ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Mathematics Foundations for Computer Science - CO5263

Automatic Differentiation

Giảng viên TS. Nguyễn An Khương
TS. Nguyễn Tuấn Anh

Nhóm 3: Lê Tử Quân
Phan An Đông
Nguyễn Lê Gia Hinh
Lương Thanh Hiếu Bảo
Trần Hữu Anh Triết
Ủ Minh Quân

Thành phố Hồ Chí Minh, Tháng 5/2025

CSE

Content

WHAT AUTODIFF IS NOT?

CSE

1 WHAT AUTODIFF IS NOT ?

Autodiff is not finite differences

Autodiff is not symbolic differentation

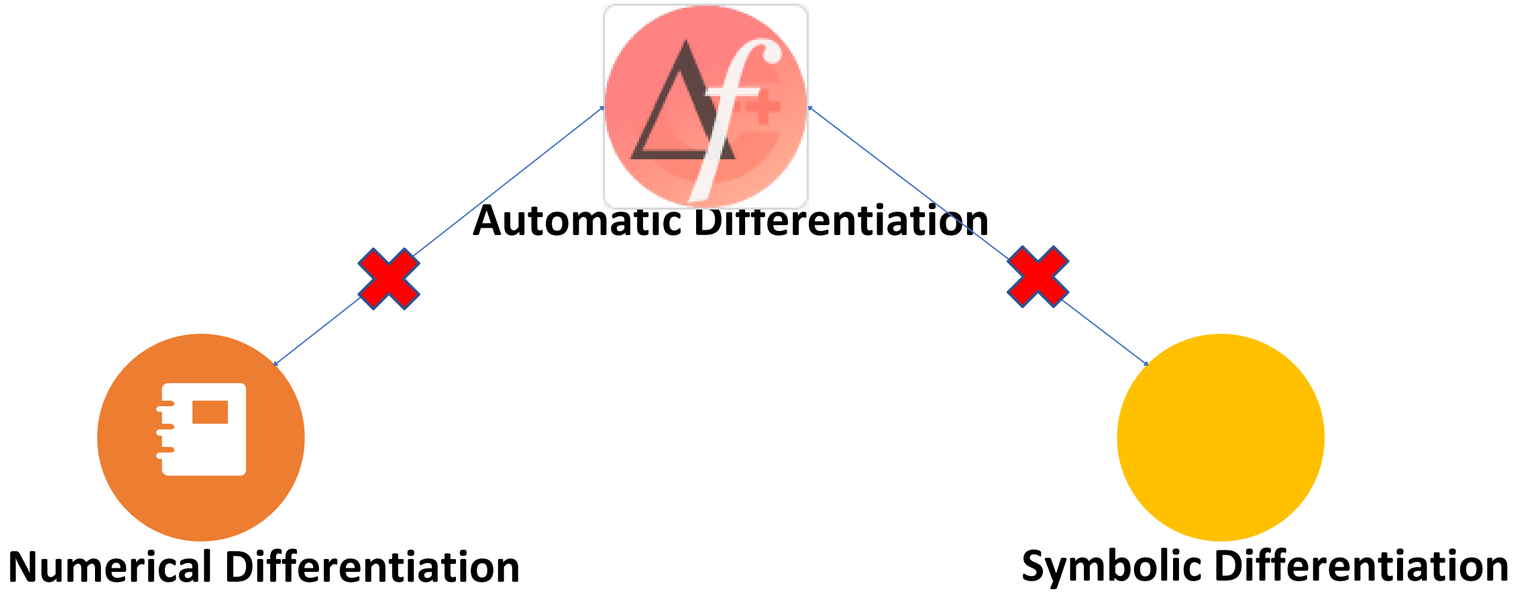
What autodiff is ?

Types of Autodiff

1 WHAT AUTODIFF IS NOT

AUTODIFF IS NOT FINITE DIFFERENTIATE

Without a clear introduction, many people place automatic differentiation in the same category as numerical differentiation or symbolic differentiation.



1.1 WHAT AUTODIFF IS NOT

AUTODIFF IS NOT NUMERICAL DIFFERENTiATION

- Numerical differentiation estimates a derivative by applying finite-difference formulas to function values sampled at specific points.
  - For example, for a multivariate function $f : R^n \to R$, one can approximate the gradient $\left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$ using:

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h}$$

where e_i is the i-th unit vector and $h > 0$ is a small step size.

- The instability arises from truncation errors and round-off errors introduced by finite computational precision and the selection of the step size .
- While the truncation error vanishes as approaches zero, reducing increases the round-off error until it becomes the primary source of inaccuracy.

