymbol{g}_b = \boldsymbol{G}_Z^\top \mathbf{1}_B$	$(m \times 1) = (m \times B)(B \times 1)$
$\boldsymbol{X}$	$\boldsymbol{g}_X = \boldsymbol{G}_Z \boldsymbol{W}$	$(B \times d) = (B \times m)(m \times d)$

Batch matrix rules stack the same arithmetic

D DERIVATION

Let $\boldsymbol{G}_Z = \frac{\boldsymbol{R}}{B} \in \mathbb{R}^{B\times m}$. The formulas below package the rowwise returns; they do not replace the derivation.

Recipient	Stacked rule	Operand shapes
$\boldsymbol{W}$	$g_W = \boldsymbol{G}_Z^\top \boldsymbol{X}$	$(m \times d) = (m \times B)(B \times d)$
$\boldsymbol{b}$	$g_b = \boldsymbol{G}_Z^\top \mathbf{1}_B$	$(m \times 1) = (m \times B)(B \times 1)$
$\boldsymbol{X}$	$g_X = \boldsymbol{G}_Z \boldsymbol{W}$	$(B \times d) = (B \times m)(m \times d)$

$$g_X = \frac{1}{3}\begin{pmatrix} 8 & 10 \\ -10 & -2 \\ -1 & 4 \end{pmatrix}$$

Batch matrix rules stack the same arithmetic

D DERIVATION

Let $\boldsymbol{G}_Z = \frac{\boldsymbol{R}}{B} \in \mathbb{R}^{B\times m}$. The formulas below package the rowwise returns; they do not replace the derivation.

Recipient	Stacked rule	Operand shapes
$\boldsymbol{W}$	$g_W = \boldsymbol{G}_Z^\top \boldsymbol{X}$	$(m\times d) = (m\times B)(B\times d)$
$\boldsymbol{b}$	$g_b = \boldsymbol{G}_Z^\top \mathbf{1}_B$	$(m\times 1) = (m\times B)(B\times 1)$
$\boldsymbol{X}$	$g_X = \boldsymbol{G}_Z \boldsymbol{W}$	$(B\times d) = (B\times m)(m\times d)$

$$g_X = \frac{1}{3}\begin{pmatrix} 8 & 10 \\ -10 & -2 \\ -1 & 4 \end{pmatrix}$$

> $\boldsymbol{W}, \boldsymbol{b}$ are shared, so their example paths add. The three input rows are distinct, so their gradients stay in three separate rows.

Sum or mean? The reduction chooses the scale

D DERIVATION

SUM OVER EXAMPLES

$$\mathcal{L}_{\text{sum}} = \sum_n \ell_n$$
$$\boldsymbol{G}_Z = \boldsymbol{R}$$
$$\boldsymbol{g}_W = \boldsymbol{R}^\top \boldsymbol{X}$$

MEAN OVER EXAMPLES

$$\mathcal{L}_{\text{mean}} = \frac{1}{B} \sum_n \ell_n$$
$$\boldsymbol{G}_Z = \frac{\boldsymbol{R}}{B}$$
$$\boldsymbol{g}_W = \boldsymbol{R}^\top \frac{\boldsymbol{X}}{B}$$

Sum or mean? The reduction chooses the scale

D DERIVATION

SUM OVER EXAMPLES

$$\mathcal{L}_{\text{sum}} = \sum_n \ell_n$$
$$\boldsymbol{G}_Z = \boldsymbol{R}$$
$$\boldsymbol{g}_W = \boldsymbol{R}^\top \boldsymbol{X}$$

MEAN OVER EXAMPLES

$$\mathcal{L}_{\text{mean}} = \frac{1}{B} \sum_n \ell_n$$
$$\boldsymbol{G}_Z = \frac{\boldsymbol{R}}{B}$$
$$\boldsymbol{g}_W = \boldsymbol{R}^\top \frac{\boldsymbol{X}}{B}$$

```
loss = 0.5 * ((Z - Y)**2).sum(dim=1).mean()
# sum output coordinates inside each example; mean over examples
```

Sum or mean? The reduction chooses the scale

D DERIVATION

SUM OVER EXAMPLES

$$\mathcal{L}_{\text{sum}} = \sum_n \ell_n$$
$$\boldsymbol{G}_Z = \boldsymbol{R}$$
$$\boldsymbol{g}_W = \boldsymbol{R}^\top \boldsymbol{X}$$

MEAN OVER EXAMPLES

$$\mathcal{L}_{\text{mean}} = \frac{1}{B} \sum_n \ell_n$$
$$\boldsymbol{G}_Z = \frac{\boldsymbol{R}}{B}$$
$$\boldsymbol{g}_W = \boldsymbol{R}^\top \frac{\boldsymbol{X}}{B}$$

```
loss = 0.5 * ((Z - Y)**2).sum(dim=1).mean()
# sum output coordinates inside each example; mean over examples
```

the dense local rules do not change; the scalar loss reduction chooses their final scale

