

# Math4CS - PROJECT

## Automatic Differentiation

*Supervisor: PhD. Nguyen An Khuong*

*Student:*

Nguyen Quoc Bao	2470076
Nguyen Minh Chien	2570179
Nguyen Trong Nhat	2570472
Nguyen Duc Tai	2470321
Phan Thi Minh Tho	2470112

# Table of Contents

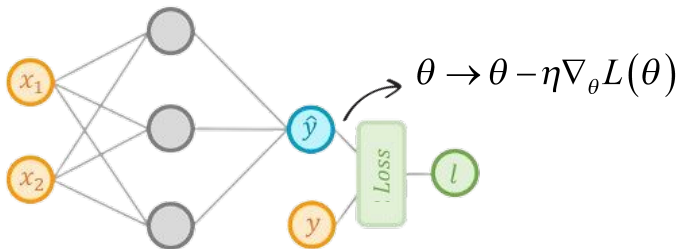
- 1 Introduction
- 2 Multi-Variables Chain Rule
- 3 Backpropagation Function in PyTorch
- 4 Detaching Computation
- 5 Gradients and Python Control Flow
- 6 Comparison between Backward and Forward Propagation
- 7 Conclusion

# Table of Contents

- 1 Introduction**
- 2 Multi-Variables Chain Rule
- 3 Backpropagation Function in PyTorch
- 4 Detaching Computation
- 5 Gradients and Python Control Flow
- 6 Comparison between Backward and Forward Propagation
- 7 Conclusion

# Introduction

- Modern deep learning models are **large and complex**, with **millions to billions of parameters**
- Training requires **optimizing a loss function** that measures prediction error



- To update parameters  $\theta$ , we need the gradients  $\nabla_{\theta} L$

# Introduction - Traditional Differentiation Methods

To compare different differentiation techniques, we use the same test function and compute its derivative at  $x = 2$

$$f(x) = x^3 + 2x^2 - 5x + 1$$

## Symbolic Differentiation

$$f'(x) = 3x^2 + 4x - 5$$

$$\Rightarrow f'(2) = 15$$

**(Exact)**

## Numerical Differentiation

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}; h = 0,01$$

$$\Rightarrow f'(2) \approx \frac{f(2,01) - f(2)}{0,01} = 15,0801$$

**(Approximate)**

# Introduction - Traditional Differentiation Methods

## Symbolic Differentiation

**Pros:** Exact, mathematically correct derivatives; **Good for small**, clean symbol expressions.

**Cons:** **Expression explosion** for complex functions; Does not work with arbitrary Python code or control flow; Cannot scale to deep neural networks.

## Numerical Differentiation

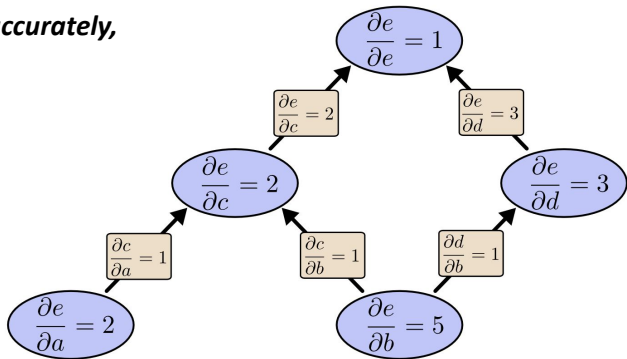
**Pros:** Easy to implement; Works without knowing the internal structure of the function.

**Cons:** High **floating-point** error; Very sensitive to the choice of  $h$ ; Requires **one forward computation per parameters** ☐ extremely slow; Not stable or efficient for large-scale optimization.

# Introduction - Automation Differentiation

**Automatic Differentiation (AD)** is a core mathematical and computational tool that enables modern machine learning - especially deep learning. It provides a way to compute derivatives accurately and automatically *by applying the chain rule over a computational graph*.

