

Automatic differentiation

Lecture 1A

Mikhail Mikhasenko

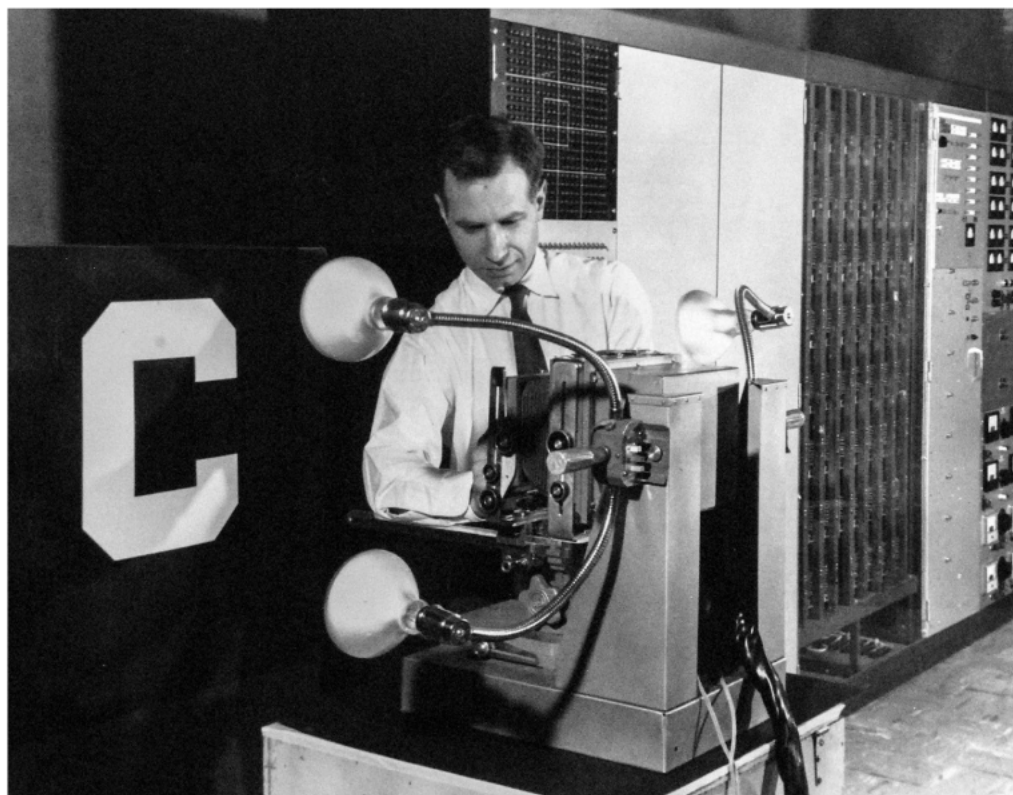
Why automatic differentiation?

From early neural networks to the need for reliable
derivatives

Mark I Perceptron

to reporters by Frank Rosenblatt. July 7, 1958

1958, The New York Times



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo
of Computer Designed to
Read and Grow Wiser

WASHINGTON, July 7 (UPI)
—The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's \$2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of \$100,000.

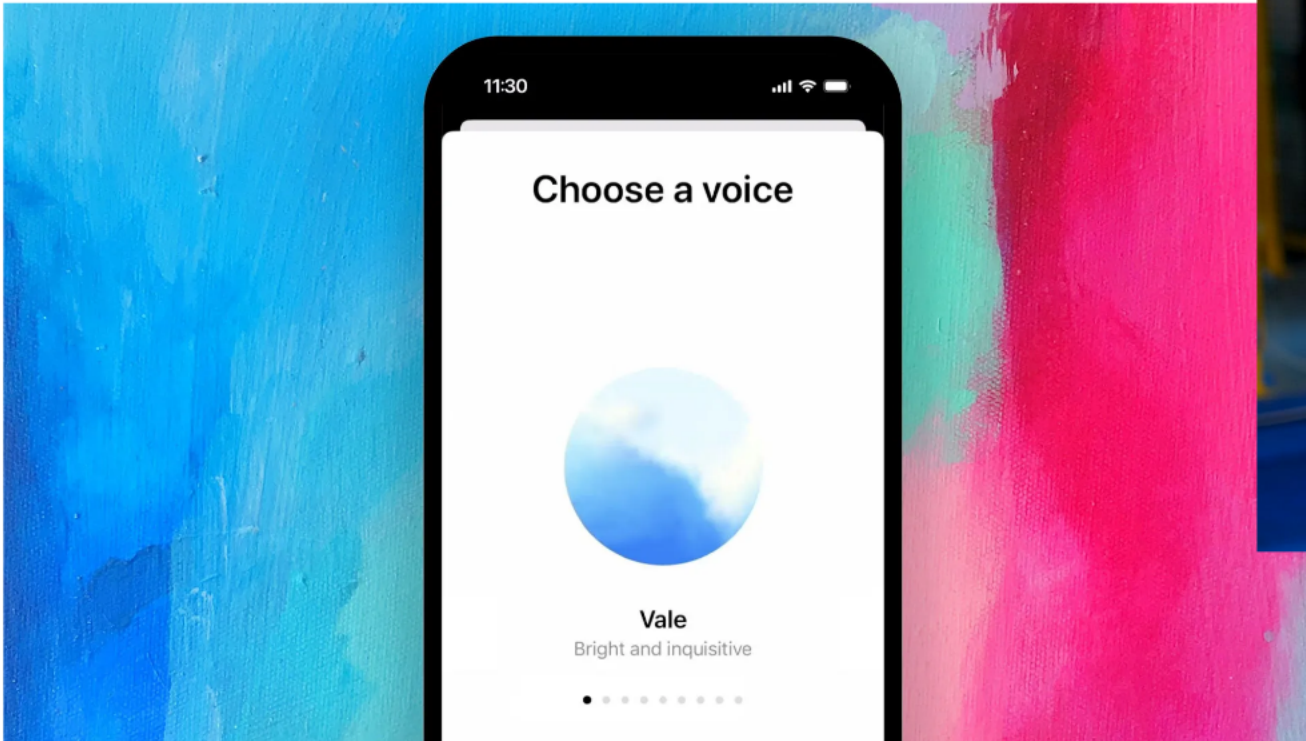
Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.



Expected at ~1960 happened in 2025

- Walk, talk, see, and write

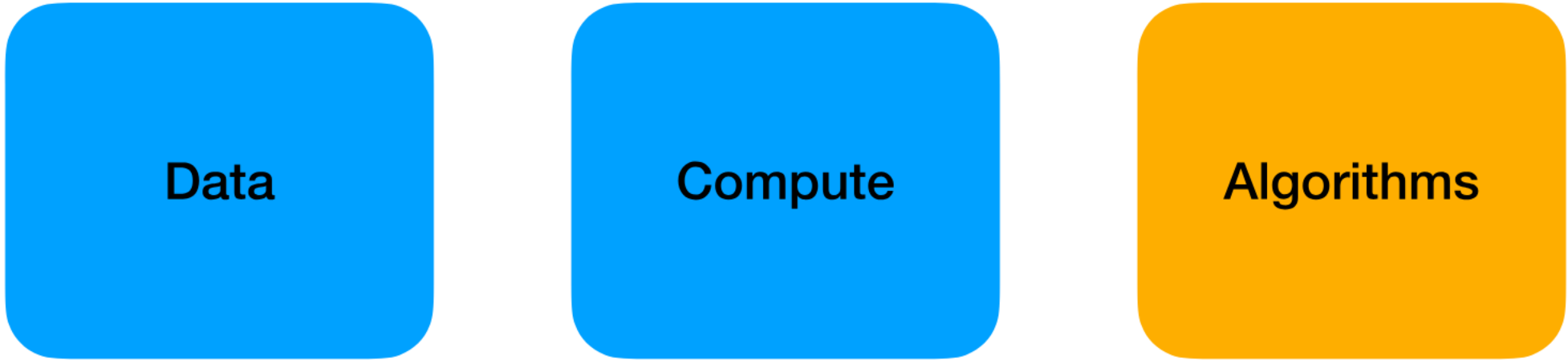


chatGPT voice mode



Robot Atlas by Boston Dynamics

What held the progress?



The diagram consists of three rounded rectangular boxes arranged horizontally. The first two boxes, labeled 'Data' and 'Compute', are blue. The third box, labeled 'Algorithms', is orange. All boxes have black text centered within them.

