Automatic Differentiation: The Engine of Machine Learning

Krishna Kumar

University of Texas at Austin

krishnak@utexas.edu

Overview

Why Do We Need Derivatives?

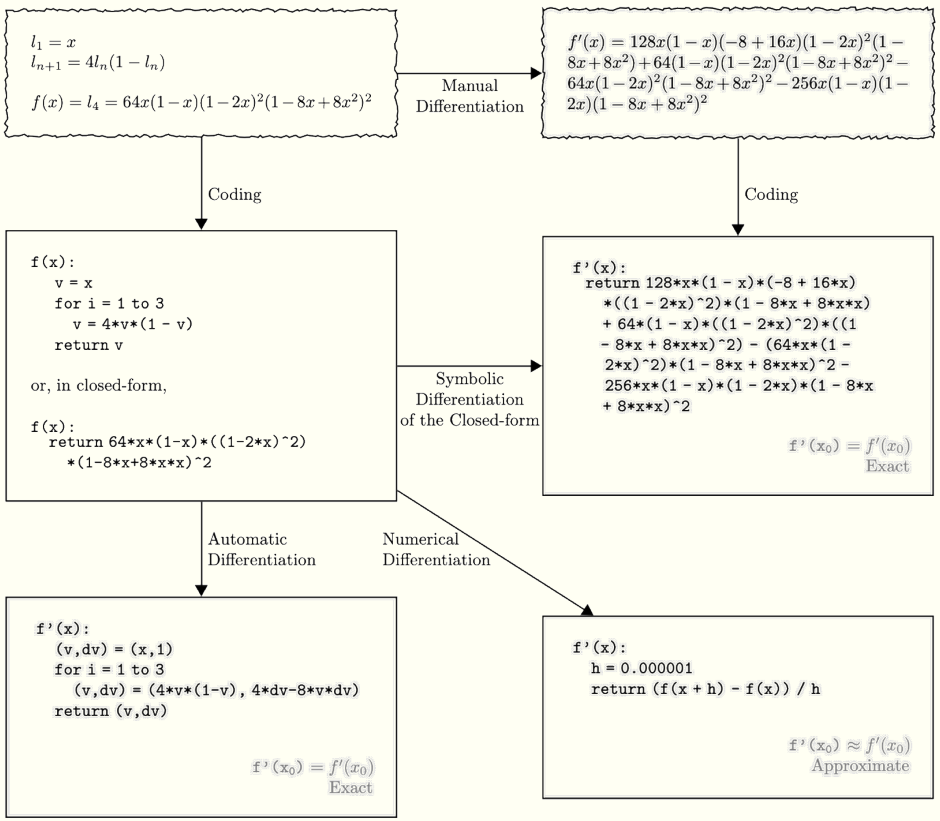
Derivatives are the mathematical engine driving modern scientific computing.

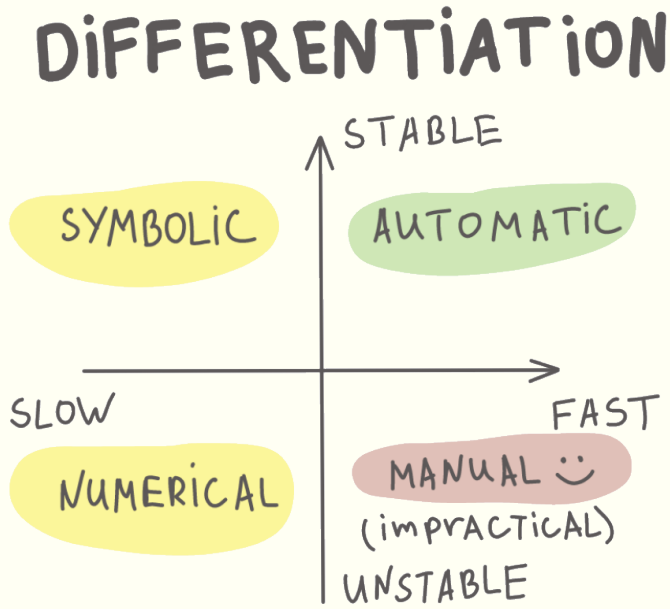
- **Optimization:** Finding the minimum of a loss function.
- **Machine Learning:** Training neural networks via gradient descent.
- **Inverse Problems:** Estimating parameters from data (e.g., geophysics).
- **Scientific Machine Learning:** Differentiable simulators, PINNs, Neural ODEs.