AD can compute derivatives *fast, accurately, automatically, every training step*



# List of Content

- 1 Introduction
- 2 Multi-Variables Chain Rule**
- 3 Backpropagation Function in PyTorch
- 4 Detaching Computation
- 5 Gradients and Python Control Flow
- 6 Comparison between Backward and Forward Propagation
- 7 Conclusion



# A Multi-Layer Function Composition

Considering a function of four variables, which we can make a composing many terms:

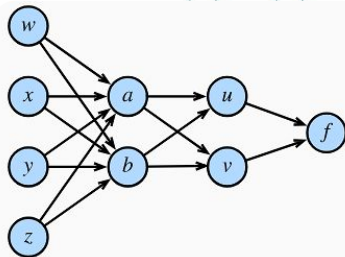
$$f(u, v) = (u + v)^2$$

$$u(a, b) = (a + b)^2, \quad v(a, b) = (a - b)^2$$

$$a(w, x, y, z) = (w + x + y + z)^2, \quad b(w, x, y, z) = (w + x - y - z)^2$$

We can fully substitute four variables into the function  $f$  to obtain a very large composite expression:

$$\begin{aligned} f(w, x, y, z) = & \left( ((w + x + y + z)^2 + (w + x - y - z)^2)^2 \right. \\ & \left. + ((w + x + y + z)^2 - (w + x - y - z)^2)^2 \right)^2 \end{aligned}$$



# The Limitations of Direct Differentiation

$$f(w, x, y, z) = \left( ((w + x + y + z)^2 + (w + x - y - z)^2)^2 \right. \\ \left. + ((w + x + y + z)^2 - (w + x - y - z)^2)^2 \right)^2$$

This expression is already extremely large, and when we try to differentiate it directly using basic derivative rules, the complexity grows even more. For instance, computing the partial derivative with respect to  $w$  gives the huge expression:

$$\frac{\partial f}{\partial w} = 2 \left( 2(2(w + x + y + z) - 2(w + x - y - z))((w + x + y + z)^2 - (w + x - y - z)^2) \right. \\ \left. + 2(2(w + x - y - z) + 2(w + x + y + z))((w + x - y - z)^2 + (w + x + y + z)^2) \right) \\ \times \left( ((w + x + y + z)^2 - (w + x - y - z)^2)^2 + ((w + x - y - z)^2 + (w + x + y + z)^2)^2 \right)$$

# Chain Rule Over Multiple Paths in a Computational Graph

Now we examine how the function  $f(u(a, b), v(a, b))$  changes when we slightly change the variable  $a$ . To do this, we ignore  $w, x, y$ , and  $z$  for now and focus only on how  $a$  affects  $f$ .

Suppose we add a small amount  $\epsilon$  to  $a$ . Then:  $f(u(a + \epsilon, b), v(a + \epsilon, b))$

Using a first-order approximation, we get:

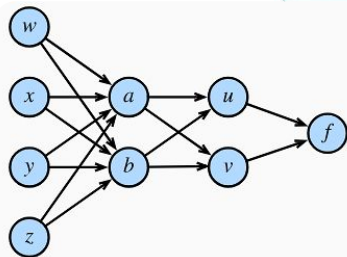
$$\approx f\left(u(a, b) + \epsilon \frac{\partial u}{\partial a}(a, b), v(a, b) + \epsilon \frac{\partial v}{\partial a}(a, b)\right)$$

Expanding this with the definition of the gradient:

$$\approx f(u(a, b), v(a, b)) + \epsilon \left[ \frac{\partial f}{\partial u} \frac{\partial u}{\partial a} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial a} \right]$$

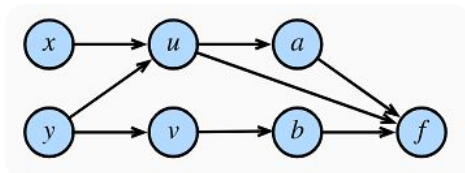
This leads to the multivariable chain rule:

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u} \frac{\partial u}{\partial a} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial a}$$



# Multi-Variables Chain Rule - Example

Consider the following computation graph:



There are multiple paths through which a change in  $y$  can influence  $f$ :

1.  $y \rightarrow u \rightarrow a \rightarrow f$
2.  $y \rightarrow u \rightarrow f$
3.  $y \rightarrow v \rightarrow b \rightarrow f$