Data

Compute

Algorithms

Ways to compute derivatives

Problem formulation, symbolic differentiation, and
finite differences

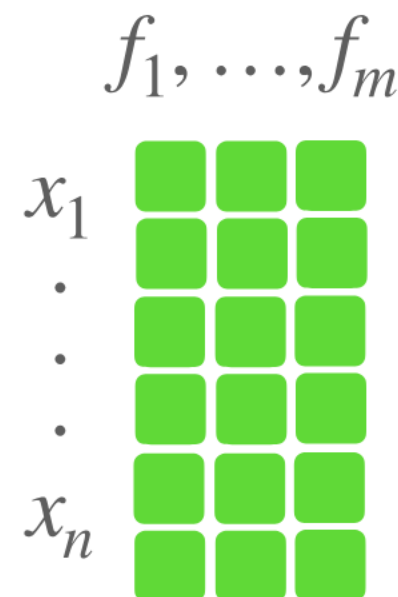
Diff: Problem formulation

- Function $f_1(x) : \vec{x} \in R^n \xrightarrow{f_1} R$ (scalar)
 $R^n \xrightarrow{f} R^m \quad f_1(x), \dots, f_m(x)$ (multiple)

- Derivative $\frac{\partial f_1}{\partial x_1}$

- Gradient $(\frac{\partial f_1}{\partial x_1}, \frac{\partial f_1}{\partial x_2}, \dots, \frac{\partial f_1}{\partial x_n})$

- Jacobian $\frac{\partial(f_1, \dots, f_m)}{\partial(x_1, \dots, x_n)}$



Several ways to get derivative

Symbolic, Numeric, Automatic

$$f(x) = \cos(x) - \log\left(\frac{x^5}{1+x^5}\right) + (3x + x^2)^2$$

$$\frac{df}{dx} = ?$$

Symbolic

Numeric

"Automatic"

Analytic Diff

Warm up

- Follow standard rules compute a derivative

```
• example_f(x) = (x^2 + 3x)^2 - log(x^5 / (1 + x^5)) + cos(x);
```

expr =

$$\cos(x) - \log\left(\frac{x^5}{1+x^5}\right) + (3x + x^2)^2$$

$$-\sin(x) + \frac{-\left(\frac{5x^4}{1+x^5} - 5x^4 \frac{x^5}{(1+x^5)^2}\right)(1+x^5)}{x^5} + 2(3+2x)(3x+x^2)$$

```
• scalar_diff(expr, x)
```

Rule Name	Function	Derivative
Constant Rule	c	0
Constant Multiple Rule	$c \cdot f(x)$	$c \cdot f'(x)$
Power Rule	x^n	$n \cdot x^{n-1}$
Sum Rule	$f(x) + g(x)$	$f'(x) + g'(x)$
Difference Rule	$f(x) - g(x)$	$f'(x) - g'(x)$
Product Rule	$f(x) \cdot g(x)$	$f'(x) \cdot g(x) + f(x) \cdot g'(x)$
Quotient Rule	$\frac{f(x)}{g(x)}$	$\frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{[g(x)]^2}$
Chain Rule	$f(g(x))$	$f'(g(x)) \cdot g'(x)$
Euler's Number Exponent Rule	e^x	e^x
Constant Exponent Rule	a^x	$a^x \cdot \ln(a)$
Natural Log Rule	$\ln(x)$	$\frac{1}{x}$
Logarithm Rule	$\log_a(x)$	$\frac{1}{x \cdot \ln(a)}$
Sine Rule	$\sin(x)$	$\cos(x)$
Cosine Rule	$\cos(x)$	$-\sin(x)$
Tangent Rule	$\tan(x)$	$\sec^2(x)$
Cosecant Rule	$\csc(x)$	$-\csc(x) \cdot \cot(x)$
Secant Rule	$\sec(x)$	$\sec(x) \cdot \tan(x)$
Cotangent Rule	$\cot(x)$	$-\csc^2(x)$

Finite Diff

Choose right answer

$$\frac{df}{dx} \approx \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}$$

- `num_dfdx(f,x1,x2) = (f(x1)-f(x2)) / (x1-x2);`

15.693147180559944

- `example_f(1.0)`

A

39.12720209285954

- `num_dfdx(example_f, 1.5, 0.5)`

B

37.50060420072195

- `num_dfdx(example_f, 1.01, 0.99)`

C

37.50000604167019

- `num_dfdx(example_f, 1.001, 0.999)`

D

53.333333333333336

- `num_dfdx(example_f, nextfloat(1.0), prevfloat(1.0))`

Forward mode

Propagate values and derivative components
together

Forward Diff

Dual numbers

- Augment each number with a derivative component.
- We use dual numbers: $\hat{x} = x + \dot{x} \varepsilon$, where $\varepsilon^2 = 0$
- Define operations on dual numbers

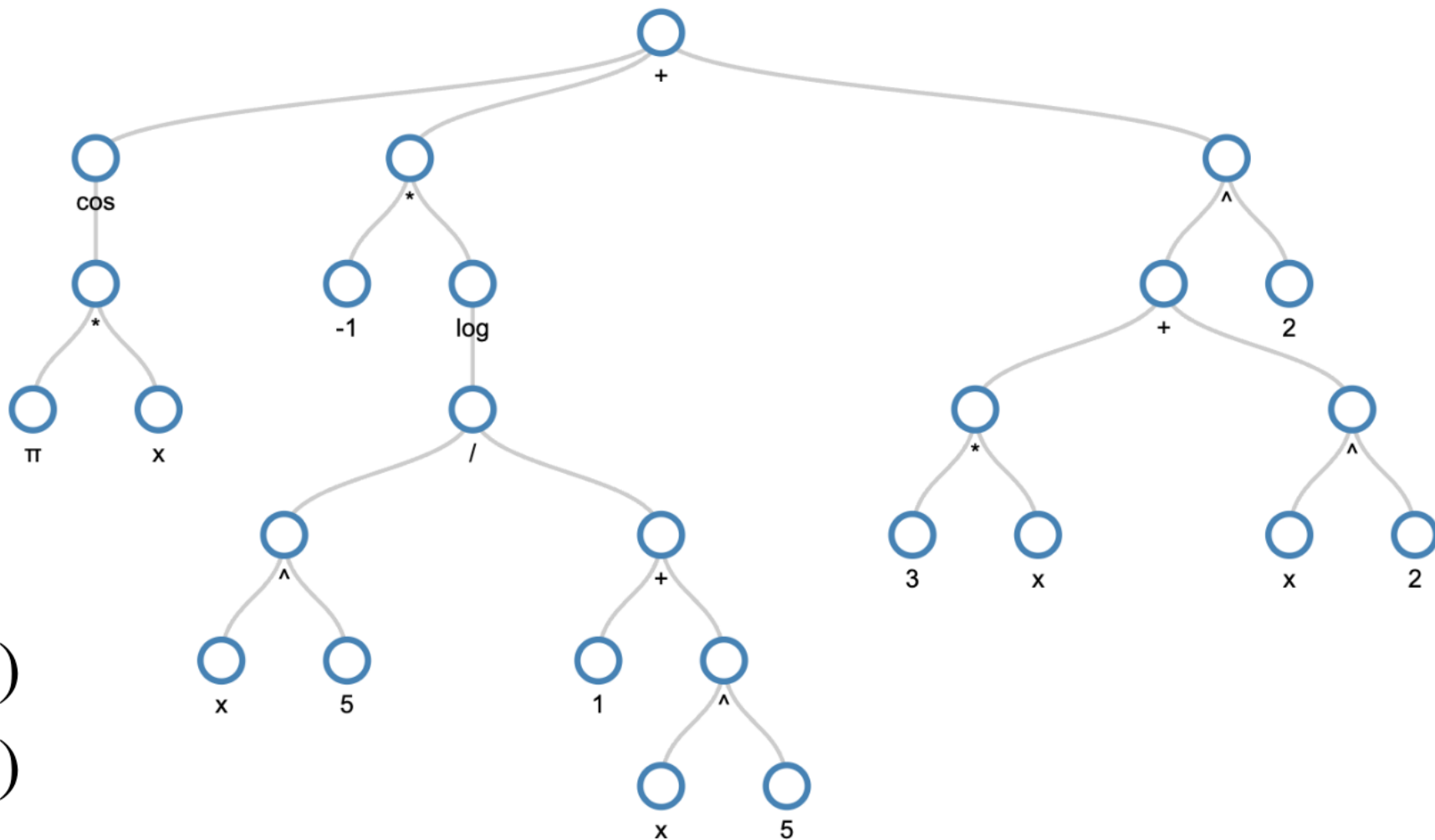
$$f(x + \dot{x}\varepsilon) = f(x) + f'(x)\dot{x} \varepsilon$$

- For higher dims, one column per pass

$$J = \begin{bmatrix} f(x_1 + \varepsilon, x_2, \dots, x_n) & f(x_1, x_2 + \varepsilon, \dots, x_n) \dots \end{bmatrix}$$

Example to work out

$$\cos(x) - \log\left(\frac{x^5}{1+x^5}\right) + (3x + x^2)^2$$



$$c \rightarrow (c, 0)$$

$$x \rightarrow (x, 1)$$

Reverse mode and backpropagation

Evaluate forwards, then propagate sensitivities
backwards

Reverse Diff

- Operate with a computation graph, pulling back effects on the final computation node to the leaves.
- Two operations to understand:
 - * for the chain rule
 - + node combination
- Not expression tree but computation graph