Challenge: How do we compute derivatives of complex, computer-implemented functions efficiently and accurately?

The Four Modes of Differentiation

Traditionally, we have had three approaches, each with fundamental flaws.

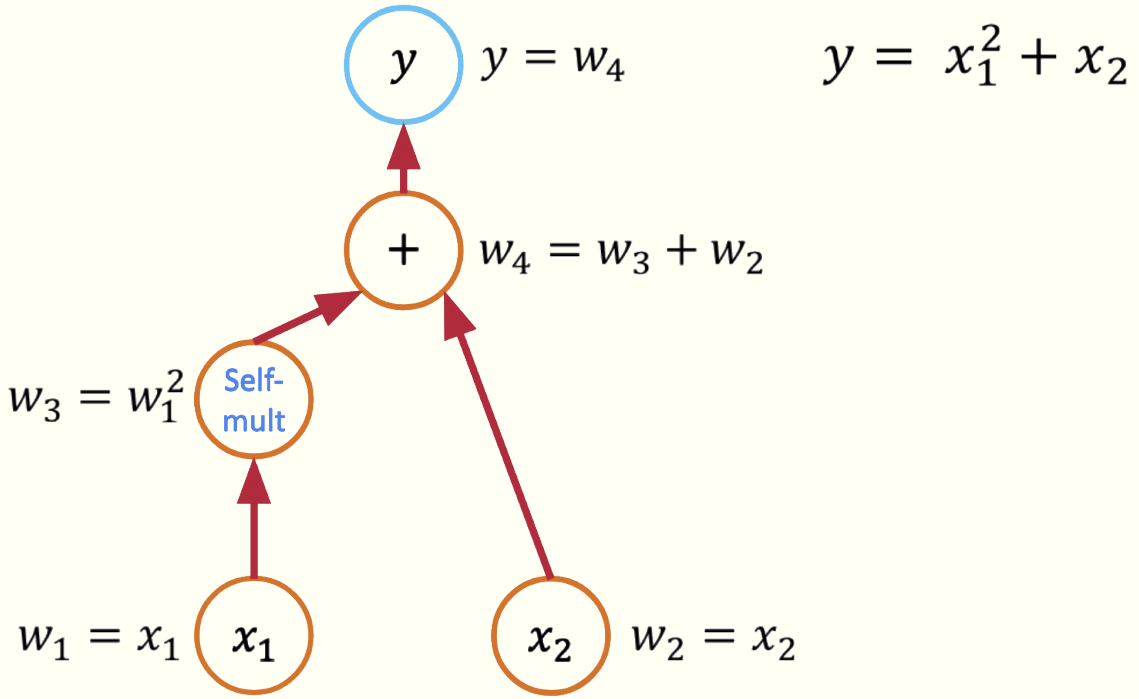




- **Manual:** Exact, but slow, error-prone, and not scalable.
- **Symbolic:** Exact, but suffers from "expression swell" and is computationally expensive.
- **Numerical (Finite Diff.):** Simple, but suffers from accuracy issues.

Functions are Computational Graphs

Every function implemented as a computer program can be decomposed into a sequence of elementary operations.
Consider the function: $y = x_1^2 + x_2$
This can be viewed as a computational graph where each operation is a node.



