# PRESENTATION
# AUTOMATIC DIFFERENTIATION


*Presenters:*

| | |
|---|---|
| Nguyễn Quốc Bảo | 2470076 |
| Nguyễn Minh Chiến | 2570179 |
| Nguyễn Trọng Nhật | 2570472 |
| Nguyễn Đức Tài | 2470321 |
| Phan Thị Minh Thơ | 2470112 |

VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
**HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY**

# Table of Contents

# Table of Contents

# Table of Contents

# 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
          |
          ↓
   ∂z/∂x = 3x²  (gradient flows
through entire graph)
```

```
┌──────────┐
│    x     │ (requires_grad=True)
└────┬─────┘
     │
┌────┴─────┐
│          │
↓          ↓
[x²]      [x]
│          │
└────┬─────┘
     │
     ↓
┌─────────┐
│    z    │ = x³
└─────────┘

Gradient: ∂z/∂x = 3x²
```

# 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
          |
          ↓
       ∂z/∂u = x  (gradient ONLY through direct x,
    NOT through u)
```

```
    ┌─────────┐
    │    x    │  (requires_grad=True)
    └─────────┘
         │
    ┌────┴────────────┐
    │                 │
    ↓                 ↓
  [x²] ── detach()→ [u]  (constant, no gradient)
                      │
                      │
    ┌─────────┐       │
    │    x    │───────┘
    └─────────┘
         │
         ↓
    ┌────────┐
    │   z    │  = x * u
    └────────┘
```

Gradient: ∂z/∂x = u = x² (only direct path, treating u as constant)

# 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
        |
        ↓
      ∂z/∂u = x  (gradient ONLY through direct x,
  NOT through u)
```

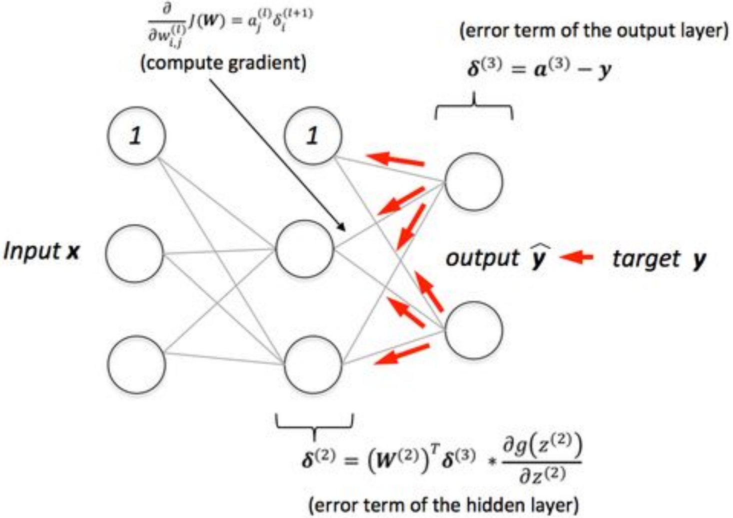
```
   ┌───────┐
   │   x   │ (requires_grad=True)
   └───────┘
       │
   ┌───┴────────────┐
   │                │
   ↓                ↓
[x²] ── detach()→ [u] (constant, no gradient)
                    │
   ┌───────┐        │
   │   x   │ ───────┘
   └───────┘
       │
       ↓
   ┌───────┐
   │   z   │ = x * u
   └───────┘
```

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

# Table of Contents

# 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

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

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

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

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

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

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

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

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

# 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:

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

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

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:

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

# Table of Contents

# 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?**

```python
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.

Iteration 1:

↓

Iteration 2:

↓

...

↓

Iteration n:

## 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 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 DerivativeThe 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 $k$:

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

ConclusionTherefore:

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

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

```python
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 caculated by python: {a.grad.item():10f}")
print(f"Gradient caculated by manual (f(a)/a): {d.item() / a.item():10f}")
print(f"a.grad == d/a: {a.grad == d / a}")
```

```
Value of a: -0.09854736179113388
Value of f(a): -161460.0
Gradient caculated by python: 1638400.000000
Gradient caculated by manual (f(a)/a): 1638400.024774
a.grad == d/a: True
```