PyTorch verifies the single example and the batch

INTERACTIVE

```
X = torch.tensor([[ 2., -1.], [-1., 2.], [2., 2.]],
                 requires_grad=True)
Y = torch.tensor([[-5., -2.], [ 7., 1.], [7., -2.]])
W = torch.tensor([[1., 3.], [-2., 1.]], requires_grad=True)
b = torch.tensor([0., 1.], requires_grad=True)

Z = X @ W.T + b
loss = 0.5 * ((Z - Y)**2).sum(dim=1).mean()
loss.backward()

print(loss.item())  # 7.0
print(W.grad)        # [[ 4., -2.], [-2.,  4.]]
print(b.grad)        # [1., 1.]
print(X.grad)        # [[ 8/3, 10/3], [-10/3, -2/3], [-1/3, 4/3]]
```

PyTorch verifies the single example and the batch

I INTERACTIVE

```
X = torch.tensor([[ 2., -1.], [-1., 2.], [2., 2.]],
                 requires_grad=True)
Y = torch.tensor([[-5., -2.], [ 7., 1.], [7., -2.]])
W = torch.tensor([[1., 3.], [-2., 1.]], requires_grad=True)
b = torch.tensor([0., 1.], requires_grad=True)

Z = X @ W.T + b
loss = 0.5 * ((Z - Y)**2).sum(dim=1).mean()
loss.backward()

print(loss.item())  # 7.0
print(W.grad)        # [[ 4., -2.], [-2.,  4.]]
print(b.grad)        # [1., 1.]
print(X.grad)        # [[ 8/3, 10/3], [-10/3, -2/3], [-1/3, 4/3]]
```

> Continue in the complete lecture Colab ↗: checkpoints 11–16 derive the dense VJP, expose every batch contribution, verify microbatches, run one update, and finish with the tiny MLP.

The training loop now has a concrete batch

INTERACTIVE

```
for X, Y in loader:
    optimizer.zero_grad()       # clear the previous update's buffers
    Z = model(X)                # all examples reuse the same parameters
    loss = loss_fn(Z, Y)        # reduce example losses to one scalar
    loss.backward()             # aggregate this batch's contributions
    optimizer.step()            # update once
```

The training loop now has a concrete batch

I INTERACTIVE

```
for X, Y in loader:
    optimizer.zero_grad()       # clear the previous update's buffers
    Z = model(X)                # all examples reuse the same parameters
    loss = loss_fn(Z, Y)        # reduce example losses to one scalar
    loss.backward()             # aggregate this batch's contributions
    optimizer.step()            # update once
```

`zero_grad()`	starts one intentional update window
`model(X)`	records one graph whose rows share the same parameters
`loss.backward()`	returns the reduction’s sum or mean parameter gradient
`step()`	uses that gradient; it neither computes nor clears it

The training loop now has a concrete batch

I INTERACTIVE

```
for X, Y in loader:
    optimizer.zero_grad()        # clear the previous update's buffers
    Z = model(X)                 # all examples reuse the same parameters
    loss = loss_fn(Z, Y)         # reduce example losses to one scalar
    loss.backward()              # aggregate this batch's contributions
    optimizer.step()             # update once
```

`zero_grad()`	starts one intentional update window
`model(X)`	records one graph whose rows share the same parameters
`loss.backward()`	returns the reduction’s sum or mean parameter gradient
`step()`	uses that gradient; it neither computes nor clears it

clear → batch forward → scalar loss → batch backward → update

One batch can arrive in pieces

Microbatches can reproduce the same mean gradient

D DERIVATION

ONE FULL BATCH

```
zero_grad()
((ell1 + ell2 + ell3) /
3).backward()
step()
```

THREE PIECES · ONE UPDATE

```
zero_grad()
(ell1 / 3).backward()
(ell2 / 3).backward()
(ell3 / 3).backward()
step()
```

Microbatches can reproduce the same mean gradient

D DERIVATION

ONE FULL BATCH

```
zero_grad()
((ell1 + ell2 + ell3) /
3).backward()
step()
```

THREE PIECES · ONE UPDATE

```
zero_grad()
(ell1 / 3).backward()
(ell2 / 3).backward()
(ell3 / 3).backward()
step()
```

$$\boldsymbol{W}.\mathrm{grad}: \quad \frac{\boldsymbol{G}_W^1}{3} \to \frac{\boldsymbol{G}_W^1+\boldsymbol{G}_W^2}{3} \to \begin{pmatrix} 4 & -2 \\ -2 & 4 \end{pmatrix}$$

Microbatches can reproduce the same mean gradient

D DERIVATION

ONE FULL BATCH

```
zero_grad()
((ell1 + ell2 + ell3) /
3).backward()
step()
```

THREE PIECES · ONE UPDATE

```
zero_grad()
(ell1 / 3).backward()
(ell2 / 3).backward()
(ell3 / 3).backward()
step()
```

$$\boldsymbol{W}.\mathrm{grad}: \quad \frac{\boldsymbol{G}_W^1}{3} \to \frac{\boldsymbol{G}_W^1+\boldsymbol{G}_W^2}{3} \to \begin{pmatrix} 4 & -2 \\ -2 & 4 \end{pmatrix}$$

> Equivalence requires fixed parameters, the same overall reduction, no `step()`, and no `zero_grad()` between pieces. Batch-coupled operations such as BatchNorm need extra care.