The Chain rule

$$\begin{aligned}
 & \left[\mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{i}(\mathbf{j}(\mathbf{x})))) \right]' \\
 &= \\
 & \mathbf{f}'(\mathbf{g}(\mathbf{h}(\mathbf{i}(\mathbf{j}(\mathbf{x})))) \\
 & \quad * \mathbf{g}'(\mathbf{h}(\mathbf{i}(\mathbf{j}(\mathbf{x})))) \\
 & \quad \quad * \mathbf{h}'(\mathbf{i}(\mathbf{j}(\mathbf{x}))) \\
 & \quad \quad \quad * \mathbf{i}'(\mathbf{j}(\mathbf{x}))
 \end{aligned}$$



Backpropagation, step by step

We want the derivative of one loss with respect to every parameter.

$$L(\boldsymbol{\theta}), \quad \nabla_{\boldsymbol{\theta}} L = \left(\frac{\partial L}{\partial \theta_1}, \dots, \frac{\partial L}{\partial \theta_n} \right)$$

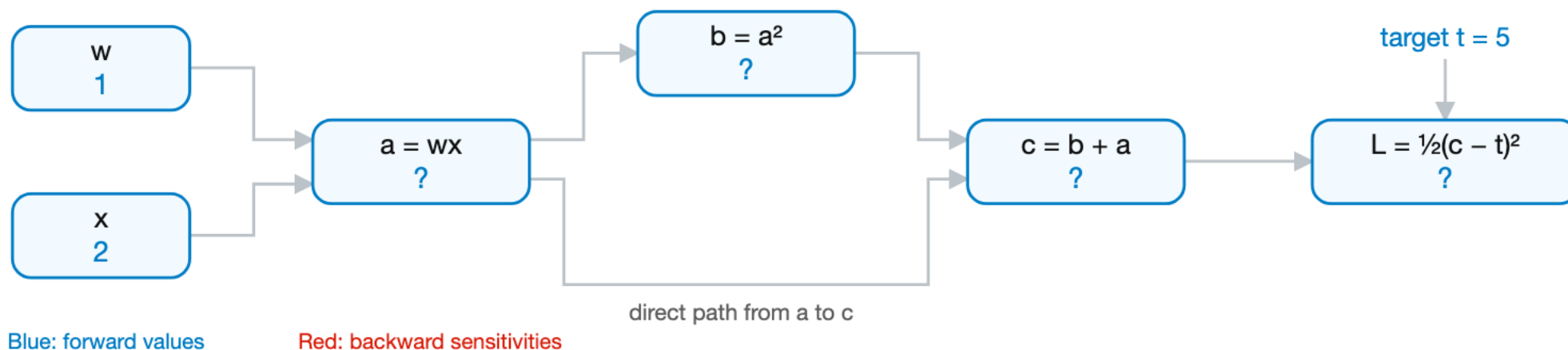
Evaluate the program forwards and retain the values needed for derivatives.

Then propagate sensitivities backwards, using one local derivative at a time.

We will compute every step by hand before running the demonstration.

One graph, two paths

$$a = wx, \quad b = a^2, \quad c = b + a, \quad L = \frac{1}{2}(c - t)^2$$

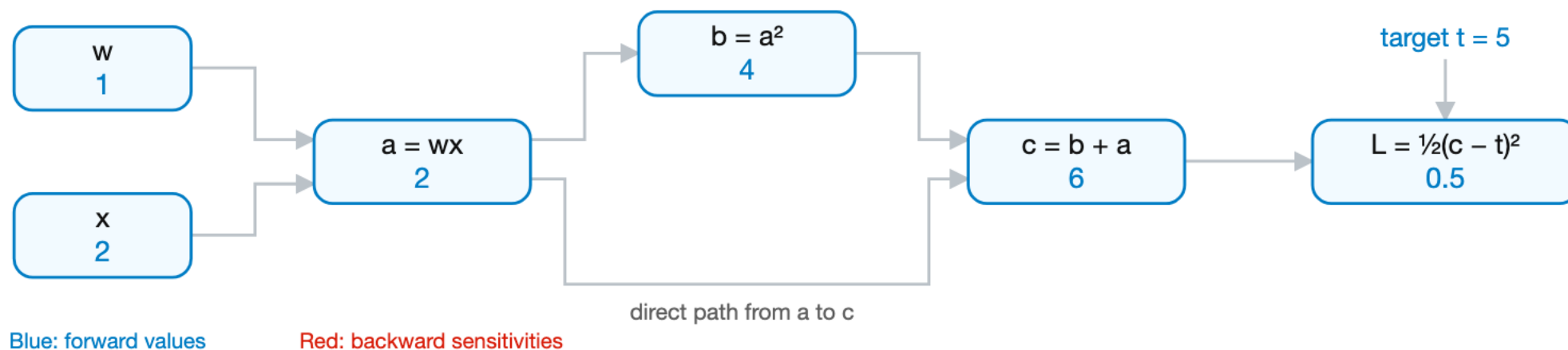


The value a affects the loss through **two paths**.

We train w . The input $x = 2$ and target $t = 5$ stay fixed.

1. Forward pass

Start with $w = 1$, $x = 2$, and $t = 5$.



$$a = 1 \cdot 2 = 2, \quad b = 2^2 = 4, \quad c = 4 + 2 = 6$$

$$L = \frac{1}{2}(6 - 5)^2 = 0.5$$

Keep the intermediate values. The backward pass will use them.

2. What travels backwards?

The **adjoint** of an intermediate v is its sensitivity:

$$\bar{v} \equiv \frac{\partial L}{\partial v}$$

For a small perturbation, $\Delta L \approx \bar{v} \Delta v$.

Forward value

“What number did this operation produce?”

Backward sensitivity

“How does this number affect the loss?”

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

Local backward rules

For an incoming sensitivity $\bar{z}$:

Addition

$$z = u + v$$

$$\bar{u} + = \bar{z}$$

$$\bar{v} + = \bar{z}$$

Send the sensitivity to both inputs.

Multiplication

$$z = uv$$

$$\bar{u} + = v\bar{z}$$

$$\bar{v} + = u\bar{z}$$

Use the other input's forward value.

Squaring

$$z = u^2$$

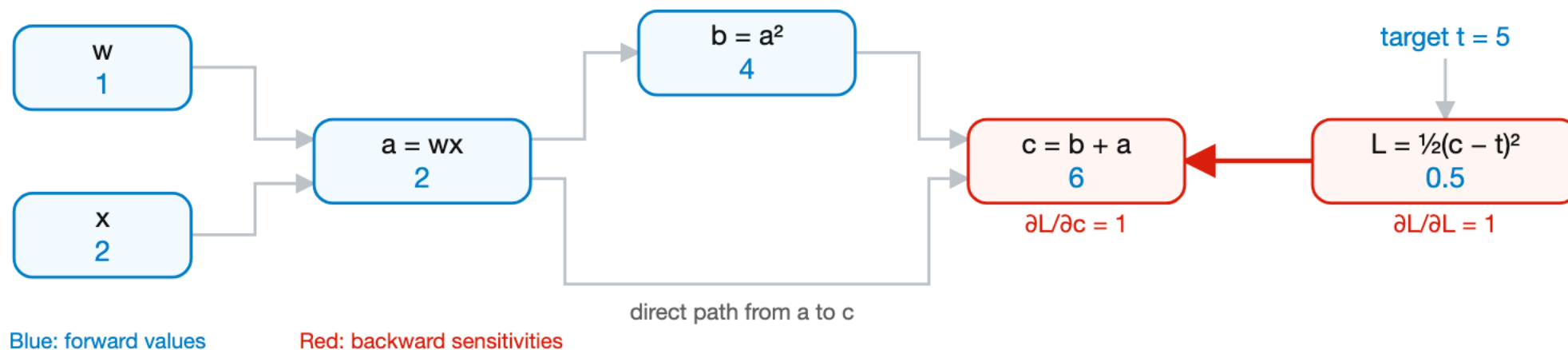
$$\bar{u} + = 2u\bar{z}$$

Use the local slope at the stored value.

The notation $+ =$ means **add this contribution** to the adjoint.