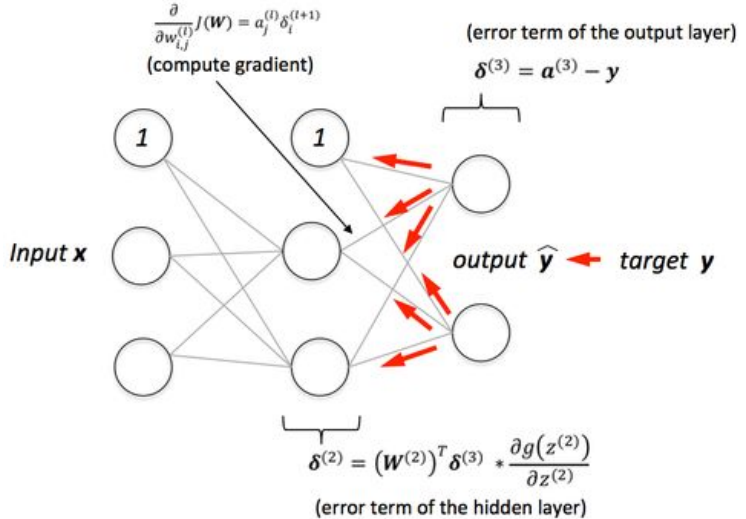
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial f}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial v} \frac{\partial v}{\partial y}$$

# Table of Contents

- 1 Introduction
- 2 Multi-Variables Chain Rule
- 3 Backpropagation Function in PyTorch**
- 4 Detaching Computation
- 5 Gradients and Python Control Flow
- 6 Comparison between Backward and Forward Propagation
- 7 Conclusion

# Backpropagation

Backpropagation is an algorithm used in neural networks to compute the gradients of the loss function with respect to the model's weights, by propagating the error backward through the network. These gradients are then used to update the weights using optimization methods like gradient descent.



# Example

$$f(u, v) = (u + v)^2$$

$$u(a, b) = (a + b)^2, \quad v(a, b) = (a - b)^2$$

$$a(w, x, y, z) = (w + x + y + z)^2, \quad b(w, x, y, z) = (w + x - y - z)^2$$

Applying Chain Rule

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial u} \frac{\partial u}{\partial w} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial w}$$

$$\frac{\partial u}{\partial w} = \frac{\partial u}{\partial a} \frac{\partial a}{\partial w} + \frac{\partial u}{\partial b} \frac{\partial b}{\partial w}$$

$$\frac{\partial v}{\partial w} = \frac{\partial v}{\partial a} \frac{\partial a}{\partial w} + \frac{\partial v}{\partial b} \frac{\partial b}{\partial w}$$

Single differentiations

$$\frac{\partial f}{\partial u} = 2(u + v), \quad \frac{\partial f}{\partial v} = 2(u + v)$$

$$\frac{\partial u}{\partial a} = 2(a + b), \quad \frac{\partial u}{\partial b} = 2(a + b)$$

$$\frac{\partial v}{\partial a} = 2(a - b), \quad \frac{\partial v}{\partial b} = -2(a - b)$$

$$\frac{\partial a}{\partial w} = 2(w + x + y + z), \quad \frac{\partial b}{\partial w} = 2(w + x - y - z)$$

# Backpropagation for non scalar variables

## 1. When `y` is a scalar

PyTorch can directly compute:

$$\frac{\partial y}{\partial x}$$

## 2. When `y` is a vector

The correct derivative is a **Jacobian matrix**:

$$J = \frac{\partial y}{\partial x}$$

This produces a **matrix**, not a vector.

## 3. Why PyTorch does not compute Jacobians automatically

Computing Jacobians is:

- Memory-intensive
- Slow

According to Pytorch document, if the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally requires specifying a gradient. It should be a tensor of matching type and shape, that represents the gradient of the differentiated function w.r.t. self.

In deep learning, we usually only need the **sum of gradients**, not the full Jacobian.

## 4. Why `.backward()` fails for vector outputs

PyTorch cannot guess how to reduce a vector to a scalar, so you must provide a gradient manually.



# Example

We consider the input vector:

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad x \in \mathbb{R}^3$$

We define the vector-valued function:

$$y = f(x) = x^T x$$

That is:

$$y = \begin{bmatrix} x_1^2 \\ x_2^2 \\ x_3^2 \end{bmatrix}$$

```
... x: [1. 2. 3.]  
    y: [1. 4. 9.]
```

```
---- PyTorch Behavior ----
```

```
Vector-Jacobian Product (v = [1,1,1]):  
Result from backward(): [2. 4. 6.]
```

```
---- Manual Full Jacobian ----
```

	dy1/dx1	dy2/dx1	dy3/dx1
y1	2.0	0.0	0.0
y2	0.0	4.0	0.0
y3	0.0	0.0	6.0

The Jacobian of the function is:

$$J = \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \\ \frac{\partial y_3}{\partial x_1} & \frac{\partial y_3}{\partial x_2} & \frac{\partial y_3}{\partial x_3} \end{bmatrix}$$