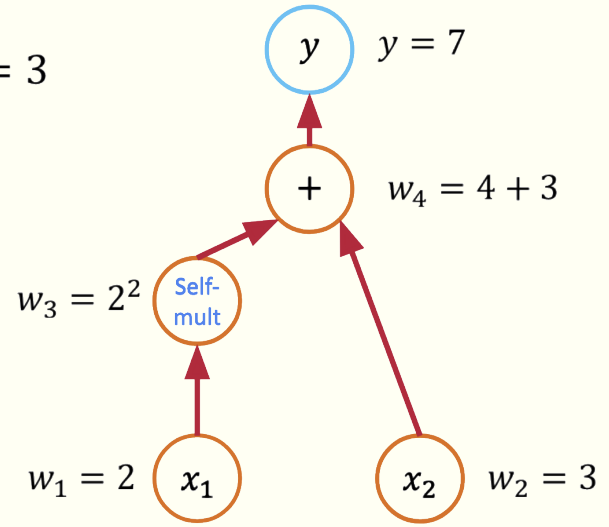
Decomposition into Elementary Steps

We can break down the graph and assign intermediate variables.

Evaluation Trace:

1. $v_1 = x_1^2$
2. $y = v_1 + x_2$

$x_1 = 2, x_2 = 3$



$y = x_1^2 + x_2$

$y = 2^2 + 3$

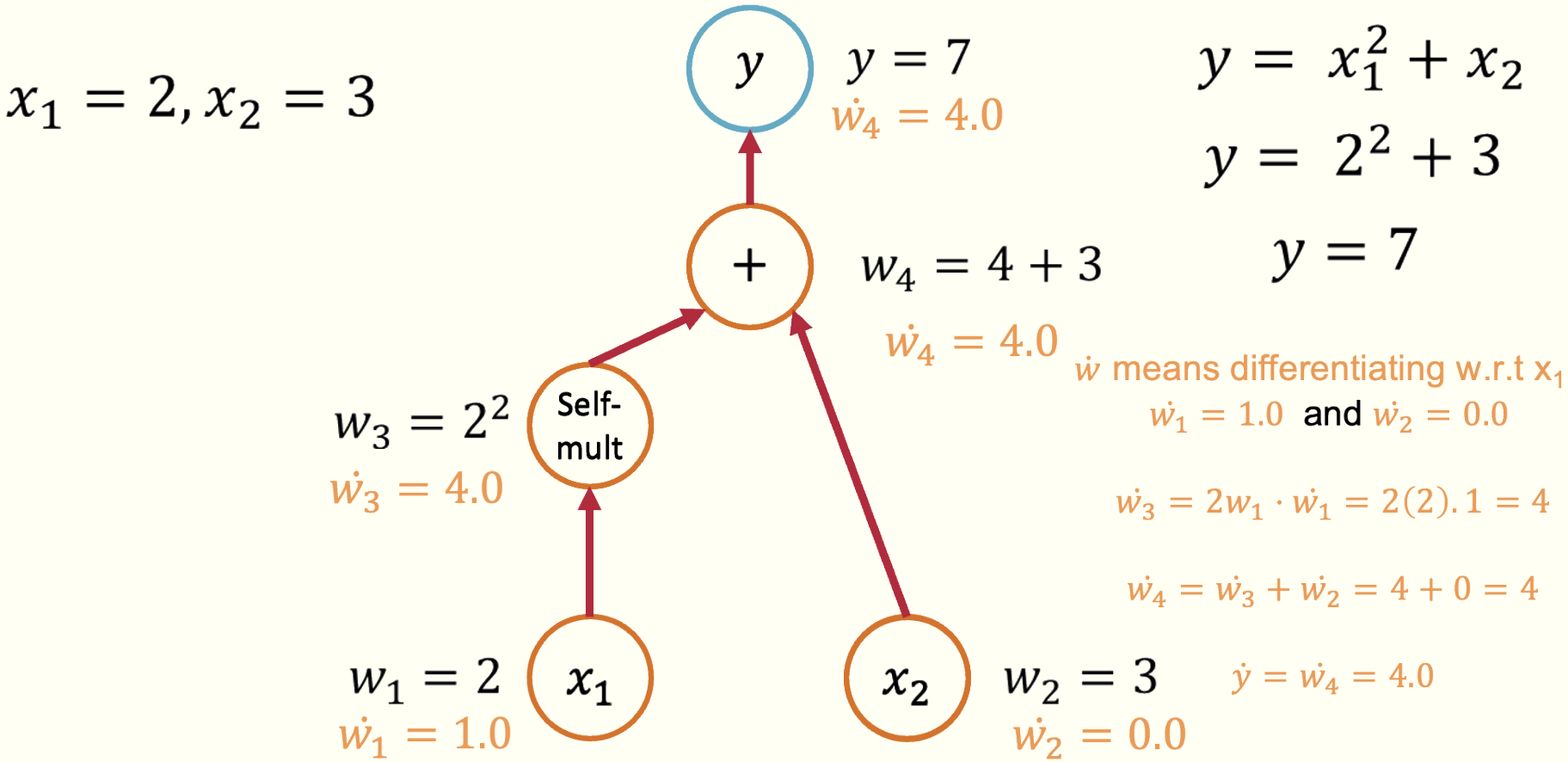
$y = 7$

This decomposition is the key that makes AD possible. By applying the chain rule to each elementary step, we can compute exact derivatives.

Outline

Forward Mode Automatic Differentiation

Forward mode AD propagates derivative information **forward** through the graph, alongside the function evaluation.



Forward Mode: Step-by-Step Example

Let's compute $\frac{\partial y}{\partial x_1}$ for $y = x_1^2 + x_2$.