3. Backwards through the loss

$$L = \frac{1}{2}(c - t)^2, \quad c = 6, \quad t = 5$$

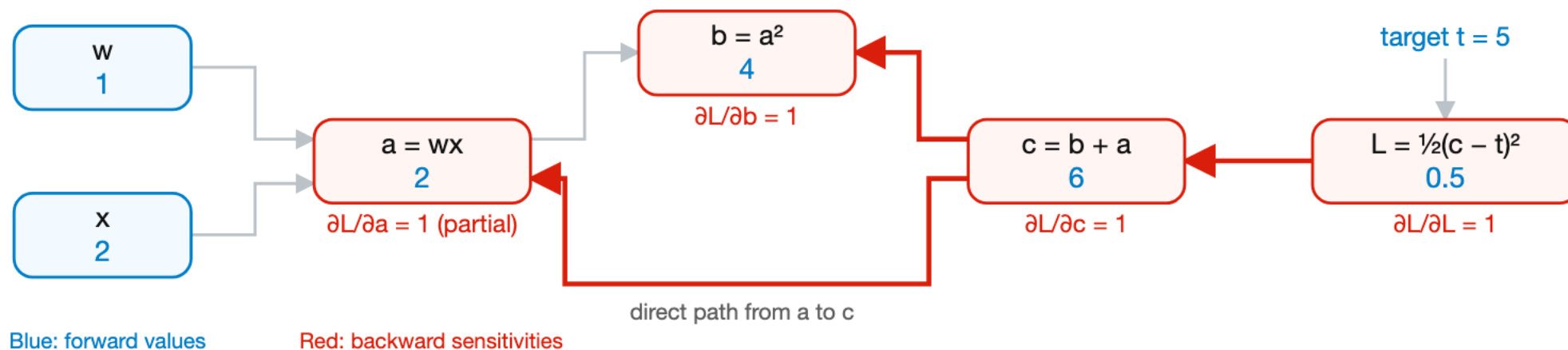


$$\bar{c} = \bar{L} \frac{\partial L}{\partial c} = 1 \cdot (c - t) = 1$$

At this point, increasing c slightly would increase the loss.

4. Backwards through the sum

$$c = b + a, \quad \bar{c} = 1$$

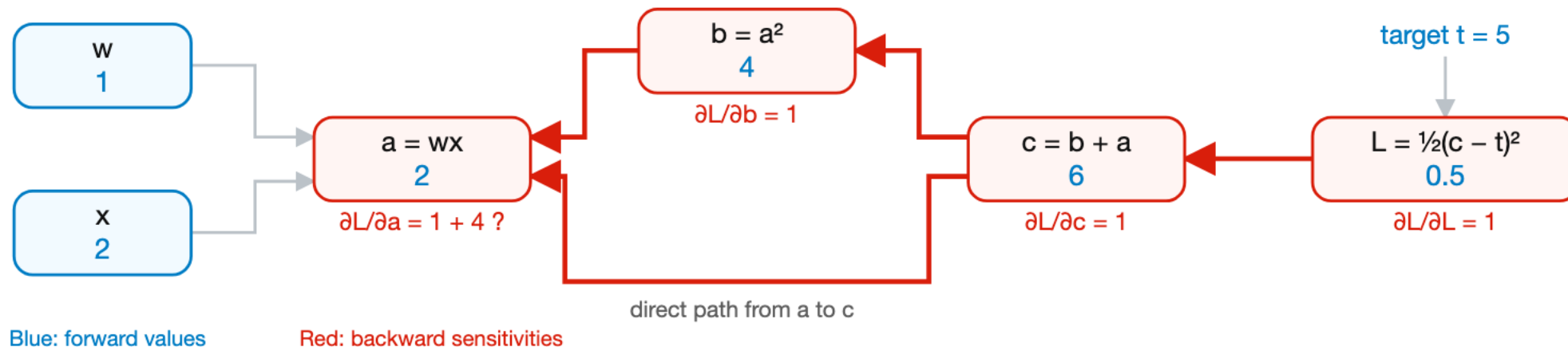


$$\bar{b} + = \bar{c} \cdot 1 = 1, \quad \bar{a} + = \bar{c} \cdot 1 = 1$$

This is only the **direct-path contribution** to $\bar{a}$.

5. Backwards through the square

$$b = a^2, \quad a = 2, \quad \bar{b} = 1$$

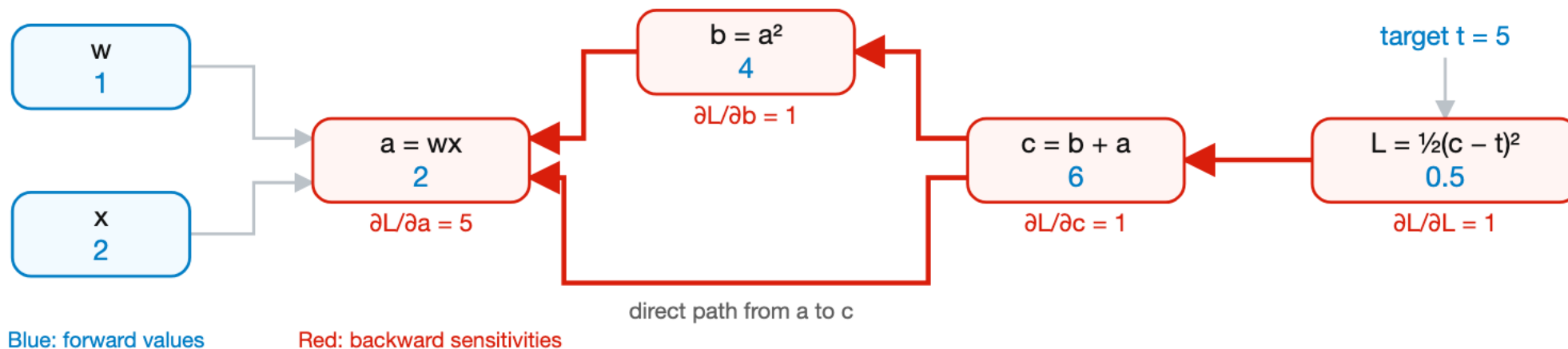


$$\bar{a} + = \bar{b} \frac{\partial b}{\partial a} = 1 \cdot 2a = 4$$

Is $\bar{a}$ now 1, 4, or 5?

6. Accumulation at a shared node

$$\bar{a} = \underbrace{\bar{c} \frac{\partial c}{\partial a}}_{\text{direct path}} + \underbrace{\bar{c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a}}_{\text{path through } b} = 1 + 4 = 5$$

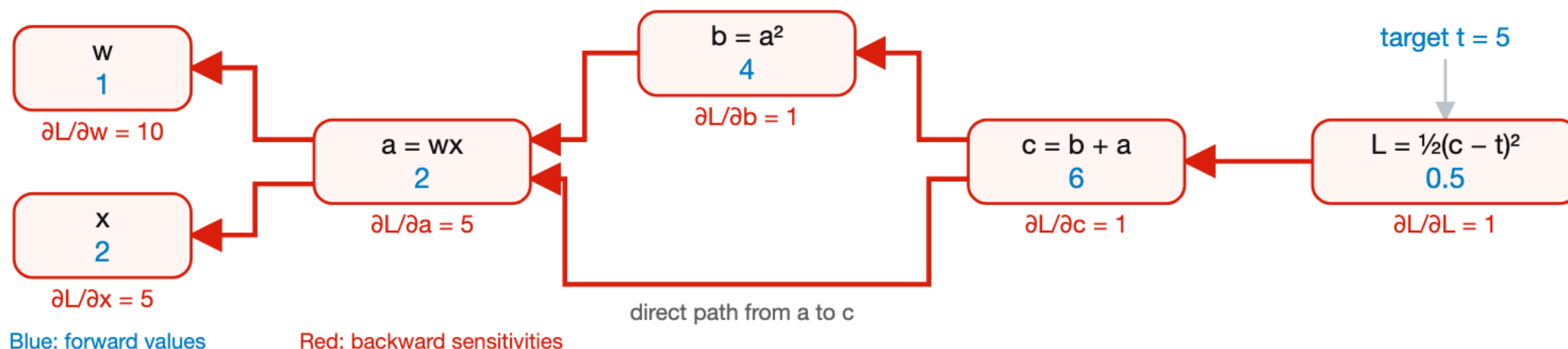


Multiply derivatives along each path. **Add** contributions from different paths.

Only propagate further once all contributions to the node have arrived.

7. Backwards to the parameter

$$a = wx, \quad w = 1, \quad x = 2, \quad \bar{a} = 5$$



$$\bar{w} = \bar{a} x = 5 \cdot 2 = 10, \quad \bar{x} = \bar{a} w = 5 \cdot 1 = 5$$

The derivative with respect to the trainable parameter is $\partial L / \partial w = 10$. We can also compute $\bar{x}$, even though x is fixed during this update.

8. A parameter update

Backpropagation has finished. Now apply gradient descent with $\eta = 0.01$.

$$w_{\text{new}} = w - \eta \bar{w} = 1 - 0.01 \cdot 10 = 0.9$$

	Before	After
Parameter w	1	0.9
Prediction c	6	5.04
Loss L	0.5	0.0008

The new loss requires a **new forward pass**. A larger step may increase it.