1.2 WHAT AUTODIFF IS NOT

AUTODIFF IS NOT SYMBOL DIFFERENTiATION

- Symbolic differentiation refers to the automated transformation of mathematical expressions to derive their exact derivative formulas. This is achieved by systematically applying differentiation rules, for example:

$$\frac{d}{dx}(f(x)+g(x)) \approx \frac{d}{dx}f(x)+\frac{d}{dx}g(x)$$

$$\frac{d}{dx}(f(x)g(x)) \approx \left(\frac{d}{dx}f(x)\right)g(x)+f(x)\left(\frac{d}{dx}g(x)\right)$$

- Symbolic derivatives often expand into expressions that grow exponentially in size compared to the original formula, making them impractical for efficient runtime computation of derivative values. This issue is called expression swell.

derivatives of l_n with respect to x, illustrating expression swell.

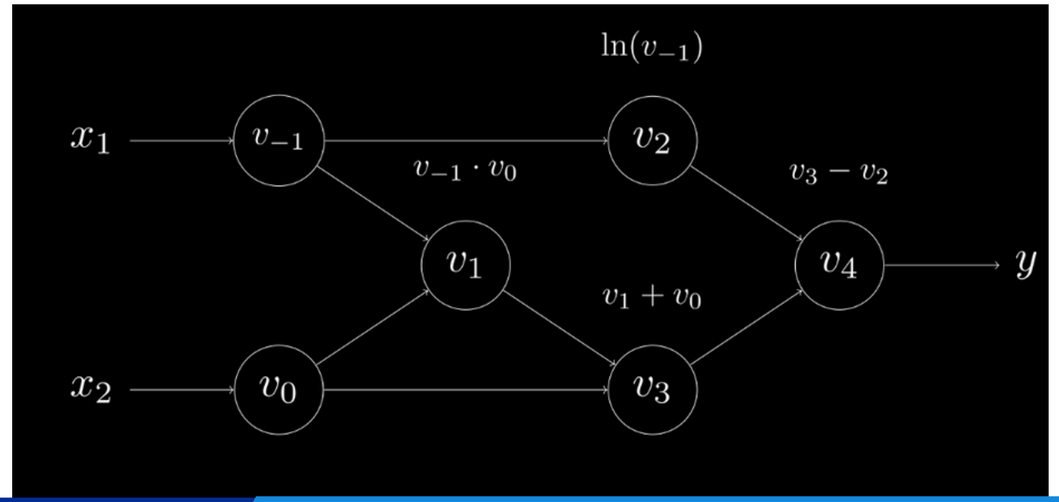
n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

1.3 WHAT IS AUTODIFF ? AUTODIFF IS AUTOMATIC DIFFENTIATION

Automatic differentiation can be viewed as an alternative way to execute a program in which the original calculations are extended to include derivative computations. Every numerical routine ultimately breaks down into a finite collection of basic operations whose derivatives are known.

To implement this, we can leverage an evaluation trace. An evaluation trace is a special table that keeps track of intermediate variables as well as the operations that created them. Every row corresponds to an intermediate variable and the elementary operation that caused it. These variables, called primals, are typically denoted v_i for functions $f: R^n \to R^m$ and follow these rules:

- Input variables: $v_{i-n} = x_i, i = 1, \dots, n$
- Intermediate variables: $v_i,\ \ i = 1, \dots, l$
- Output variables: $y_{m-i} = v_{l-i}\ , i = m-1, \dots, 0$



$$y = f(x_1, x_2) = x_1 x_2 + x_2 - \ln(x_1)$$

$$x_1 = 2, x_2 = 4$$

Forward Primal Trace	Output
$v_{-1} = x_1$	2
$v_0 = x_2$	4
$v_1 = v_{-1} v_0$	2(4) = 8
$v_2 = \ln(v_{-1})$	ln(2) = 0.693
$v_3 = v_1 + v_0$	8 + 4 = 12
$v_4 = v_3 - v_2$	12 – 0.693 = 11.307
$y = v_4$	11.307

Table 1: Forward Primal Trace

1.4 WHAT IS AUTODIFF MODE? AUTODIFF FORWARD MODE

Forward-mode AD enhances the evaluation trace by linking each primary value v_i to a tangent $\dot{v}_i$. These tangents record the partial derivative of each value relative to the selected input variable.