1. Seed the input w.r.t. x_1

Set $\dot{x}_1 = 1$ and $\dot{x}_2 = 0$.

2. Forward Propagation

- $v_1 = x_1^2 \implies \dot{v}_1 = 2x_1 \cdot \dot{x}_1 = 2x_1 \cdot 1 = 2x_1$
- $y = v_1 + x_2 \implies \dot{y} = \dot{v}_1 + \dot{x}_2 = 2x_1 + 0 = 2x_1$

Result

The final propagated tangent $\dot{y}$ is the derivative: $\frac{\partial y}{\partial x_1} = 2x_1$.

Key Idea: To get the derivative w.r.t. x_2, we would need a *new pass* with seeds $\dot{x}_1 = 0, \dot{x}_2 = 1$.

Outline

Reverse Mode Automatic Differentiation

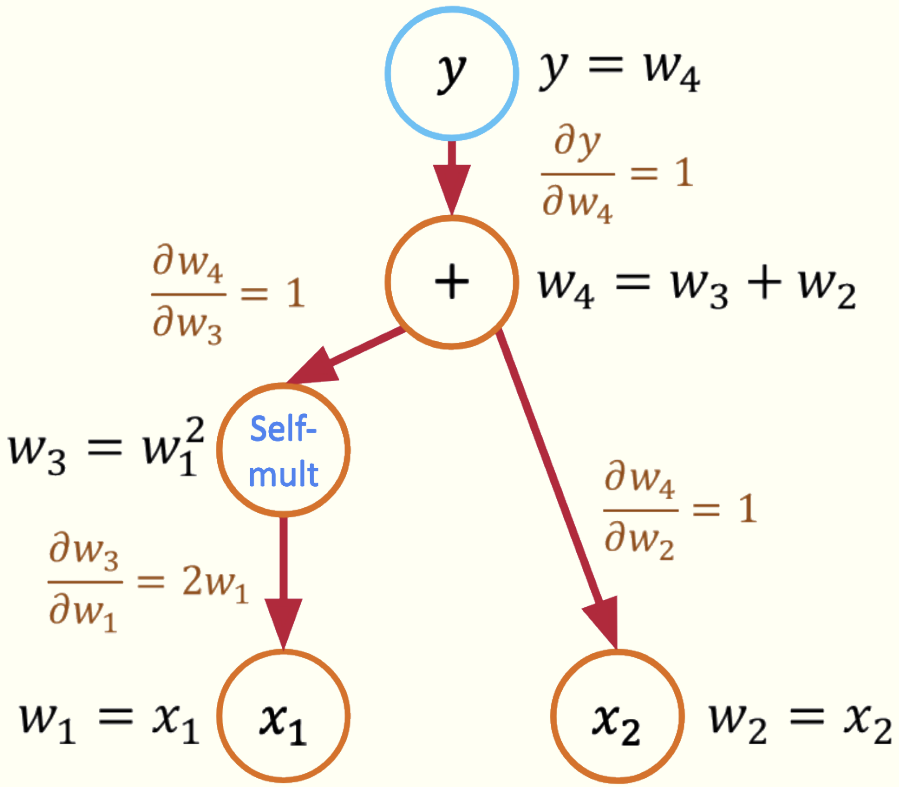
Reverse mode AD (or **backpropagation**) computes derivatives by propagating information **backward** from the output.