A neuron uses the same rules

$$z = w_1 x_1 + w_2 x_2 + b, \quad h = \sigma(z), \quad L = \frac{1}{2}(h - t)^2$$

Work backwards, starting at the scalar loss:

$$\begin{aligned} \bar{h} &= h - t, & \bar{z} &= \bar{h} \sigma'(z) \\ \bar{w}_1 &= \bar{z} x_1, & \bar{w}_2 &= \bar{z} x_2, & \bar{b} &= \bar{z} \end{aligned}$$

For a sigmoid, $\sigma'(z) = h(1 - h)$: reuse the activation from the forward pass.

Backpropagation through layers

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \quad \mathbf{h}^{(\ell)} = \sigma(\mathbf{z}^{(\ell)})$$

Given the incoming sensitivity $\bar{\mathbf{h}}^{(\ell)}$:

$$\begin{aligned} \bar{\mathbf{z}}^{(\ell)} &= \bar{\mathbf{h}}^{(\ell)} \odot \sigma'(\mathbf{z}^{(\ell)}) \\ \bar{\mathbf{W}}^{(\ell)} &= \bar{\mathbf{z}}^{(\ell)} (\mathbf{h}^{(\ell-1)})^T, \quad \bar{\mathbf{b}}^{(\ell)} = \bar{\mathbf{z}}^{(\ell)} \\ \bar{\mathbf{h}}^{(\ell-1)} &= (\mathbf{W}^{(\ell)})^T \bar{\mathbf{z}}^{(\ell)} \end{aligned}$$

Repeat for the previous layer. Here $\odot$ denotes elementwise multiplication.

The reverse sweep in code

```
1 # Forward: record operations and their needed input values.
2 value = evaluate_in_dependency_order(graph)
3
4 # Backward: start with zero contributions everywhere.
5 adjoint = {node: 0.0 for node in graph}
6 adjoint[loss] = 1.0
7
8 for node in reversed(dependency_order):
9     for input_edge in node.input_edges:
10         parent = input_edge.source
11         adjoint[parent] += (
12             adjoint[node] * local_derivative(input_edge, value)
```

Process **edges**, including repeated inputs. For $z = u \cdot u$, both edges contribute.

Checking a gradient

For our example, compute an independent central finite difference:

$$g_{\text{FD}} = \frac{L(w + \varepsilon) - L(w - \varepsilon)}{2\varepsilon}$$

At $w = 1$, $x = 2$, $t = 5$, and $\varepsilon = 10^{-5}$:

$$g_{\text{FD}} \approx 10.0000000004, \quad g_{\text{backprop}} = 10$$

Use this as a diagnostic on a small, smooth example.

Very small finite-difference steps amplify roundoff. At a nondifferentiable point, the comparison needs special care.

One reverse pass, many parameters

For one scalar loss, the reverse sweep computes all parameter derivatives.

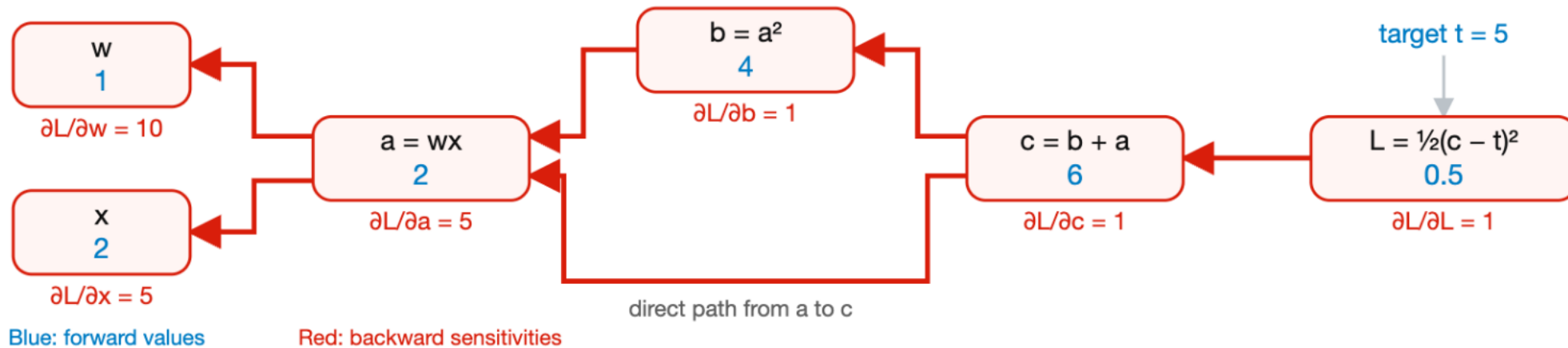
Each operation sends sensitivities to its inputs. The work scales with the computation graph, with a constant-factor overhead for standard operations.

The tradeoff is **memory**: backward rules need values from the forward pass.

Checkpointing saves selected values and recomputes others.

Backpropagation demonstration

$$a = wx \quad b = a^2 \quad c = b + a \quad L = \frac{1}{2}(c - t)^2$$

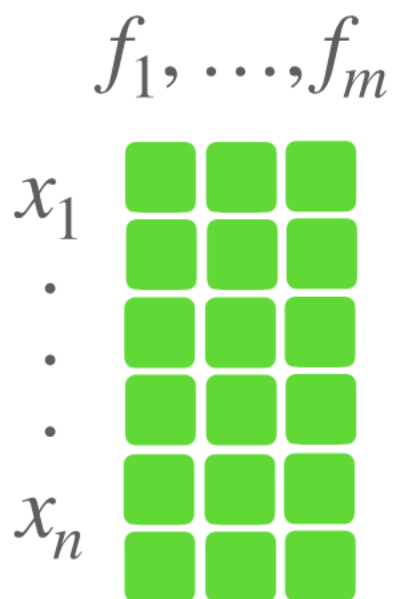


6. Parameter gradient

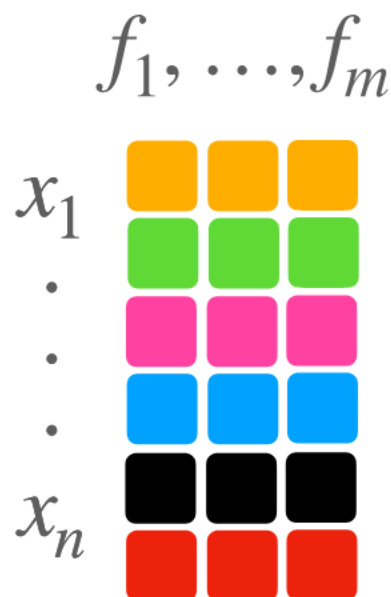
$\frac{\partial L}{\partial w} = x \times \frac{\partial L}{\partial a} = 10$. Also $\frac{\partial L}{\partial x} = 5$.

Computation cost of Jacobian matrix

$$R^n \xrightarrow{f} R^m \quad f_1(x), \dots, f_m(x) \quad (\text{multiple})$$

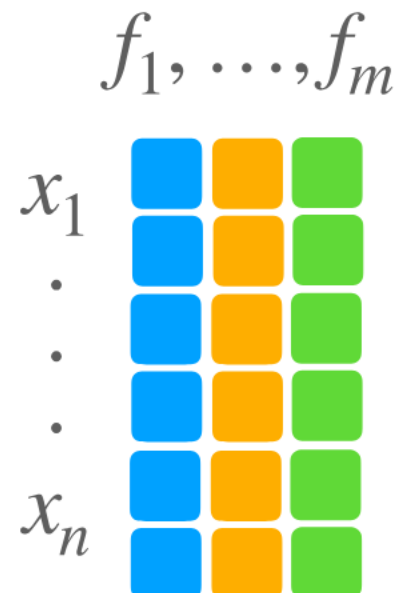


$$\frac{\partial(f_1, \dots, f_m)}{\partial(x_1, \dots, x_n)}$$



Forward AD

1. x_1 dual
2. x_2 dual
3. ...



Reverse AD

1. effect on f_1
2. effect on f_2
3. ...