Microbatches can reproduce the same mean gradient

D DERIVATION

ONE FULL BATCH

```
zero_grad()
((ell1 + ell2 + ell3) /
3).backward()
step()
```

THREE PIECES · ONE UPDATE

```
zero_grad()
(ell1 / 3).backward()
(ell2 / 3).backward()
(ell3 / 3).backward()
step()
```

$$\boldsymbol{W}.\mathrm{grad}: \quad \frac{\boldsymbol{G}_W^1}{3} \to \frac{\boldsymbol{G}_W^1 + \boldsymbol{G}_W^2}{3} \to \begin{pmatrix} 4 & -2 \\ -2 & 4 \end{pmatrix}$$

> Equivalence requires fixed parameters, the same overall reduction, no `step()`, and no `zero_grad()` between pieces. Batch-coupled operations such as BatchNorm need extra care.

clear once before the update window · add every scaled piece · step once after it

A tiny neural network

ReLU is what prevents two dense layers from collapsing into one

V VISUAL

Without a nonlinearity, two affine maps are still one affine map:

$$\boldsymbol{W}_2(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

$$= (\boldsymbol{W}_2\boldsymbol{W}_1)\boldsymbol{x} + (\boldsymbol{W}_2\boldsymbol{b}_1 + \boldsymbol{b}_2).$$

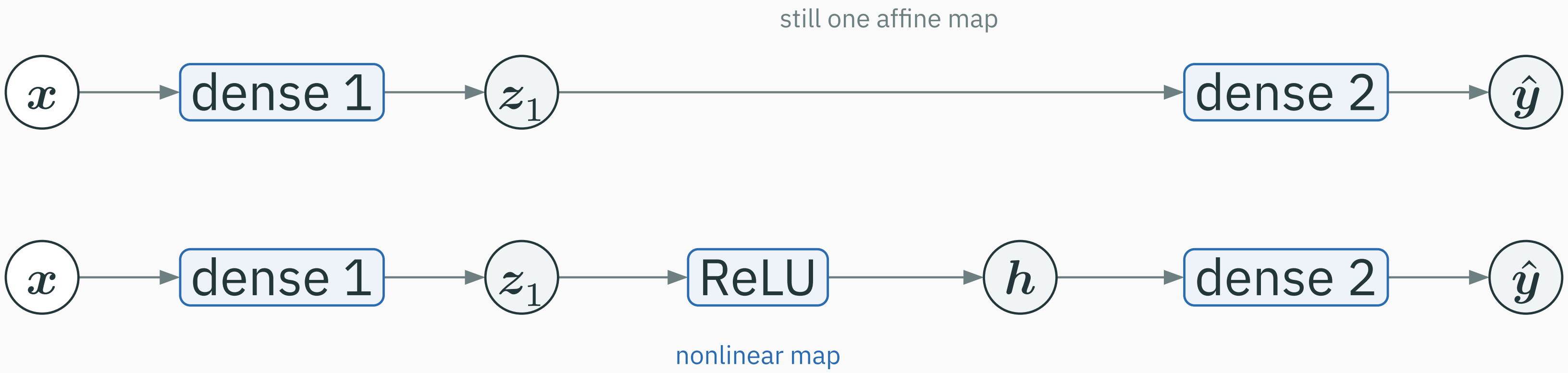
ReLU is what prevents two dense layers from collapsing into one

VISUAL

Without a nonlinearity, two affine maps are still one affine map:

$$\boldsymbol{W}_2(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

$$= (\boldsymbol{W}_2\boldsymbol{W}_1)\boldsymbol{x} + (\boldsymbol{W}_2\boldsymbol{b}_1 + \boldsymbol{b}_2).$$



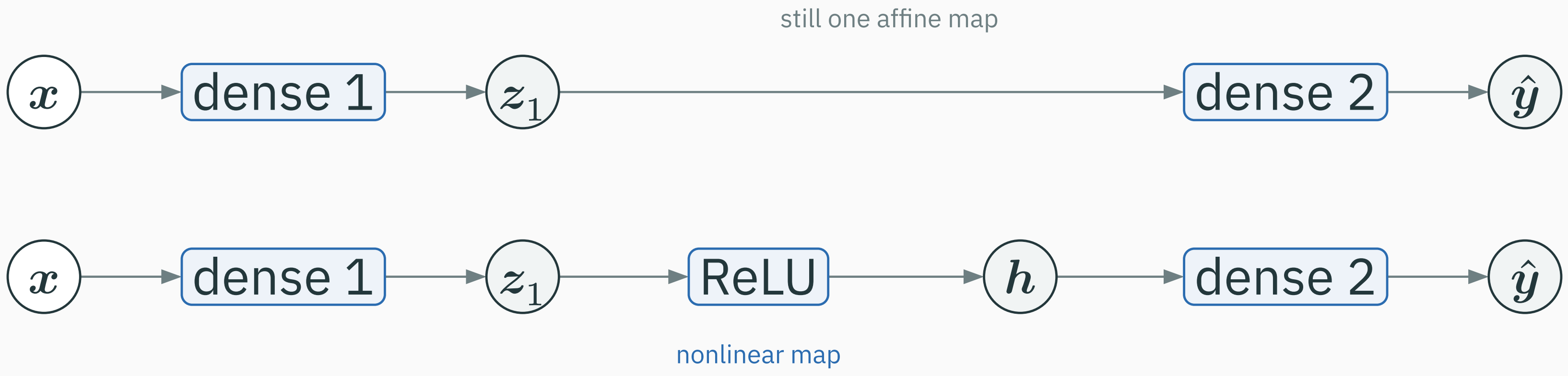
ReLU is what prevents two dense layers from collapsing into one