Since:

$$y_i = x_i^2$$

we have:

$$\frac{\partial y_i}{\partial x_j} = \begin{cases} 2x_i & i = j, \\ 0 & i \neq j. \end{cases}$$

Thus, the Jacobian is:

$$J = \begin{bmatrix} 2x_1 & 0 & 0 \\ 0 & 2x_2 & 0 \\ 0 & 0 & 2x_3 \end{bmatrix}$$

# Example comparison symbolic, numerical and auto differentiation

Consider the below function, test the error and the running time with 3 methods: numerical, symbolic and automatic differentiation

$$f(\mathbf{x}) = \sum_{i=1}^N x_i^2$$

Criteria	Symbolic Diff	Numerical Diff	Auto Diff
Runtime (N = 100)	0.3216s	0.000616s	0.000123s
Runtime (N = 500)	6.1724s	0.003368s	0.000114s
Error (N = 100)	0	9.95e-6	0
Error (N = 500)	0	2.27e-5	0

# Table of Contents

- 1 Introduction
- 2 Multi-Variables Chain Rule
- 3 Backpropagation Function in PyTorch
- 4 Detaching Computation**
- 5 Gradients and Python Control Flow
- 6 Comparison between Backward and Forward Propagation
- 7 Conclusion

# Detaching Computation

Detaching treats a tensor as a constant during backpropagation by severing its link to the computational history

Suppose we have

$z = x * y$

and

$y = x * x$

In normal

$\partial z / \partial x = 3x^2$

`x (input, requires_grad=True)`

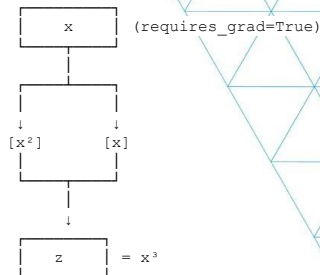
|  
| (square operation)

↓  
`y = x2`

|  
| (multiply by x)

↓  
`z = x * y = x3`

↓  
 $\partial z / \partial x = 3x^2$  (gradient flows  
through entire graph)

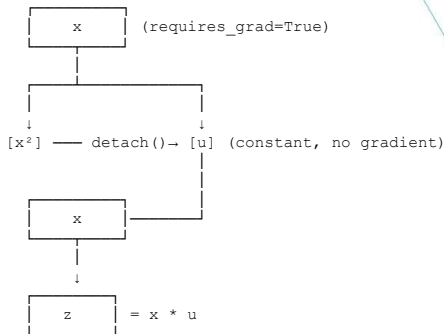


Gradient:  $\partial z / \partial x = 3x^2$

# Detaching Computation

But we want to focus on the **direct** influence of  $x$  on  $z$  rather than the influence conveyed via  $y$   
=> we can create a new variable  $u$  that takes the same value as  $y$  but its ancestor is removed  
=> `u.detach()` will have `grad_fn = None` ( $y$  have `grad_fn = MulBackward0`)

```
x (input, requires_grad=True)
|
| (square operation)
↓
y = x2
|
| .detach() ← BREAKS GRADIENT FLOW
↓
u = x2 NO grad_fn, treated as constant
|
| (multiply by x)
↓
z = x * u
|
|
↓
 $\partial z / \partial u = x$  (gradient ONLY through direct x,
NOT through u)
```



Gradient:  $\partial z / \partial x = u = x^2$  (only direct path, treating  $u$  as constant)

# Detaching Computation

1. Logic: Prevents unintended weight updates (e.g., Training GANs, Reinforcement Learning).
2. Memory : Frees up old computation graphs to prevent memory overflow (e.g., RNNs).
3. Speed: Reduces computational overhead by skipping unnecessary derivative calculations.