Higher-Order Derivatives

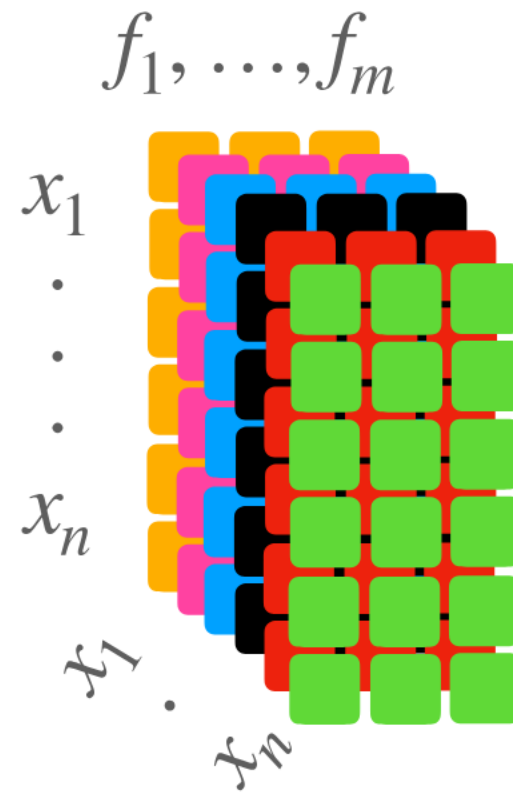
Hessian

$$R^n \xrightarrow{f} R^m \quad f_1(x), \dots, f_m(x) \quad (\text{multiple})$$

- **Forward mode** can be applied twice, or use Hyper-dual numbers
- **Reverse mode** can be applied to each gradient component to get row-wise Hessians

$$f_1(x) : \quad \vec{x} \in R^n \xrightarrow{f_1} R \quad (\text{scalar})$$

$$\frac{\partial^2 f_1}{\partial x_i \partial x_j}$$

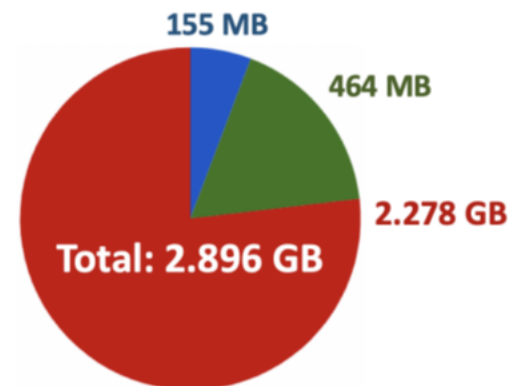


Practical Considerations

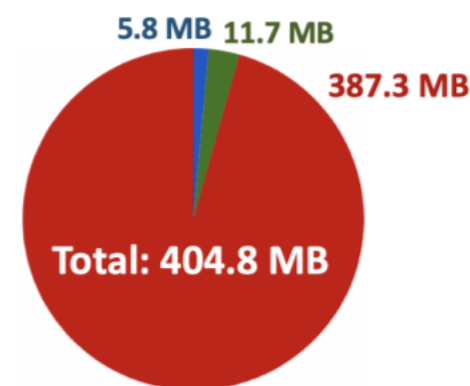
Memory & Complexity

- fAD has a small memory footprint — carry derivative alongside the computation
- rAD must remember all intermediate results (GPT-3: 175 billion parameters Float16 ~350 GB), some mitigated with re-computation (**Gradient-checkpointing**) <https://en.wikipedia.org/wiki/GPT-3>

Transformer (IWSLT'14)



ResNet (CIFAR-10)



Automatic differentiation in practice

Frameworks, transformations, compilers, and
implementation trade-offs

Existing AD techniques

- **Differentiable DSLs** (TensorFlow, PyTorch) provide a new language where everything is differentiable. Must rewrite code in DSL.
- **Operator Overloading** (Adept, JAX) tools provide differentiable versions of language constructs (e.g. `np.sum` => `jax.sum`). Require rewriting non-standard utilities for support.
- **Source Rewriting** tools statically analyze code to produce a new gradient function in the source language. Requires all code available ahead of time, difficult with external libraries.



AD in Major ML Frameworks

		Default graph	AD style
PyTorch		Dynamic	Tape reverse
JAX		Trace to HLO	Transform reverse
TF 2		Eager → Opt graph	Tape or grad ops

Python ecosystem

[JAX](#) · [PyTorch](#) · [TensorFlow](#)

Tool	Role	Differentiation interface
NumPy / SciPy	Numerical arrays and scientific algorithms	Derivatives supplied separately or approximated
SymPy	Symbolic expressions	<code>diff(expression, variable)</code>
JAX	Array programs and function transformations	<code>jvp</code> , <code>vjp</code> , <code>grad</code>
PyTorch	Tensors, neural networks, training	<code>torch.func</code> , <code>.backward()</code>
TensorFlow	Tensors, models, training	<code>ForwardAccumulator</code> , <code>GradientTape</code>
marimo	Reactive Python notebooks	Runs the chosen library's code

Forward evaluation and forward-mode AD

For $\mathbf{y} = F(\mathbf{x})$, ordinary evaluation computes $\mathbf{y}$.

Forward-mode AD also propagates an **input direction** $\mathbf{v}$:

$$\dot{\mathbf{y}} = J_F(\mathbf{x}) \mathbf{v}$$

Value

$$z = uv$$

Tangent

$$\dot{z} = \dot{u}v + u\dot{v}$$

A Jacobian-vector product (**JVP**) answers: how does the output change along this input direction?

Reverse mode: an output sensitivity

Evaluate $\mathbf{y} = F(\mathbf{x})$, then choose an output sensitivity $\mathbf{u}$.

$$\bar{\mathbf{x}} = J_F(\mathbf{x})^T \mathbf{u}$$

```
1 y, pullback = jax.vjp(F, x)
2 (sensitivity_x,) = pullback(u)
```

The **pullback** maps an output sensitivity back to the inputs.

For a scalar loss, seed the output with 1 to obtain its gradient.

JVP: a direction through the inputs. VJP: a weighted combination of output sensitivities.

JAX: the lecture's scalar loss

```
1 import jax
2 jax.config.update("jax_enable_x64", True)
3
4 def loss(w):
5     a = 2.0 * w
6     return 0.5 * (a**2 + a - 5.0)**2
7
8 value, tangent = jax.jvp(loss, (1.0,), (1.0,))
9 value, pullback = jax.vjp(loss, 1.0)
10 (gradient,) = pullback(1.0)
11
12 print(value, tangent, gradient)    # 0.5, 10.0, 10.0
13 print(jax.grad(loss)(1.0))        # 10.0
```

The input direction is 1. The scalar-loss output seed is also 1.

marimo: compare the two modes

[Live notebook](#) · [Snapshot](#) · [Python source](#)

Weight w  1

Computation	Result
Loss	0.5
Forward mode: <code>jax.jvp</code> , input tangent 1	10
Reverse mode: <code>jax.vjp</code> , output seed 1	10
Scalar gradient: <code>jax.grad</code>	10

Change w from 1 to 0.9: predict the loss and gradient before rerunning.

A vector function

$$F(x_1, x_2) = \begin{pmatrix} x_1 x_2 \\ x_1^2 + x_2 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \quad J = \begin{pmatrix} 3 & 2 \\ 4 & 1 \end{pmatrix}$$

Forward: select a column

$$\mathbf{v} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad J\mathbf{v} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

Reverse: select a row

$$\mathbf{u} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad J^T \mathbf{u} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

For general seeds, compute linear combinations of columns or rows.

JAX: products and full Jacobians

```
1 import jax.numpy as jnp
2
3 def F(x):
4     return jnp.array([x[0] * x[1], x[0]**2 + x[1]])
5
6 x = jnp.array([2.0, 3.0])
7 seed = jnp.array([1.0, 0.0])
8 y, Jv = jax.jvp(F, (x,), (seed,)) # [3., 4.]
9 y, pullback = jax.vjp(F, x)
10 (JT_u,) = pullback(seed)          # [3., 2.]
11
12 J_forward = jax.jacfwd(F)(x)      # [[3., 2.],
13 J_reverse = jax.jacrev(F)(x)      # [4., 1.]
```

Both Jacobian functions return the same matrix. JVP and VJP need not build it.

marimo: input direction and output seed

[Live notebook](#) · [Snapshot](#) · [Python source](#)

Input direction v $e_1 = (1, 0)$ ▼

Output seed u First output: $u = (1, 0)$ ▼

$$J = \begin{pmatrix} 3 & 2 \\ 4 & 1 \end{pmatrix}$$

Quantity	Result
$F(x)$	[6.0, 7.0]
Input tangent v	[1.0, 0.0]
Jv	[3.0, 4.0]
Output sensitivity u	[1.0, 0.0]
$J^T u$	[3.0, 2.0]

Switch each seed to $(0, 1)$. Which column or row should appear?

PyTorch: forward and reverse transforms

```
1 import torch
2 from torch.func import jvp, vjp
3
4 w = torch.tensor(1.0, dtype=torch.float64)
5 one = torch.ones_like(w)
6
7 value, tangent = jvp(loss, (w,), (one,))
8 value, pullback = vjp(loss, w)
9 (gradient,) = pullback(one)
10
11 print(value.item())      # 0.5
12 print(tangent.item())    # 10.0
13 print(gradient.item())   # 10.0
```

Use the same scalar `loss` function. These transforms return derivatives directly.