# 1.4 Old Framework

```python
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 (condition) và hàm thân vòng lặp (body) riêng biệt
    def condition(b):
        return tf.norm(b) < 1000

    def body(b):
        return b * 2

    # Dùng tf.while_loop để gắn node vòng lặp vào đồ thị
    b_loop = tf.while_loop(condition, body, [b_init])

    # 3. CÂU 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)
```

```python
    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}")
```

```
WARNING:tensorflow:From /usr/local/lib/python3.12/dist-packages/tensorflow/python/compat/v2_compat.py:98: disable_resource_variable
Instructions for updating:
non-resource variables are not supported in the long term
Input a: 0.488517164473414l4
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 | **Define-and-Run:** You must define the entire graph structure first (build a fixed "factory") before feeding data into it. | **Define-by-Run:** Computation happens immediately as code is executed (on-the-fly). You calculate as you define. |
| Control Flow (Loops & Branching) | **Complex:** Requires specific APIs (e.g., tf.while_loop, tf.cond). Native Python logic (if, while) is ignored during graph compilation. | **Simple:** Uses native Python control flow (if, while, for) directly. |
| Debugging | **Difficult:** You cannot use print() or breakpoints inside the graph definition (body/condition functions) because the code is not actually running data yet. | **Easy:** You can use print() or standard debuggers inside loops to inspect intermediate values in real-time. |
| Gradient Calculation | **Symbolic:** You must explicitly define gradient nodes (tf.gradients) within the graph before starting the Session. | **Imperative:** You can call .backward() at any time after the forward pass is complete. |
| Coding Style | **Declarative:** The code can feel unintuitive or "foreign," distinct from standard Python programming. | **Imperative:** The code feels natural and "Pythonic," similar to writing standard algorithms. |

# Table of Contents

# Backward and forward comparison

## Forward differentiation calculation (x–>f)

Example function:

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

Define the intermediate variables:

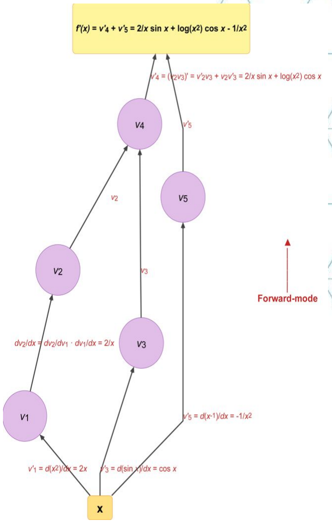
- $v_1 = x^2$
- $v_2 = \log(v_1)$
- $v_3 = \sin(x)$
- $v_4 = v_2 \cdot v_3$
- $v_5 = x^{-1}$
- $f = v_4 + v_5$

Forward Differentiation (x ⟶ f)

$$v_1' = \frac{dv_1}{dx} = \frac{d(x^2)}{dx} = 2x$$
$$v_2' = \frac{dv_2}{dx} = \frac{dv_2}{dv_1} \cdot \frac{dv_1}{dx} = \frac{d(\log(v_1))}{dv_1} \cdot v_1' = \frac{1}{v_1} \cdot 2x = \frac{2}{x}$$
$$v_3' = \frac{dv_3}{dx} = \frac{d(\sin x)}{dx} = \cos x$$
$$v_4' = (v_2 v_3)' = v_2' v_3 + v_2 v_3' = \frac{2}{x}\sin x + \log(x^2)\cos x$$
$$v_5' = \frac{dv_5}{dx} = \frac{d(x^{-1})}{dx} = -\frac{1}{x^2}$$

**Derivative of (f):**

$$f'(x) = v_4' + v_5'$$
$$f'(x) = \frac{2}{x}\sin x + \log(x^2)\cos x - \frac{1}{x^2}$$



# Backward and forward differentiation comparison

## Backward differentiation calculation (f–>x)

### Backward Differentiation (f ⟶ x)

Start with:

$$\bar{f} = 1$$

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

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

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

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

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

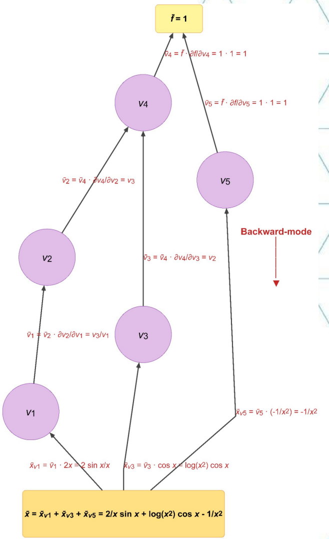
$$\bar{x}_{v_5} = \bar{v}_5 \cdot \frac{\partial v_5}{\partial x} = 1 \cdot \left( -\frac{1}{x^2} \right) = -\frac{1}{x^2}$$

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

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

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 | Outputs → Inputs |
| Computation Cost | $\propto$ Inputs (n), good if n << m | $\propto$ Outputs (m), good if m << n |
| Purpose | Compute gradients for inputs, suitable for functions with few inputs and many outputs | Compute gradients for many inputs, suitable for functions with many inputs and few outputs (e.g., loss function in ML) |
| Applications | Functions with few inputs and many outputs | Neural networks, machine learning, large-scale optimization |
| Implementation | Simple, no need to store intermediate values | More complex, requires storing intermediate values (tape) |
| Memory | Low | High (store intermediate values) |

# Table of Contents

# References

# - THE END -

*Thank you for your attention*