	Forward pass ( $x=2$ )	Backward pass ( $x=2$ )
with <code>.detach()</code>	$z = u * x = x^3 = 8$	$\partial z / \partial x = u = x^2 = 4$
without <code>.detach()</code>	$z = y * x = x^3 = 8$	$\partial z / \partial x = 3x^2 = 12$

# Table of Contents

- 1 Introduction
- 2 Multi-Variables Chain Rule
- 3 Backpropagation Function in PyTorch
- 4 Detaching Computation
- 5 Gradients and Python Control Flow**
- 6 Comparison between Backward and Forward Propagation
- 7 Conclusion

## 1.1 Problem: Control Flow

Unlike Static Graph Frameworks, PyTorch allows differentiation through Python control structures like while, if, for.

**Challenge:** The function structure changes depending on input data. How to compute derivatives?

💡 **Question:** Is  $f(a)$  a complex non-linear function?

```
def f(a):
```

```
    """
```

```
    A function demonstrating dynamic control flow (while loop and if statement).
```

```
    """
```

```
    b = a * 2
```

```
    while b.norm() < 1000:
```

```
        b = b * 2
```

```
    if b.sum() > 0:
```

```
        c = b
```

```
    else:
```

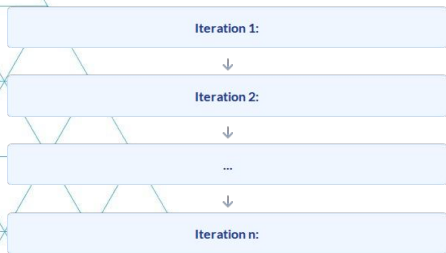
```
        c = 100 * b
```

```
    return c
```



## 1.2 "Define-by-Run" Mechanism

PyTorch uses a **Dynamic Graph** mechanism. It doesn't pre-compile the while loop. Instead, it records the operations that *\*actually happen\** during execution.



### Loop Unrolling

For the Autograd engine, the loop above is simply a sequence of consecutive multiplications.

$$a \xrightarrow{\times 2} b_0 \xrightarrow{\times 2} b_1 \xrightarrow{\times 2} b_2 \cdots \xrightarrow{\times 2} b_n \cdots \rightarrow c$$

The system "forgets" the while logic and only remembers this operation sequence for Backpropagation.

It effectively calculates the gradient using the Chain Rule for this specific sequence of multiplications:

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b_n} \cdot \frac{\partial b_n}{\partial b_{n-1}} \cdots \frac{\partial b_0}{\partial a}$$

## 1.3 Algebraic Analysis: Loops

### Step 1: Analyzing the while loop

- Initialization:  $b_0 = 2a$ .
- Loop Logic: In each iteration,  $b$  is simply multiplied by 2.

After  $n$  iterations, the value of  $b$  will be:

$$b_n = 2^n \cdot (2a) = 2^{n+1} \cdot a$$

Where  $n$  is a positive integer that depends on the value of  $a$  (to satisfy the condition  $\text{norm} < 1000$ ). However, for a specifically fixed  $a$  during a single forward pass,  $n$  is a constant.

### Step 2: Analyzing the if statement

Finally,  $c$  will equal either  $b$  or  $100b$ .

In summary:

$$c = k \cdot a$$

Where  $k$  is an aggregate constant (it can be either  $2^{n+1}$  or  $100 \cdot 2^{n+1}$ ).

### Step 3: Calculating the Derivative

The function is essentially a Linear Function with respect to the variable  $a$ :

$$f(a) = k \cdot a$$

The derivative of  $f(a)$  with respect to  $a$  is:

$$\frac{df}{da} = k$$

On the other hand, from the function equation, we can extract  $k$ :

$$k = \frac{f(a)}{a}$$

Therefore:

$$\frac{df}{da} = \frac{f(a)}{a}$$

This proves why `a.grad == d/a` always returns True (within the limits of floating-point machine precision).

```
import torch
```

```
def f(a):
```

```
    """
```

```
    A function demonstrating dynamic control flow (while loop and if statement).
```

```
    """
```

```
    b = a * 2
```

```
    while b.norm() < 1000:
```

```
        b = b * 2
```

```
    if b.sum() > 0:
```

```
        c = b
```

```
    else:
```

```
        c = 100 * b
```

```
    return c
```

```
# We verify the derivative:
```

```
a = torch.randn(size=(), requires_grad=True) # a is a random scalar/tensor
```

```
d = f(a)
```

```
d.backward()
```

```
# Check: Is a.grad equal to d / a?
```

```
print(f"Value of a: {a.item()}")
```

```
print(f"Value of f(a): {d.item()}")
```

```
print(f"Gradient calculated by python: {a.grad.item():10f}")
```

```
print(f"Gradient calculated by manual (f(a)/a): {d.item() / a.item():10f}")
```

```
print(f"a.grad == d/a: {a.grad == d / a}")
```

```
Value of a: -0.7185292840003967
```

```
Value of f(a): -147154.796875
```

```
Gradient calculated by python: 204800.000000
```

```
Gradient calculated by manual (f(a)/a): 204799.999320
```

```
a.grad == d/a: True
```