PyTorch: a training step

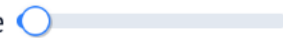
```
1 w = torch.tensor(1.0, dtype=torch.float64, requires_grad=True)
2 optimizer = torch.optim.SGD([w], lr=0.01)
3
4 optimizer.zero_grad(set_to_none=True)
5 value = loss(w)           # forward evaluation: 0.5
6 value.backward()          # reverse mode
7 print(w.grad.item())      # 10.0
8
9 optimizer.step()           # optimizer changes w
10 print(w.item())           # 0.9
11 print(loss(w).item())     # approximately 0.0008
```

`.backward()` **accumulates** into `.grad`. Clear gradients before the next step.

marimo: one optimizer step

[Live notebook](#) · [Snapshot](#) · [Python source](#)

Initial w  1

Learning rate  0.01

Quantity	Result
Loss before update	0.5
Gradient from <code>.backward()</code>	10
Forward mode: <code>torch.func.jvp</code>	10
Reverse mode: <code>torch.func.vjp</code>	10
Updated weight	0.9
Loss after update	0.0008

Increase the learning rate to 0.5. The gradient is still correct. What happens to the loss?

Choosing and checking an AD mode

For $F: \mathbb{R}^n \rightarrow \mathbb{R}^m$, think about what derivative you need.

Task	Usual starting point
One input direction	JVP / forward mode
Scalar loss, many parameters	Gradient / reverse mode
Full Jacobian, few inputs	<code>jacfwd</code>
Full Jacobian, few outputs	<code>jacrev</code>

Keep operations inside the library's differentiable computation. Convert to NumPy or Python numbers **after** computing the derivative.

Check small examples analytically or with finite differences, and benchmark the real workload.

Example: mutations and branchings



f (generic function with 1 method)

```
• function f(x)
•   y = 2x
•   y[3] = y[2]
•   y .^ 2
•   return y[5]>0 ? y[1] : sum(y)
• end
```

```
CodeInfo(
1 - %1 = Main.var"workspace#11"::~*
  y = (%1)(2, x)
  %3 = y
  %4 = Base.getindex(%3, 2)
  %5 = y
  Base.setindex!(%5, %4, 3)
  %7 = Main.var"workspace#11"::^
  %8 = y
  %9 = Core.apply_type(Base.Val, 2)
  %10 = (%9)()
  %11 = Base.broadcasted(Base.literal_pow, %7, %8, %10)
  Base.materialize(%11)
  %13 = Main.var"workspace#11"::>
  %14 = y
  %15 = Base.getindex(%14, 5)
  %16 = (%13)(%15, 0)
  goto #3 if not %16
2 - %18 = y
  %19 = Base.getindex(%18, 1)
  return %19
3 - %21 = Main.var"workspace#11".sum
  %22 = y
  %23 = (%21)(%22)
  return %23
)
```

• @code_lowered f(x0)

Error message

Mutating arrays is not supported -- called setindex!(Vector{Float64}, ...)
This error occurs when you ask Zygote to differentiate operations that change the elements of arrays in place (e.g. setting values with `x .= ...`)

Possible fixes:

- avoid mutating operations (preferred)
- or read the documentation and solutions for this error
<https://fluxml.ai/Zygote.jl/latest/limitations>

```
5. f
from [Other cell: line 3]
1 function f(x)
2   y = 2x
3   y[3] = y[2]
4   y .^ 2
5   return y[5]>0 ? y[1] : sum(y)
```

cell preview

6. [Show more...](#)

```
1 Zygote.gradient(f, x0)
```

Example: differentiating LLVM

```
+  
⦿ f (generic function with 1 method)  
• function f(x)  
•     y = 2x  
•     y[3] = y[2]  
•     y .^ 2  
•     return y[5]>0 ? y[1] : sum(y)  
• end
```

Enzyme

```
• using Enzyme ✓
```

```
x0 = ▶ [-1.49182, 0.413275, -0.5286, -0.136647, -0.433667]
```

```
• x0 = randn(5)
```

```
▶ [2.0, 4.0, 0.0, 2.0, 2.0]
```

```
• let  
•     dx = zeros(5)  
•     autodiff(Reverse, f, Active, Duplicated(x0, dx))  
•     dx  
• end
```

Enzyme

```
• using Enzyme ✓
```

```
+  
> x0 = ▶ [0.942497, 0.261366, -0.363999, 0.360958, 0.166457]
```

```
• x0 = randn(5)
```

▶ 1.0 μs

```
+  
▶ [2.0, 0.0, 0.0, 0.0, 0.0]
```

```
• let  
•     dx = zeros(5)  
•     autodiff(Reverse, f, Active, Duplicated(x0, dx))  
•     dx  
• end
```

From Source to Machine Code

XLA & LLVM

- Pipeline
Source (Julia/Python) → Front-end IR → Optimizer → Machine Code
- **XLA**: linear-algebra-centric compiler used by JAX / TF; IR = HLO
- **LLVM**: language-agnostic compiler back-end (Julia, C/C++, Rust)

Historical and conceptual epilogue

An old algorithm, modern scale, and the cheap-
gradient principle

Backpropagation is older than deep learning

1970 — Seppo Linnainmaa

Systematic reverse accumulation for differentiating a program.

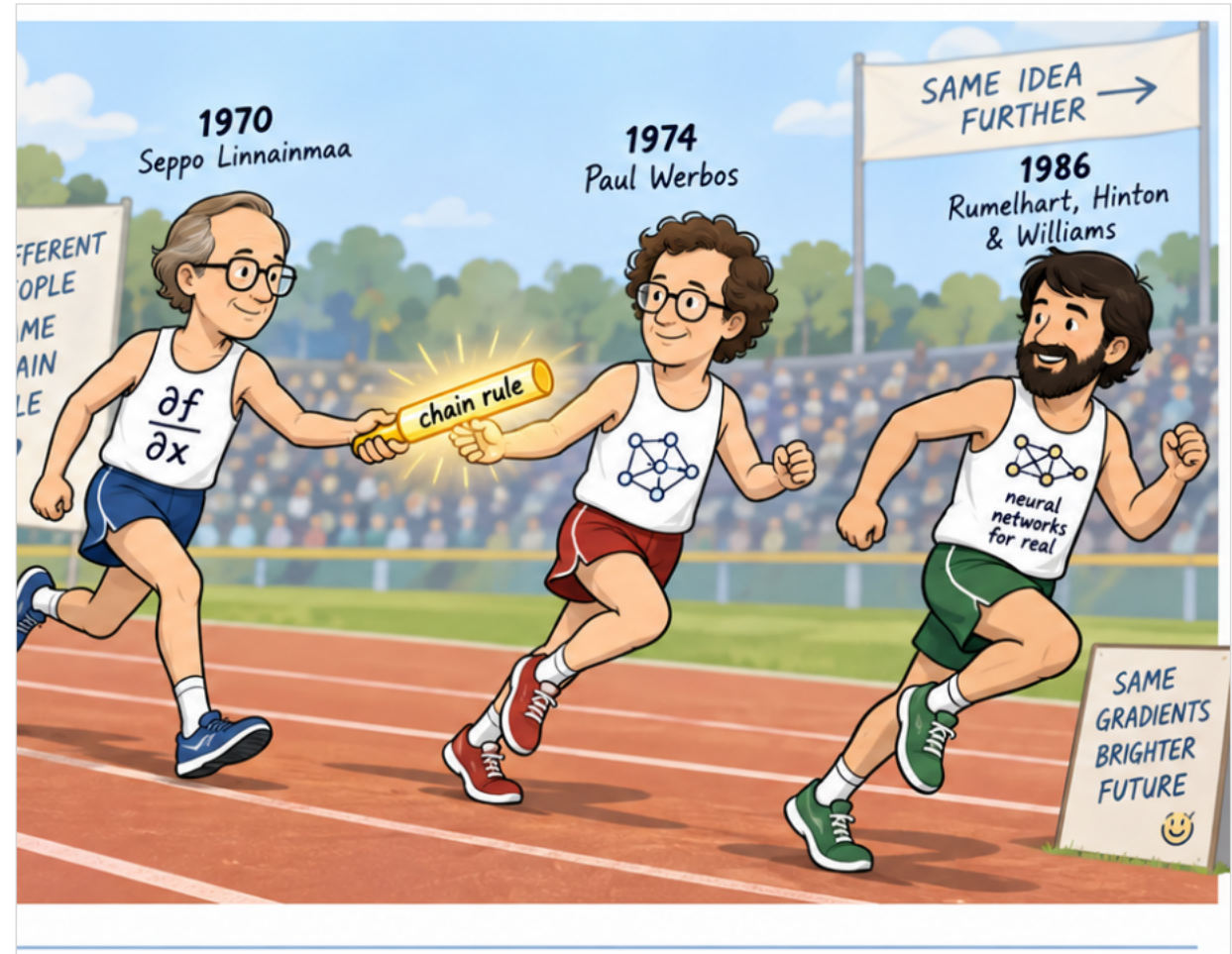
1974 — Paul Werbos

Connected backward differentiation with learning systems.

1986 — Rumelhart, Hinton & Williams

Demonstrated and popularized backpropagation for multilayer neural networks.

Backpropagation is reverse-mode automatic differentiation applied to a learning problem.