V VISUAL

Without a nonlinearity, two affine maps are still one affine map:

$$\boldsymbol{W}_2(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

$$= (\boldsymbol{W}_2\boldsymbol{W}_1)\boldsymbol{x} + (\boldsymbol{W}_2\boldsymbol{b}_1 + \boldsymbol{b}_2).$$



ReLU makes the composition nonlinear; depth can now represent a richer function

A tiny MLP composes three vector blocks

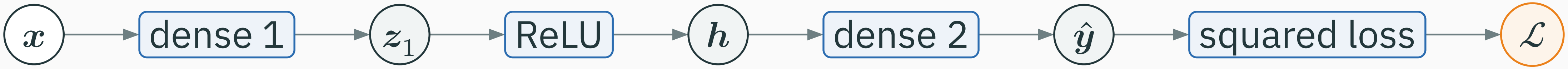
V VISUAL

The network now repeats the operations we already know: affine map, elementwise ReLU, affine map, scalar loss.

A tiny MLP composes three vector blocks

V VISUAL

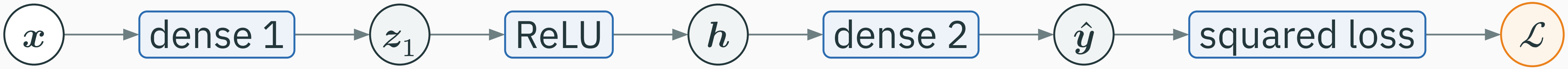
The network now repeats the operations we already know: affine map, elementwise ReLU, affine map, scalar loss.



A tiny MLP composes three vector blocks

V VISUAL

The network now repeats the operations we already know: affine map, elementwise ReLU, affine map, scalar loss.



$$\boldsymbol{z}_1 = \boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1$$

$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{z}_1)$$

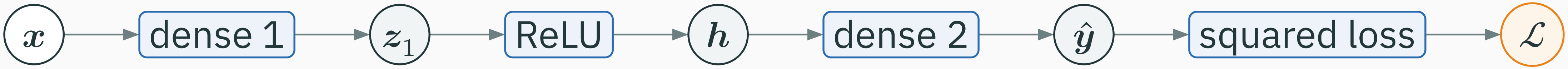
$$\hat{\boldsymbol{y}} = \boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2$$

$$\mathcal{L} = \sum_i (\hat{y}_i - y_i)^2$$

A tiny MLP composes three vector blocks

V VISUAL

The network now repeats the operations we already know: affine map, elementwise ReLU, affine map, scalar loss.



$$\boldsymbol{z}_1 = \boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1$$

$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{z}_1)$$

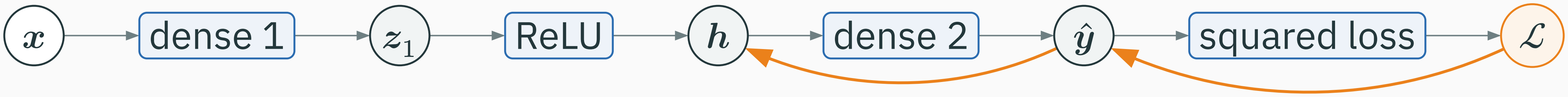
$$\hat{\boldsymbol{y}} = \boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2$$

$$\mathcal{L} = \sum_i (\hat{y}_i - y_i)^2$$

dense → ReLU → dense → squared loss

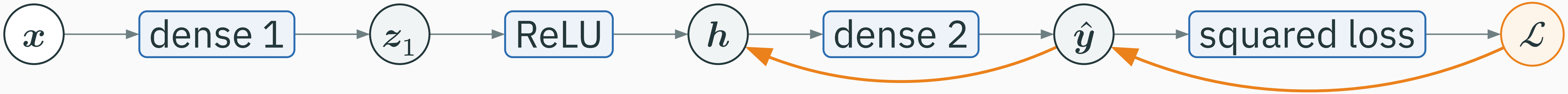
Backward, part 1: loss → dense 2 → hidden activations

D DERIVATION



Backward, part 1: loss → dense 2 → hidden activations

D DERIVATION



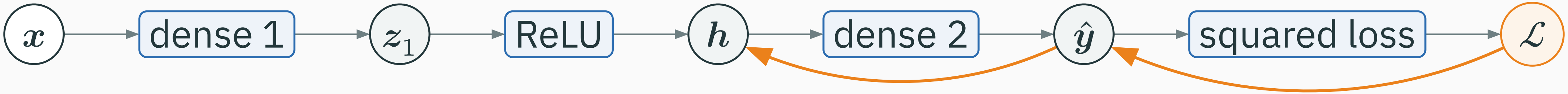
1. Squared loss. Start with the seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$. The loss operation returns

$$\boldsymbol{g}_{\hat{y}} = 2(\hat{\boldsymbol{y}} - \boldsymbol{y}).$$

This says how increasing each prediction coordinate would change the loss.