Algorithm:

1. **Forward Pass:** Evaluate the function and store all intermediate values.
2. **Backward Pass:** Starting from the output, use the chain rule to propagate "adjoints" ($\bar{v} = \frac{\partial y}{\partial v}$) backward through the graph.

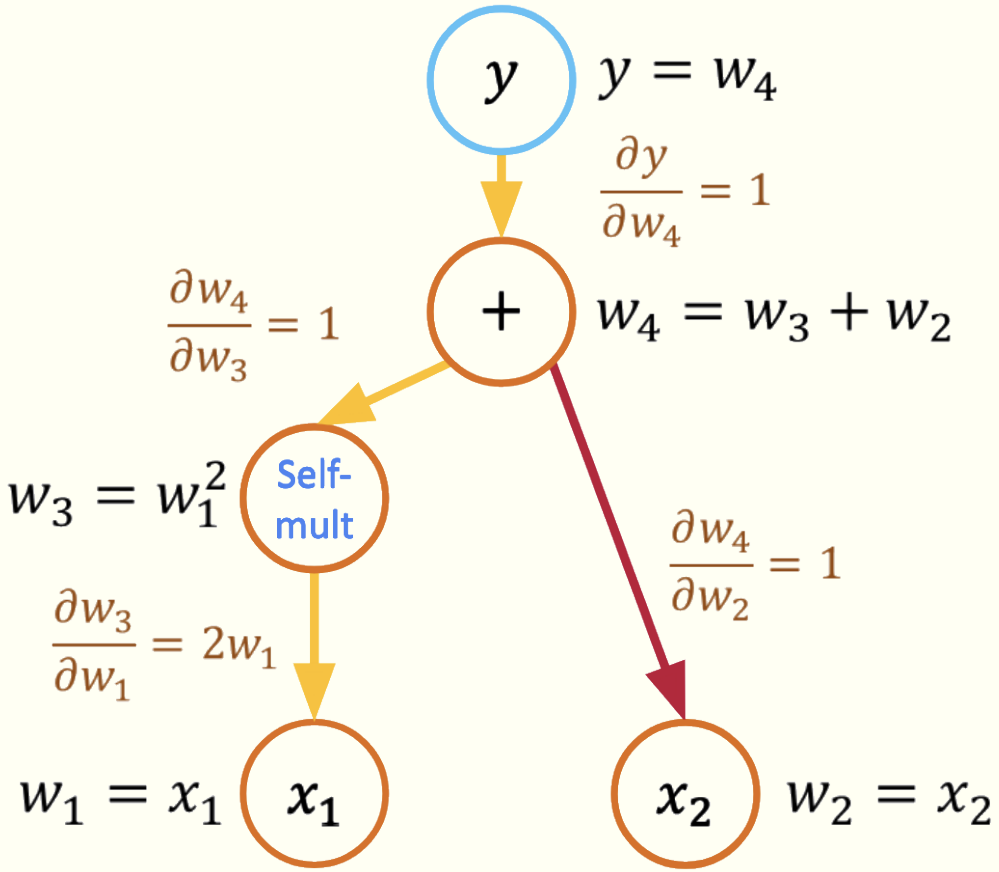
Reverse Mode Automatic Differentiation

Reverse mode AD (or **backpropagation**) computes derivatives by propagating information **backward** from the output.