The same algorithm at larger scale

1989 — handwritten ZIP codes

LeCun and colleagues trained a convolutional network with backpropagation.

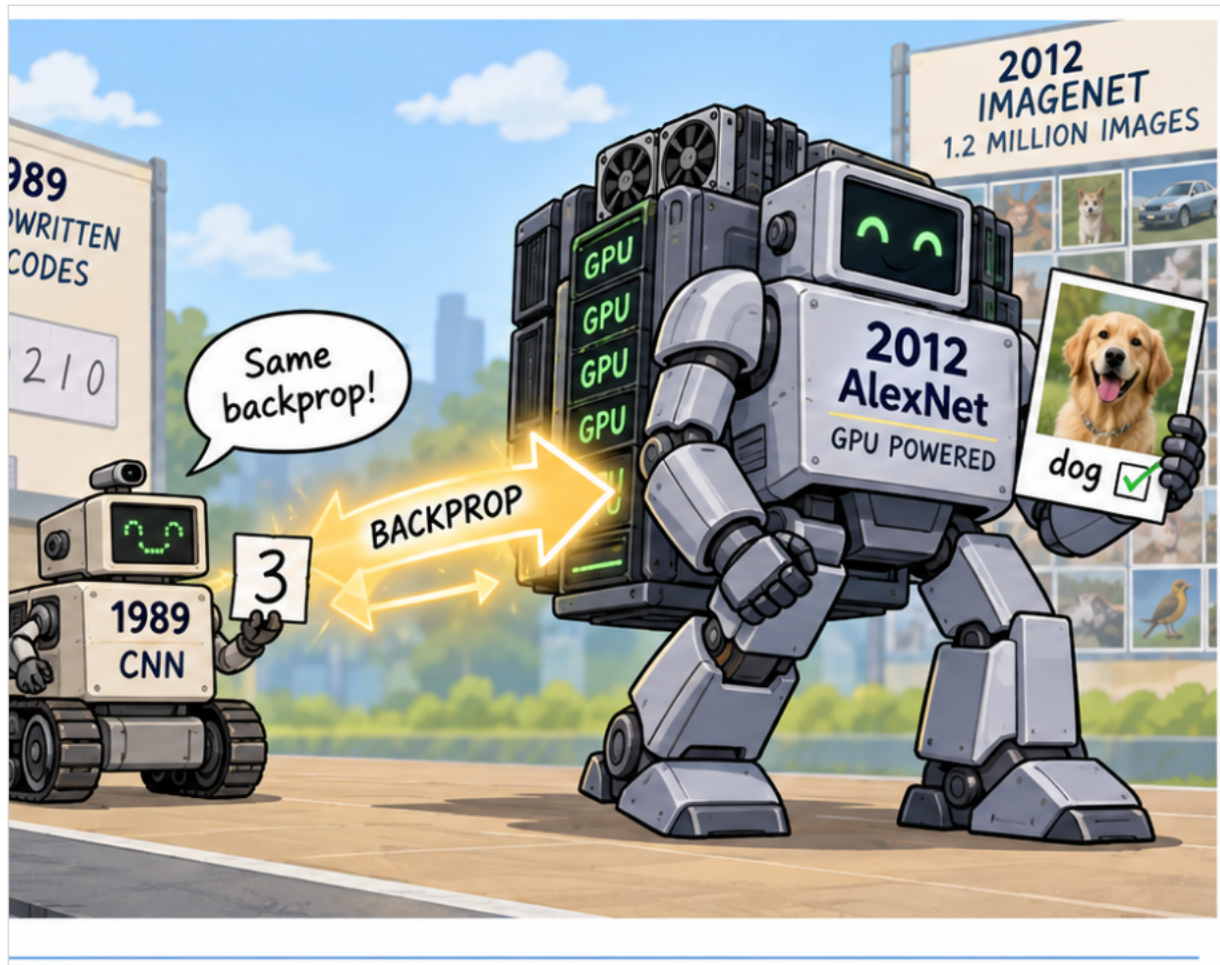
1990s — document recognition

Neural systems moved into deployed cheque and document processing.

2012 — AlexNet

GPU training, about 60 million parameters, and 1.2 million ImageNet training images.

Data, compute, architecture and scale changed dramatically. The chain rule did not.



The surprising fact: gradients are cheap

For a scalar objective

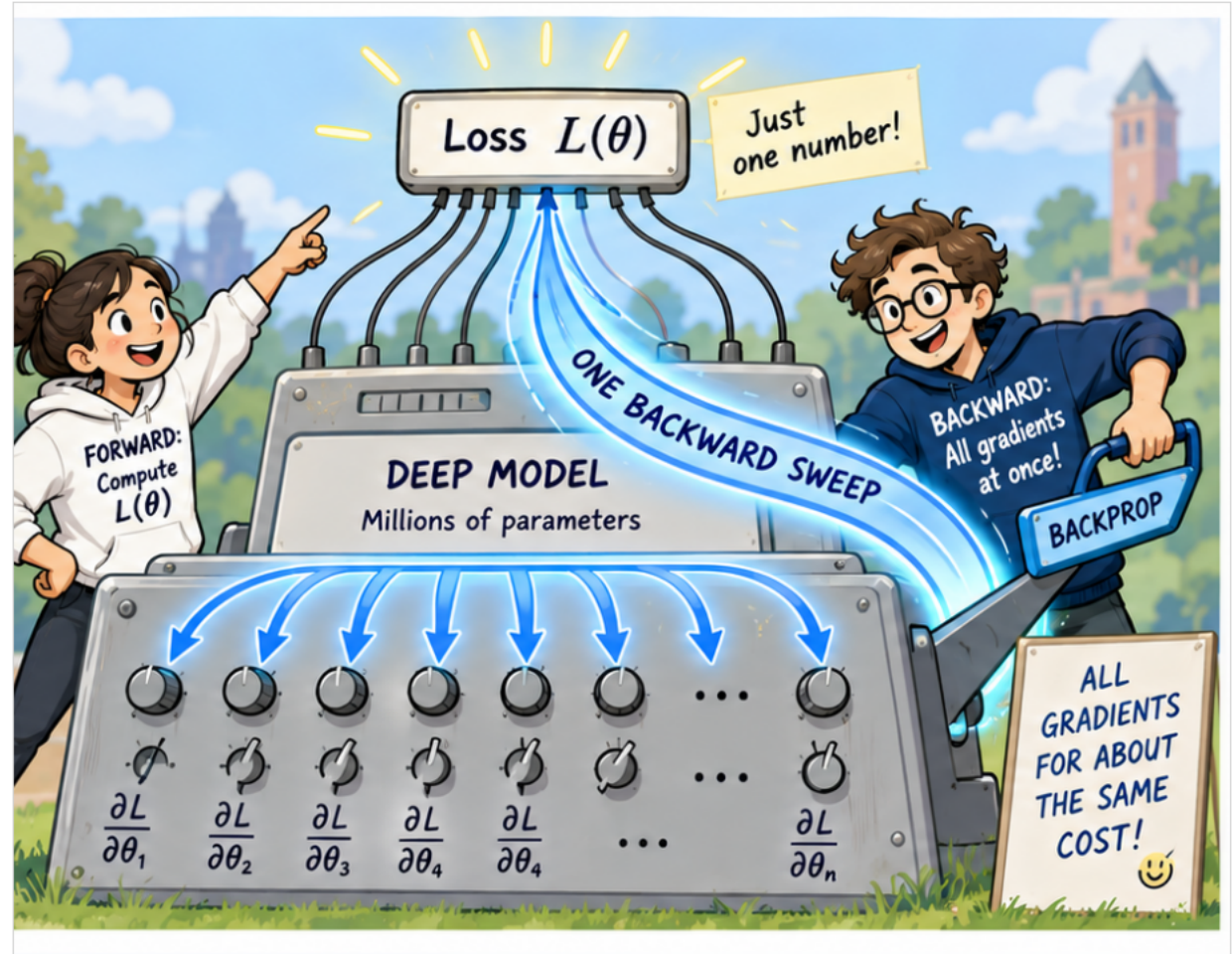
$$f: \mathbb{R}^n \rightarrow \mathbb{R},$$

one reverse sweep produces the complete gradient:

$$\text{cost}(\nabla f) = O(\text{cost}(f)).$$

In practice, the backward pass costs a small constant multiple of the forward computation. It also needs stored or recomputed intermediate values.

10^6 or 10^9 parameters do not require 10^6 or 10^9 separate derivative calculations.



End

Backup follows

Julia AD: Enzyme

- LLVM plug-in; differentiates after compiler optimizations
- Handles mutation & C/Fortran kernels; cross-language
- Benchmarks: gradient $\approx 4.5\times$ faster vs unoptimized source-level AD

Julia AD: Zygote

- Source-to-source transformation → gradient code compiled by Julia
- No runtime tape; functional style, uses ChainRules definitions
- Call: `grads = Zygote.gradient(f, params)`