We'd have the following definition of tangents if we were interested in finding $\frac{\partial y}{\partial x_2}$:

$$\dot{v}_i = \frac{\partial v_i}{\partial x_2}$$

Continuing from this definition, we can build out the forward primal and forward tangent trace to compute $\frac{\partial y}{\partial x_2}$ when $x_1 = 3, x_2 = -4, \dot{x_1} = \frac{\partial x_1}{\partial x_2} = 0, and\ \dot{x_2} = \frac{\partial x_2}{\partial x_2} = 1.$

Forward Primal Trace	Output	Forward Tangent Trace	Output
$v_{-1} = x_1$	3	$\dot{v_{-1}} = \dot{x_1}$	0
$v_0 = x_2$	-4	$\dot{v_0} = \dot{x_2}$	1
$v_1 = v_{-1}v_0$	3 x (-4) = -12	$\dot{v_1} = \dot{v_{-1}}v_0 + \dot{v_0}v_{-1}$	0 x (-4) + 1 x (3) = 3
$v_2 = \ln(v_{-1})$	ln(3) = 1.10	$\dot{v_2} = \dot{v_{-1}} \times (1/v_{-1})$	0 x (1/3) = 0
$v_3 = v_1 + v_0$	-12 + -4 = -16	$\dot{v_3} = \dot{v_1} + \dot{v_0}$	3 + 1 = 4
$v_4 = v_3 - v_2$	-16 - 1.10 = -17.10	$\dot{v_4} = \dot{v_3} - \dot{v_2}$	4 – 0 = 4
$y = v_4$	-17.10	$\dot{y} = \dot{v_4}$	4

Table 2: Forward Mode Trace

1.5 WHAT IS AUTODIFF MODE? AUTODIFF REVERSE MODE

At this stage, we introduce reverse-mode AD, which mirrors forward mode in spirit but differs in technique. We start by defining adjoints $\overline{v_i}$, each of which represents the partial derivative of a specific output y_j, with respect to an intermediate variable v_i. Formally, for a function $f : R^n \rightarrow R^m$ and indices $i = 1, \dots, n$ and $j = 1, \dots, m$, we set up these adjoints as follows:

$$\overline{v_i} = \frac{\partial y_j}{\partial v_j}$$

Forward Primal Trace	Output	Reverse Adjoint Trace	Output
$v_{-1} = x_1$	3	$\bar{v}_{-1} = \bar{x}_1 = \bar{v}_2 \times (1/v_{-1}) + \overline{v_1} \times v_0$	-1 x (1/3) + 1 x (-4) = -4.33
$v_0 = x_2$	-4	$\bar{v}_0 = \bar{x}_2 = \bar{v}_3 \times 1 + \bar{v}_1 \times v_{-1}$	1 x 1 + 1 x 3 = 4
$v_1 = v_{-1}v_0$	3 x (-4) = -12	$\bar{v}_1 = \bar{v}_3 \times 1$	1 x 1 = 1
$v_2 = \ln(v_{-1})$	ln(3) = 1.10	$\bar{v}_2 = \bar{v}_4 \times -1$	1 x -1 = -1
$v_3 = v_1 + v_0$	-12 + -4 = -16	$\bar{v}_3 = \bar{v}_4 \times 1$	1 x 1 = 1
$v_4 = v_3 - v_2$	-16 - 1.10 = -17.10	$\bar{v}_4 = \bar{y}$	1
$y = v_4$	-17.10	$\bar{y}$	1

Table 3: Reverse Mode Trace

BACKPROPAGATION ALGORITHM

CSE

2 Backpropagation

Backpropagation Definition

Backpropagation Example

Backpropagation Application

2.1 Backpropagation Definition

Backpropagation Algorithm is the central in traning neural networks with characteristics:

- It is an algorithm for computing gradients
- It is a special case of reverse-mode automatic differentiation (AD) applied to a computational graph with a scalar-output.
- Backpropagation refers to the whole process of training an artificial neural network using multiple backpropagation steps, each of which computes gradients and uses them to perform a Gradient Descent step. In contrast, reverse-mode auto diff is simply a technique used to compute gradients efficiently and it happens to be used by backpropagation.