Reverse Mode: The Power of the Chain Rule

The backward pass systematically applies the chain rule.
Let's trace the computation for $y = x_1^2 + x_2$:



$$\nabla y = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}\right)$$

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial w_4}\frac{\partial w_4}{\partial w_3}\frac{\partial w_3}{\partial w_1}$$

$$\frac{\partial y}{\partial x_1} = 1 * 1 * 2x_1 = 2x_1$$

Original function: $y = x_1^2 + x_2$

Reverse Mode: One Pass for All Gradients

The magic of reverse mode: it computes **all** partial derivatives in a single backward pass.

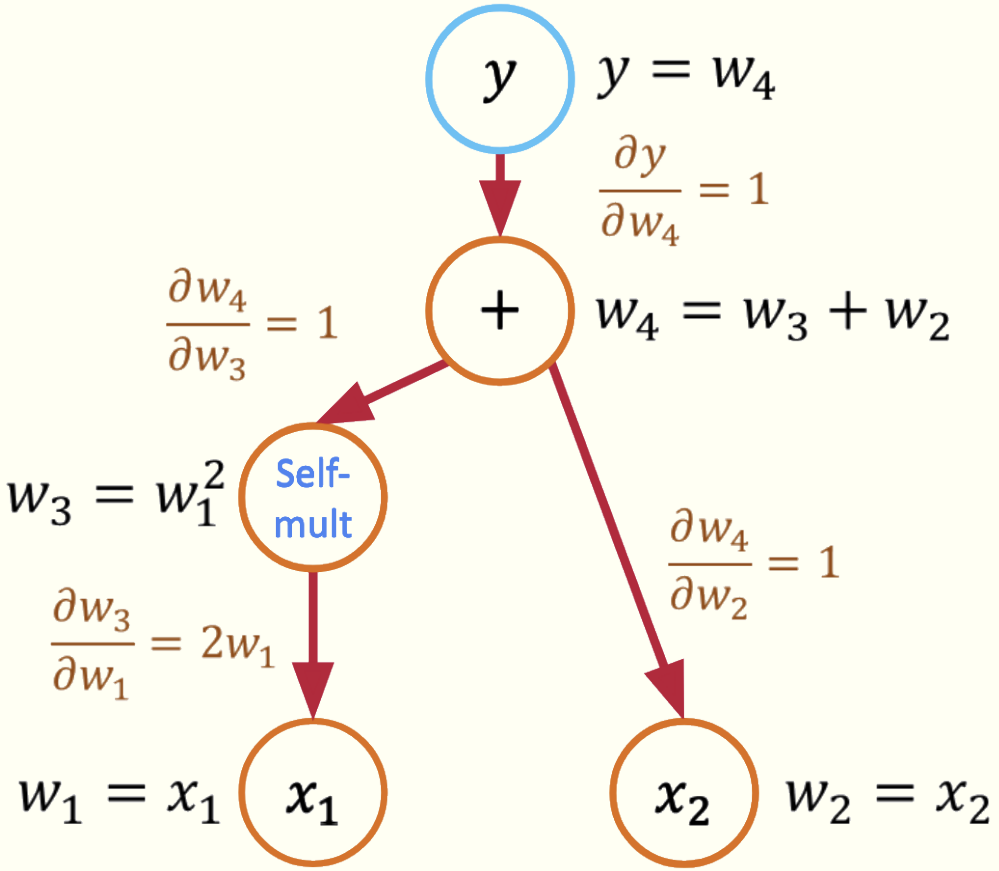
1. Forward Pass & Seed Output

Evaluate $y = x_1^2 + x_2$ and set $\bar{y} = \frac{\partial y}{\partial y} = 1$.