Backward, part 1: loss → dense 2 → hidden activations

D DERIVATION



1. Squared loss. Start with the seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$. The loss operation returns

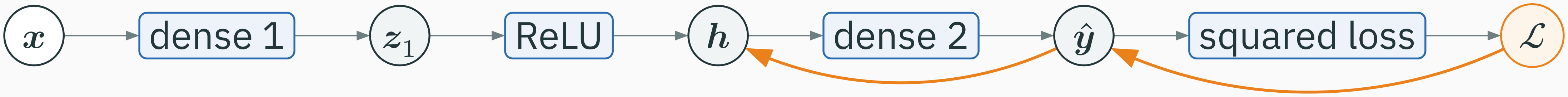
$$\boldsymbol{g}_{\hat{y}} = 2(\hat{\boldsymbol{y}} - \boldsymbol{y}).$$

This says how increasing each prediction coordinate would change the loss. **2. Second dense layer.** It receives $\boldsymbol{g}_{\hat{y}}$ and returns a gradient to its input $\boldsymbol{h}$:

$$\boldsymbol{g}_h = \boldsymbol{W}_2^\top \boldsymbol{g}_{\hat{y}}.$$

Backward, part 1: loss → dense 2 → hidden activations

D DERIVATION



1. Squared loss. Start with the seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$. The loss operation returns

$$\boldsymbol{g}_{\hat{y}} = 2(\hat{\boldsymbol{y}} - \boldsymbol{y}).$$

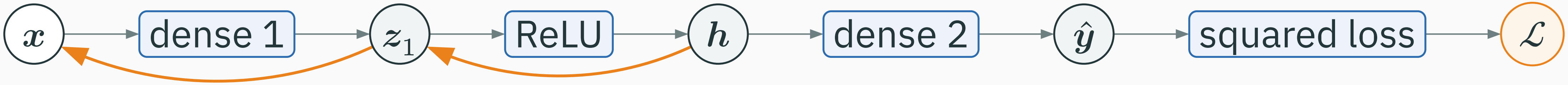
This says how increasing each prediction coordinate would change the loss. **2. Second dense layer.** It receives $\boldsymbol{g}_{\hat{y}}$ and returns a gradient to its input $\boldsymbol{h}$:

$$\boldsymbol{g}_h = \boldsymbol{W}_2^\top \boldsymbol{g}_{\hat{y}}.$$

the returned g_h is now the arriving gradient at ReLU

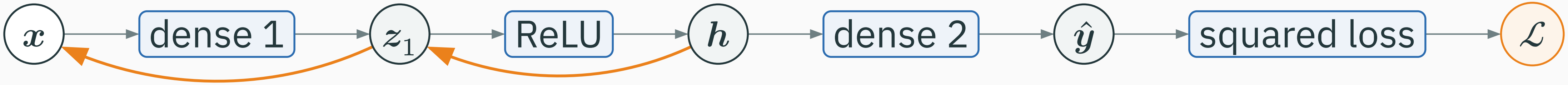
Backward, part 2: ReLU → dense 1 → input

D DERIVATION



Backward, part 2: ReLU → dense 1 → input

D DERIVATION



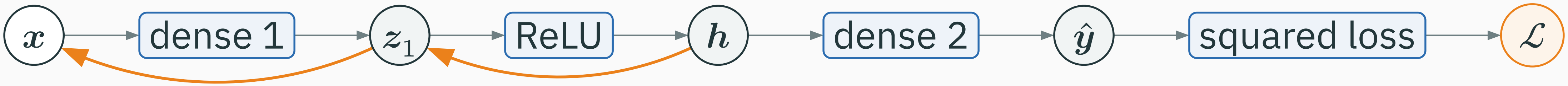
3. ReLU. Apply its local gate coordinate by coordinate:

$$\boldsymbol{g}_{z_1} = \boldsymbol{g}_h \odot \mathbb{1}[\boldsymbol{z}_1 > 0].$$

An active unit passes its arriving gradient; an inactive unit returns zero.