  => The backpropagation algorithm is a method for efficiently computing gradients of a loss function with respect to the parameters (weights and biases) of a neural network. **It uses reverse-mode automatic differentiation to propagate errors backward through the network**, enabling gradient-based optimization (e.g., gradient descent)

2.2 Backpropagation Example

- Input layer: 2 neurons (x_1, x_2)
- Hidden layer: 2 neurons (h_1, h_2) with sigmoid activation
- Output layer: 1 neuron (y) with sigmoid activation
- Loss function: Mean Squared Error (MSE)

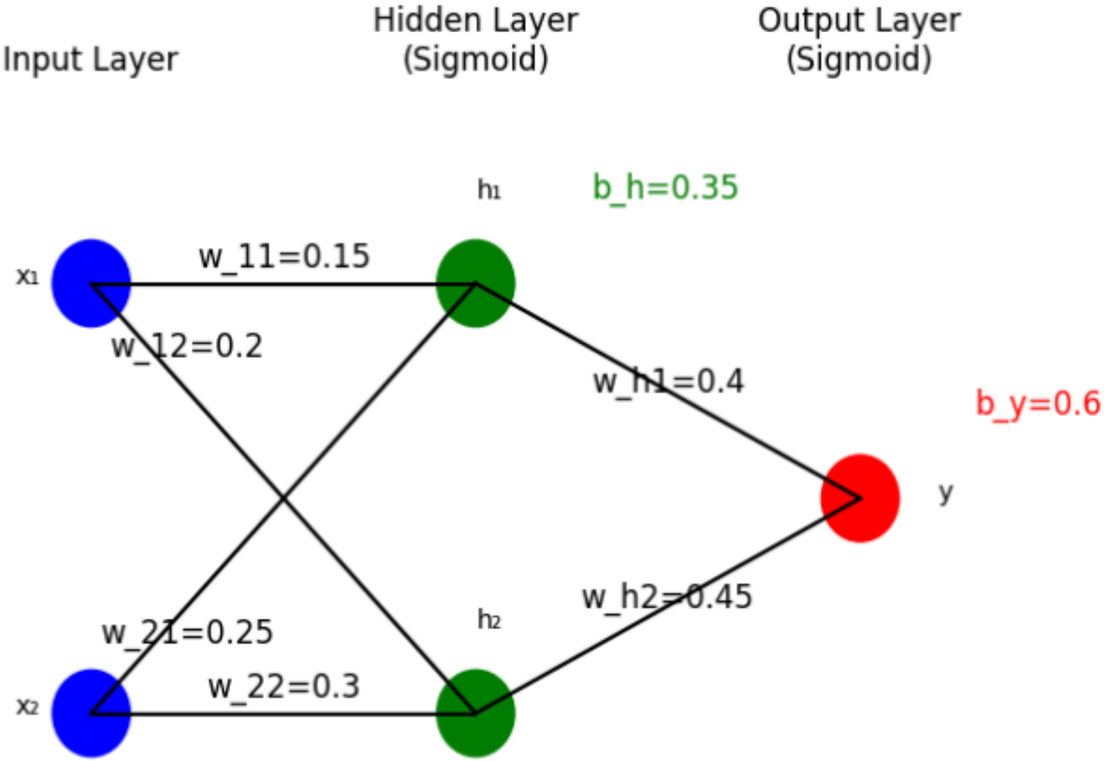
We'll compute gradients and update weights for one training example with Input: $x = [x_1, x_2] = [0.5, 0.1]$. Target output: $t = 0.8$

Weights and biases (randomly initialized):

- Input to hidden: $w_{11} = 0.15, w_{12} = 0.2, w_{21} = 0.25, w_{22} = 0.3$
- Hidden to output: $w_{h1} = 0.4, w_{h2} = 0.45$
- Biases: $b_h = 0.35$ (hidden), $b_y = 0.6$ (output)
- Learning rate: $\eta = 0.5$
- Activation function: Sigmoid, $\sigma(z) = \frac{1}{1+e^{-z}}$,
- Derivative: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

The network structure is:

- Input: x_1, x_2
- Hidden: $h_1 = \sigma(w_{11}x_1 + w_{21}x_2 + b_h), h_2 = \sigma(w_{12}x_1 + w_{22}x_2 + b_h)$
- Output: $y = \sigma(w_{h1}h_1 + w_{h2}h_2 + b_y)$
- Loss: $E = \frac{1}{2}(t - y)^2$



2.2 Backpropagation Example

Hidden Layer: Step 1: Forward Pass. Compute the activations of all neurons.