## 1.4 Old Framework

```
import tensorflow.compat.v1 as tf
import numpy as np
```

# Tắt chế độ chạy ngay (Eager execution) để mô phỏng TF 1.x chuẩn  
tf.disable\_v2\_behavior()

```
def f_static_graph(a_input):
```

```
    # 1. KHỞI TẠO BIẾN (PLACEHOLDER)
```

```
    # Trong Static Graph, ta phải tạo một "cái xô" rỗng để hứng dữ liệu sau này
    a = tf.placeholder(dtype=tf.float32, shape=(), name='a')
```

```
    # b = a * 2
```

```
    b_init = a * 2
```

```
    # 2. VÒNG LẶP WHILE (RẤT PHỨC TẠP)
```

```
    # Phải định nghĩa hàm điều kiện và hàm thân vòng lặp | riêng biệt
```

```
    def condition(b):
```

```
        return tf.norm(b) < 1000
```

```
    def body(b):
```

```
        return b * 2
```

```
    # Dùng tf.while_loop để gán node vòng lặp vào đồ thị
```

```
    b_loop = tf.while_loop(condition, body, [b_init])
```

```
    # 3. CẤU LỆNH IF/ELSE
```

```
    # Không được dùng "if b_loop.sum() > 0".
```

```
    # Phải dùng tf.cond và định nghĩa 2 hàm con cho 2 nhánh.
```

```
    def true_fn():
```

```
        return b_loop
```

```
    def false_fn():
```

```
        return 100 * b_loop
```

```
    c = tf.cond(tf.reduce_sum(b_loop) > 0, true_fn, false_fn)
```

```
    # 4. TÍNH GRADIENT
```

```
    # Phải khai báo graph tính đạo hàm ngay lúc xây dựng
```

```
    grads = tf.gradients(c, a)
```

```
    return a, c, grads
```

```
# --- PHẦN CHẠY CHƯƠNG TRÌNH (SESSION) ---
```

```
# Xây dựng đồ thị
```

```
a_ph, c_op, grad_op = f_static_graph(None)
```

```
# Tạo phiên làm việc (Session) để chạy đồ thị
```

```
with tf.Session() as sess:
```

```
    # Tạo dữ liệu giả lập (random input)
```

```
    input_val = np.random.randn()
```

```
# CHẠY: Bơm dữ liệu (feed_dict) vào placeholder để lấy kết quả
```

```
c_val, grad_val = sess.run([c_op, grad_op], feed_dict={a_ph: input_val})
```

```
print(f"Input a: {input_val}")
```

```
print(f"Output f(a): {c_val}")
```

```
print(f"Gradient: {grad_val[0]}")
```

```
# Kiểm chứng lý thuyết: grad == f(a) / a
```

```
print(f"Kiểm chứng (f(a)/a): {c_val / input_val}")
```

Instructions for updating:

non-resource variables are not supported in the long term

Input a: 0.48851716447341414

Output f(a): 1000.483154296875

Gradient: 2048.0

Kiểm chứng (f(a)/a): 2048.0

## 1.5 Comparison

Feature	Legacy Frameworks (Static Graph - e.g., TF 1.x)	Modern Frameworks (Dynamic Graph - PyTorch)
Philosophy	<b>Define-and-Run:</b> You must define the entire graph structure first (build a fixed "factory") before feeding data into it.	<b>Define-by-Run:</b> Computation happens immediately as code is executed (on-the-fly). You calculate as you define.
Control Flow (Loops & Branching)	<b>Complex:</b> Requires specific APIs (e.g., <code>tf.nn.dynamic_rnn</code> , <code>tf.nn.dynamic_rnn</code> ). Native Python logic ( <code>if</code> , <code>while</code> ) is ignored during graph compilation.	<b>Simple:</b> Uses native Python control flow ( <code>if</code> , <code>while</code> , <code>for</code> ) directly.
Debugging	<b>Difficult:</b> You cannot use <code>print()</code> or breakpoints inside the graph definition (body/condition functions) because the code is not actually running data yet.	<b>Easy:</b> You can use <code>print()</code> or standard debuggers inside loops to inspect intermediate values in real-time.
Gradient Calculation	<b>Symbolic:</b> You must explicitly define gradient nodes ( <code>tf.nn.dynamic_rnn</code> ) within the graph before starting the Session.	<b>Imperative:</b> You can call <code>.backward()</code> at any time after the forward pass is complete.
Coding Style	<b>Declarative:</b> The code can feel unintuitive or "foreign," distinct from standard Python programming.	<b>Imperative:</b> The code feels natural and "Pythonic," similar to writing standard algorithms.