2. Backward Pass

- $\bar{v}_1 = \bar{y} \cdot \frac{\partial y}{\partial v_1} = 1 \cdot 1 = 1$
- $\bar{x}_2 = \bar{y} \cdot \frac{\partial y}{\partial x_2} = 1 \cdot 1 = 1 \implies \frac{\partial y}{\partial x_2} = 1$
- $\bar{x}_1 = \bar{v}_1 \cdot \frac{\partial v_1}{\partial x_1} = 1 \cdot 2x_1 = 2x_1 \implies \frac{\partial y}{\partial x_1} = 2x_1$

Reverse Mode: One Pass for All Gradients



$$y = x_1^2 + x_2$$

$$\nabla y = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}\right)$$

$$\boxed{\nabla y = (2x_1, 1)}$$

We can find the gradient with respect to the output y

Using $x_1 = 2, x_2 = 3$:

$$\nabla y = (4, 1)$$

Outline

When to Use Forward vs. Reverse Mode

The choice depends on the dimensions of your function $f : \mathbb{R}^n \to \mathbb{R}^m$.

Forward Mode

- **Cost:** One pass per *input* variable.
- **Efficient when:** Few inputs, many outputs ($n \ll m$).
- **Computes:** Jacobian-vector products ($J \cdot v$).

Reverse Mode

- **Cost:** One pass per *output* variable.
- **Efficient when:** Many inputs, few outputs ($n \gg m$).
- **Computes:** Vector-Jacobian products ($v^T \cdot J$).

Machine Learning context: We have millions of parameters (inputs, n) and a single scalar loss function (output, $m = 1$).

Reverse mode is the clear winner!

Outline

AD in PyTorch: A Simple Example

PyTorch's 'autograd' engine is built on reverse-mode AD.

```
import torch

# Define variables that require gradients
x1 = torch.tensor(2.0, requires_grad=True)
x2 = torch.tensor(3.0, requires_grad=True)

# Define the function
y = x1**2 + x2

# Compute gradients using reverse mode AD
y.backward()

# Access the computed gradients
# dy/dx1 = 2*x1 = 4.0
print(f"dy/dx1: {x1.grad.item()}")
# dy/dx2 = 1.0
print(f"dy/dx2: {x2.grad.item()}")
```

AD in PyTorch: Neural Network

The power of AD becomes clear with complex models. PyTorch automatically computes gradients for all network parameters.

```
import torch.nn as nn

net = SimpleNet() # 2-layer NN
x = torch.tensor([[1.0, 2.0]])
target = torch.tensor([[0.5]])

# Forward pass
output = net(x)
loss = ((output - target)**2).mean()

# Backward pass - computes gradients for ALL parameters
loss.backward()

# Access gradients
for name, param in net.named_parameters():
    print(f"{name}: grad shape {param.grad.shape}")
```

The AD Revolution

Automatic differentiation has revolutionized scientific computing by making gradients:

1. **Ubiquitous:** Available for any differentiable code.
2. **Exact:** No approximation errors (up to machine precision).
3. **Efficient:** Cost is proportional to the original function evaluation.
4. **Automatic:** No tedious and error-prone manual derivation required.

This has enabled the deep learning revolution and is now reshaping scientific discovery through differentiable programming.

Looking Forward

AD is the mathematical engine powering the new era of Scientific Machine Learning:

- Physics-Informed Neural Networks (PINNs)
- Neural Differential Equations
- Differentiable Simulators
- End-to-end optimization of entire scientific workflows

The ability to compute gradients automatically through arbitrary code is a fundamental capability that is reshaping how we approach science in the 21st century.

Questions?

Thank you!

Contact:
Krishna Kumar
krishnak@utexas.edu
University of Texas at Austin