Backward, part 2: ReLU → dense 1 → input

D DERIVATION



3. ReLU. Apply its local gate coordinate by coordinate:

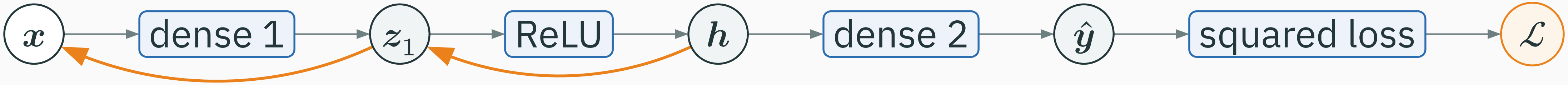
$$\boldsymbol{g}_{z_1} = \boldsymbol{g}_h \odot \mathbb{1}[\boldsymbol{z}_1 > 0].$$

An active unit passes its arriving gradient; an inactive unit returns zero. **4. First dense layer.** It receives $\boldsymbol{g}_{z_1}$ and returns

$$\boldsymbol{g}_x = \boldsymbol{W}_1^\top \boldsymbol{g}_{z_1}, \qquad \partial\mathcal{L}/\partial\boldsymbol{W}_1 = \boldsymbol{g}_{z_1}\boldsymbol{x}^\top, \qquad \partial\mathcal{L}/\partial\boldsymbol{b}_1 = \boldsymbol{g}_{z_1}.$$

Backward, part 2: ReLU → dense 1 → input

D DERIVATION



3. ReLU. Apply its local gate coordinate by coordinate:

$$\boldsymbol{g}_{z_1} = \boldsymbol{g}_h \odot \mathbb{1}[\boldsymbol{z}_1 > 0].$$

An active unit passes its arriving gradient; an inactive unit returns zero. **4. First dense layer.** It receives $\boldsymbol{g}_{z_1}$ and returns

$$\boldsymbol{g}_x = \boldsymbol{W}_1^\top \boldsymbol{g}_{z_1}, \qquad \partial\mathcal{L}/\partial\boldsymbol{W}_1 = \boldsymbol{g}_{z_1}\boldsymbol{x}^\top, \qquad \partial\mathcal{L}/\partial\boldsymbol{b}_1 = \boldsymbol{g}_{z_1}.$$

each returned gradient becomes the arriving gradient for the block immediately to its left

The same tiny MLP in PyTorch

I INTERACTIVE

```
import torch
from torch import nn

model = nn.Sequential(
    nn.Linear(2, 3), nn.ReLU(), nn.Linear(3, 2)
)
x = torch.tensor([2., -1.])
y = torch.tensor([1., 0.])

y_hat = model(x)
loss = ((y_hat - y)**2).sum()
loss.backward()

for name, p in model.named_parameters():
    print(name, tuple(p.shape), tuple(p.grad.shape))
```

The same tiny MLP in PyTorch

INTERACTIVE

```
import torch
from torch import nn

model = nn.Sequential(
    nn.Linear(2, 3), nn.ReLU(), nn.Linear(3, 2)
)
x = torch.tensor([2., -1.])
y = torch.tensor([1., 0.])

y_hat = model(x)
loss = ((y_hat - y)**2).sum()
loss.backward()

for name, p in model.named_parameters():
    print(name, tuple(p.shape), tuple(p.grad.shape))
```

PyTorch records the three blocks and applies their local rules in reverse

Interactive: follow one vector gradient

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/autograd

Step through the forward values, then follow one arriving gradient through a dense operation, a ReLU, and the preceding dense operation.

Interactive: follow one vector gradient

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/autograd

Step through the forward values, then follow one arriving gradient through a dense operation, a ReLU, and the preceding dense operation.

For practice, predict each gradient's shape before revealing its value in the interactive.

When gradients cross many layers

Deep networks repeat the same multiplication

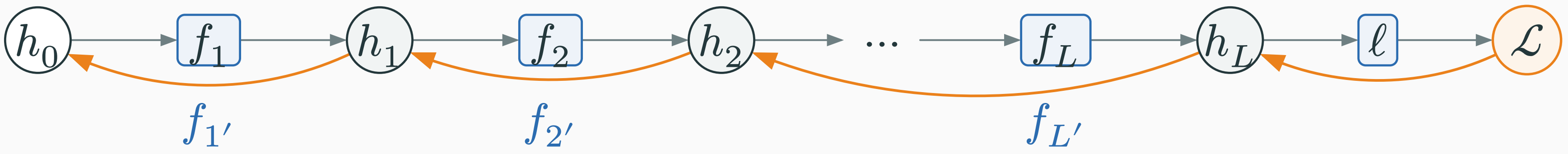
D DERIVATION

For a scalar chain $h_0 \to h_1 \to \ldots \to h_L \to \mathcal{L}$:

Deep networks repeat the same multiplication

D DERIVATION

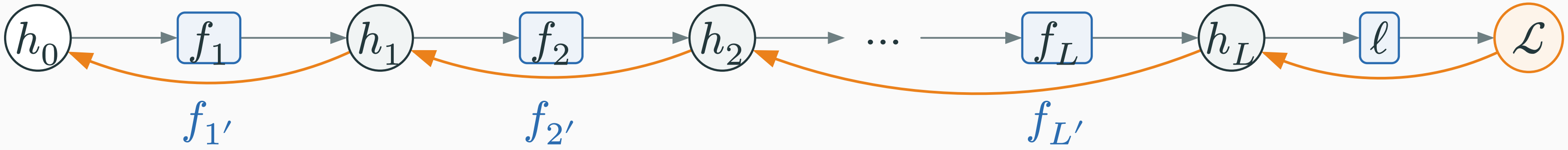
For a scalar chain $h_0 \to h_1 \to \ldots \to h_L \to \mathcal{L}$:



Deep networks repeat the same multiplication

D DERIVATION

For a scalar chain $h_0 \to h_1 \to \ldots \to h_L \to \mathcal{L}$:

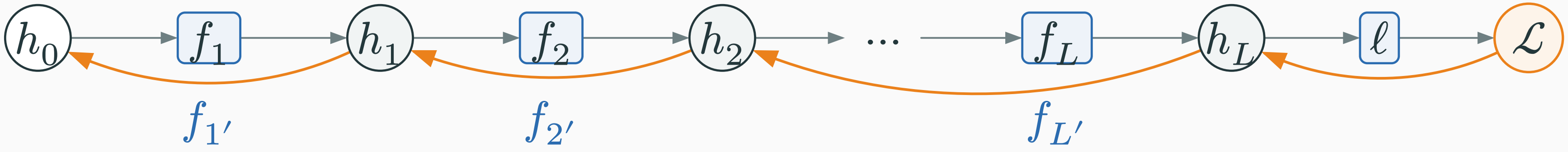


$$\frac{\partial \mathcal{L}}{\partial h_0} = \frac{\partial \mathcal{L}}{\partial h_L} \cdot f_{L'}(h_{L-1}) \ldots f_{1'}(h_0).$$

Deep networks repeat the same multiplication

D DERIVATION

For a scalar chain $h_0 \to h_1 \to \ldots \to h_L \to \mathcal{L}$:



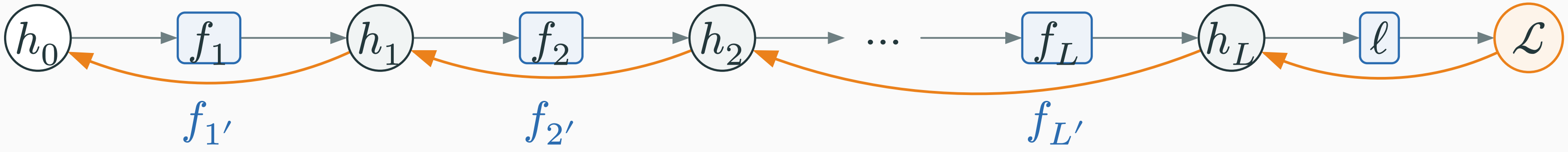
$$\frac{\partial \mathcal{L}}{\partial h_0} = \frac{\partial \mathcal{L}}{\partial h_L} \cdot f_{L'}(h_{L-1}) \ldots f_{1'}(h_0).$$

depth creates a product of many local slopes

Deep networks repeat the same multiplication

D DERIVATION

For a scalar chain $h_0 \to h_1 \to \ldots \to h_L \to \mathcal{L}$:



$$\frac{\partial \mathcal{L}}{\partial h_0} = \frac{\partial \mathcal{L}}{\partial h_L} \cdot f_{L'}(h_{L-1}) \ldots f_{1'}(h_0).$$

depth creates a product of many local slopes

> Vector layers do the same operation with matrix–vector products. The intuition is unchanged: repeatedly small factors shrink a gradient; repeatedly large factors grow it.

Ten ordinary-looking slopes can create three regimes

V VISUAL

Start with an arriving loss derivative of 1 and multiply the same local slope ten times:

Ten ordinary-looking slopes can create three regimes

V VISUAL

Start with an arriving loss derivative of 1 and multiply the same local slope ten times:

Local slope	After 10 layers	Effect
0.5	$0.5^{10} \approx 0.001$	gradient nearly disappears
1.0	$1^{10} = 1$	gradient is preserved
1.5	$1.5^{10} \approx 57.7$	gradient grows rapidly

Ten ordinary-looking slopes can create three regimes

V VISUAL

Start with an arriving loss derivative of 1 and multiply the same local slope ten times:

Local slope	After 10 layers	Effect
0.5	$0.5^{10} \approx 0.001$	gradient nearly disappears
1.0	$1^{10} = 1$	gradient is preserved
1.5	$1.5^{10} \approx 57.7$	gradient grows rapidly

the chain rule explains both vanishing and exploding gradients

Sigmoid saturation

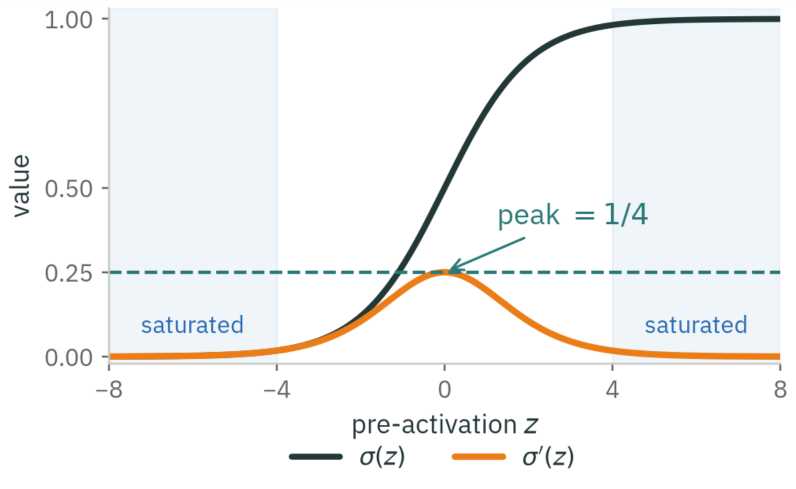
V VISUAL

$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \le \frac{1}{4}$, and ≈ 0 for large $|z|$:

Sigmoid saturation

V VISUAL

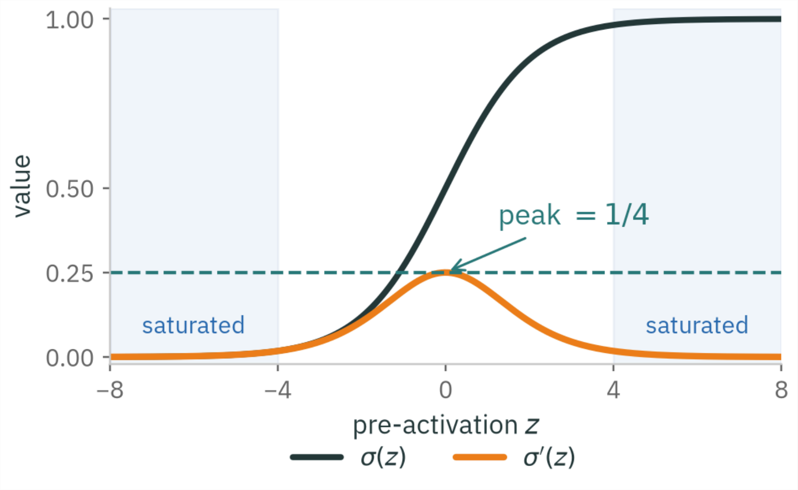
$\sigma'(z) = \sigma(z)(1-\sigma(z)) \leq \frac{1}{4}$, and ≈ 0 for large $|z|$:



Sigmoid saturation

V VISUAL

$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \leq \frac{1}{4}$, and ≈ 0 for large $|z|$:

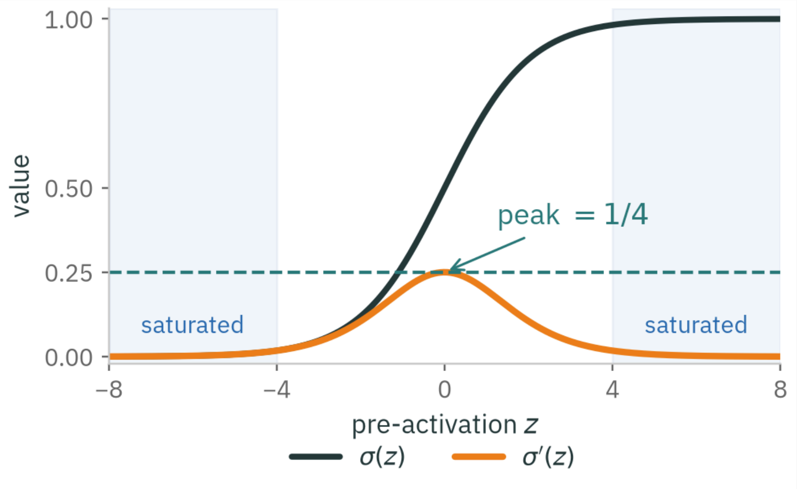


each sigmoid layer multiplies the gradient by at most $1/4$

Sigmoid saturation

V VISUAL

$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \leq \frac{1}{4}$, and ≈ 0 for large $|z|$:



each sigmoid layer multiplies the gradient by at most $1/4$

> Repeated sigmoid derivatives can shrink the signal quickly. ReLU has slope 1 on active units and 0 on inactive units; weights and normalization also shape the full Jacobian.

Interactive: gradient flow

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/vanishing-gradients

Sweep depth and activation function; watch the gradient magnitude at layer 0 collapse for sigmoid and survive for ReLU.

Interactive: gradient flow

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/vanishing-gradients

Sweep depth and activation function; watch the gradient magnitude at layer 0 collapse for sigmoid and survive for ReLU.

Sweep depth to see repeated small local slopes rapidly erase an early-layer gradient.

Summary

Backprop in one sentence

Seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$, then walk the graph **backward**, at each operation computing

downstream gradient $=$ **upstream gradient** $\times$ **local derivative**,

and **adding** gradients wherever a variable branched.

Backprop in one sentence

Seed $\partial\mathcal{L}/\partial\mathcal{L} = 1$, then walk the graph **backward**, at each operation computing

downstream gradient $=$ **upstream gradient** $\times$ **local derivative**,

and **adding** gradients wherever a variable branched.

For vector operations, the local derivative can be a vector or matrix; the same color-coded chain rule applies.

The formulas to keep

V VISUAL

Setting	**Backward rule**
scalar operation	$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot \partial v / \partial u$
dense layer	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} \boldsymbol{x}^\top \; \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \boldsymbol{W}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$
activation	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}} \odot \varphi'(\boldsymbol{z})$
branch point	gradients **add**

We can compute $\partial\mathcal{L}/\partial\theta_j$ for every parameter.

Next: turn those gradients into reliable parameter updates.

Lecture 5 · Optimization for Deep Learning