# Table of Contents

- 1 Introduction
- 2 Multi-Variables Chain Rule
- 3 Backpropagation Function in PyTorch
- 4 Detaching Computation
- 5 Gradients and Python Control Flow
- 6 Comparison between Backward and Forward Propagation**
- 7 Conclusion

# Backward and forward comparison

## Forward differentiation calculation (x→f)

Example function:

$$f(x) = \log(x^2) \sin x + x^{-1}$$

Define the intermediate variables:

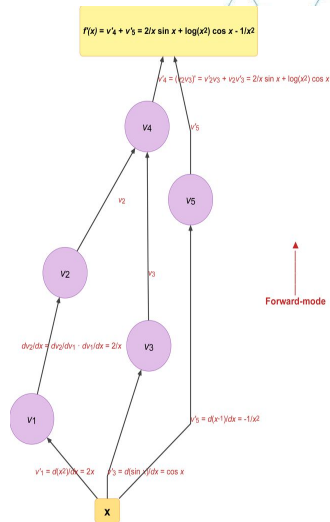
- $v_1 = x^2$
- $v_2 = \log(v_1)$
- $v_3 = \sin(x)$
- $v_4 = v_2 \cdot v_3$
- $v_5 = x^{-1}$
- $f = v_4 + v_5$

Forward Differentiation (x → f)

$$\begin{aligned}v'_1 &= \frac{dv_1}{dx} = \frac{d(x^2)}{dx} = 2x \\v'_2 &= \frac{dv_2}{dx} = \frac{dv_2}{dv_1} \cdot \frac{dv_1}{dx} = \frac{d(\log(v_1))}{dv_1} \cdot v'_1 = \frac{1}{v_1} \cdot 2x = \frac{2}{x} \\v'_3 &= \frac{dv_3}{dx} = \frac{d(\sin x)}{dx} = \cos x \\v'_4 &= (v_2 v_3)' = v'_2 v_3 + v_2 v'_3 = \frac{2}{x} \sin x + \log(x^2) \cos x \\v'_5 &= \frac{dv_5}{dx} = \frac{d(x^{-1})}{dx} = -\frac{1}{x^2}\end{aligned}$$

Derivative of (f):

$$\begin{aligned}f'(x) &= v'_4 + v'_5 \\f'(x) &= \frac{2}{x} \sin x + \log(x^2) \cos x - \frac{1}{x^2}\end{aligned}$$



# Backward and forward differentiation comparison

## Backward differentiation calculation (f→x)

Backward Differentiation ( $f \rightarrow x$ )

Start with:

$$\bar{f} = 1$$

$$\bar{v}_4 = \bar{f} \cdot \frac{\partial f}{\partial v_4} = 1 \cdot \frac{\partial(v_4 + v_5)}{\partial v_4} = 1 \cdot \left( \frac{\partial v_4}{\partial v_4} + \frac{\partial v_5}{\partial v_4} \right) = 1 \cdot (1 + 0) = 1$$

$$\bar{v}_5 = \bar{f} \cdot \frac{\partial f}{\partial v_5} = 1 \cdot \frac{\partial(v_4 + v_5)}{\partial v_5} = 1 \cdot \left( \frac{\partial v_4}{\partial v_5} + \frac{\partial v_5}{\partial v_5} \right) = 1 \cdot (0 + 1) = 1$$

$$\bar{v}_2 = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_2} = 1 \cdot \frac{\partial(v_2 \cdot v_3)}{\partial v_2} = 1 \cdot \left( \frac{\partial v_2}{\partial v_2} \cdot v_3 + v_2 \cdot \frac{\partial v_3}{\partial v_2} \right) = 1 \cdot (1 \cdot v_3 + v_2 \cdot 0) = v_3$$

$$\bar{v}_3 = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_3} = 1 \cdot \frac{\partial(v_2 \cdot v_3)}{\partial v_3} = 1 \cdot \left( v_2 \cdot \frac{\partial v_3}{\partial v_3} + v_3 \cdot \frac{\partial v_2}{\partial v_3} \right) = 1 \cdot (v_2 \cdot 1 + v_3 \cdot 0) = v_2$$

$$\bar{v}_1 = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} = v_3 \cdot \frac{\partial(\log v_1)}{\partial v_1} = v_3 \cdot \frac{1}{v_1} = \frac{v_3}{v_1}$$

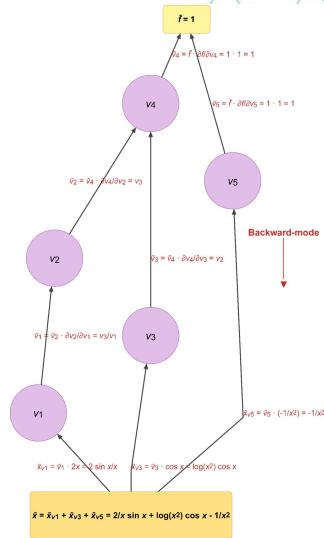
$$\bar{x}_{v_5} = \bar{v}_5 \cdot \frac{\partial v_5}{\partial x} = 1 \cdot \left( -\frac{1}{x^2} \right) = -\frac{1}{x^2}$$

$$\bar{x}_{v_3} = \bar{v}_3 \cdot \frac{\partial v_3}{\partial x} = v_2 \cdot \cos x = \log(x^2) \cos x$$

$$\bar{x}_{v_1} = \bar{v}_1 \cdot \frac{\partial v_1}{\partial x} = \frac{v_3}{v_1} \cdot 2x = \frac{2}{x} \sin x$$

Final Gradient

$$\bar{x} = \bar{x}_{v_1} + \bar{x}_{v_3} + \bar{x}_{v_5} = \frac{2}{x} \sin x + \log(x^2) \cos x - \frac{1}{x^2}$$



# Backward and forward differentiation comparison

## Backward vs Forward Differentiation

Criteria	Forward differentiation	Backward differentiation
Direction	Inputs → Outputs (same direction as function evaluation)	Outputs → Inputs (reverse direction, requires forward pass first)
Execution Process	Computes derivatives with function evaluation in a single pass	Two phases: (1) Forward pass records intermediate values onto a "tape", (2) Backward pass use the tape to compute gradients
Computational Cost	$\propto n$ (number of inputs). Efficient when $n \ll m$	$\propto m$ (number of outputs). Efficient when $m \ll n$
Memory Usage	Low - only tracks partial derivatives at each step	High - must store all intermediate values
Implementation	Simple - follows the natural flow of computation	More complex - requires tape recording mechanism
Typical Applications	Sensitivity analysis, control systems, cases where $n \approx m$ or $n < m$	Neural networks, large-scale optimization ( $n \gg 1$ , $m = 1$ for loss function)



# Table of Contents

- 1 Introduction
- 2 Multi-Variables Chain Rule
- 3 Backpropagation Function in PyTorch
- 4 Detaching Computation
- 5 Gradients and Python Control Flow
- 6 Comparison between Backward and Forward Propagation
- 7 Conclusion**

# Conclusion

- Automatic Differentiation (AD) provides ***an efficient and accurate way to compute derivatives of complex functions.***
  - AD systematically applies the chain rule through a computational graph.
  - It avoids expression explosion seen in symbolic differentiation and eliminates approximation errors of numerical differentiation.
  - AD delivers machine-precision gradients with minimal redundant computation. It forms the foundation of backpropagation in modern deep learning.
  - AD enables training of large-scale neural networks with millions or billions of parameters.
- Understanding AD explains how gradients flow through neural networks. This clarity helps ensure effective and reliable model training

# References

- Dive into Deep Learning - Chapter 2.5: Automatic Differentiation
- Dive into Deep Learning - Chapter 22.4: Multivariable Calculus
- MIT 18.S096 - Lecture 8: Automatic Differentiation
- Pytorch Libraries: <https://pytorch.org/>

**- THE END -**  
*Thank you for your attention*