Net input to h_1:

$$z_{h1} = w_{11}x_1 + w_{21}x_2 + b_h = (0.15)(0.5) + (0.25)(0.1) + 0.35 = 0.075 + 0.025 + 0.35 = 0.45$$

$$h_1 = \sigma(z_{h1}) = \frac{1}{1 + e^{-0.45}} \approx 0.6106$$

Net input to h_2:

$$z_{h2} = w_{12}x_1 + w_{22}x_2 + b_h = (0.2)(0.5) + (0.3)(0.1) + 0.35 = 0.1 + 0.03 + 0.35 = 0.48$$

$$h_2 = \sigma(z_{h2}) = \frac{1}{1 + e^{-0.48}} \approx 0.6177$$

Output Layer:

Net input to y:

$$z_y = w_{h1}h_1 + w_{h2}h_2 + b_y = (0.4)(0.6106) + (0.45)(0.6177) + 0.6 \approx 0.2442 + 0.2780 + 0.6 = 1.1222$$

$$y = \sigma(z_y) = \frac{1}{1 + e^{-1.1222}} \approx 0.7546$$

Loss:

$$E = \frac{1}{2}(t - y)^2 = \frac{1}{2}(0.8 - 0.7546)^2 = \frac{1}{2}(0.0454)^2 \approx 0.00103$$

2.2 Backpropagation Example

Step 2: Backward Pass (Compute Gradients)

We compute the partial derivatives of the loss E with respect to each weight and bias, starting from the output layer and moving backward.

Output Layer Gradients:

Error term for output neuron:

$$\delta_y = \frac{\partial E}{\partial y} \cdot \sigma'(z_y)$$

Loss derivative: $\frac{\partial E}{\partial y} = -(t - y) = -(0.8 - 0.7546) = -0.0454$

Sigmoid derivative:

$$\sigma'(z_y) = y(1 - y) = 0.7546(1 - 0.7546) \approx 0.1851$$

$$\delta_y = -0.0454 \cdot 0.1851 \approx -0.0084$$

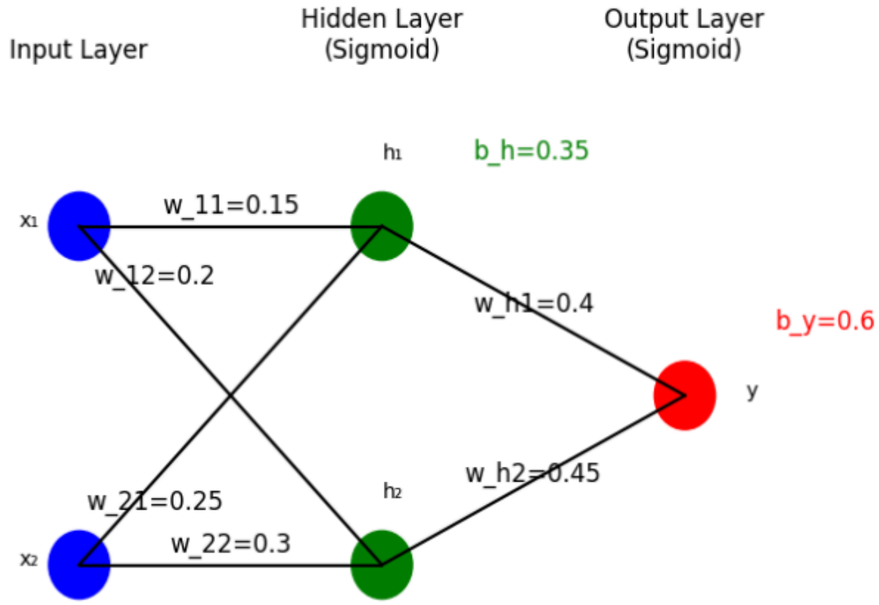
Gradients for weights to output:

$$\frac{\partial E}{\partial w_{h1}} = \delta_y \cdot h_1 = -0.0084 \cdot 0.6106 \approx -0.00513$$

$$\frac{\partial E}{\partial w_{h2}} = \delta_y \cdot h_2 = -0.0084 \cdot 0.6177 \approx -0.00519$$

Gradient for output bias:

$$\frac{\partial E}{\partial b_y} = \delta_y = -0.0084$$



2.2 Backpropagation Example

Step 2: Backward Pass (Compute Gradients)

Hidden Layer Gradients:

Error terms for hidden neurons:

$$\delta_{h1} = (w_{h1} \cdot \delta_y) \cdot \sigma'(z_{h1})$$

$w_{h1} \cdot \delta_y = 0.4 \cdot (-0.0084) \approx -0.00336$ Sigmoid derivative: $\sigma'(z_{h1}) = h_1(1 - h_1) = 0.6106(1 - 0.6106) \approx 0.2379$
$\delta_{h1} = -0.00336 \cdot 0.2379 \approx -0.0008$

$$\delta_{h2} = (w_{h2} \cdot \delta_y) \cdot \sigma'(z_{h2})$$

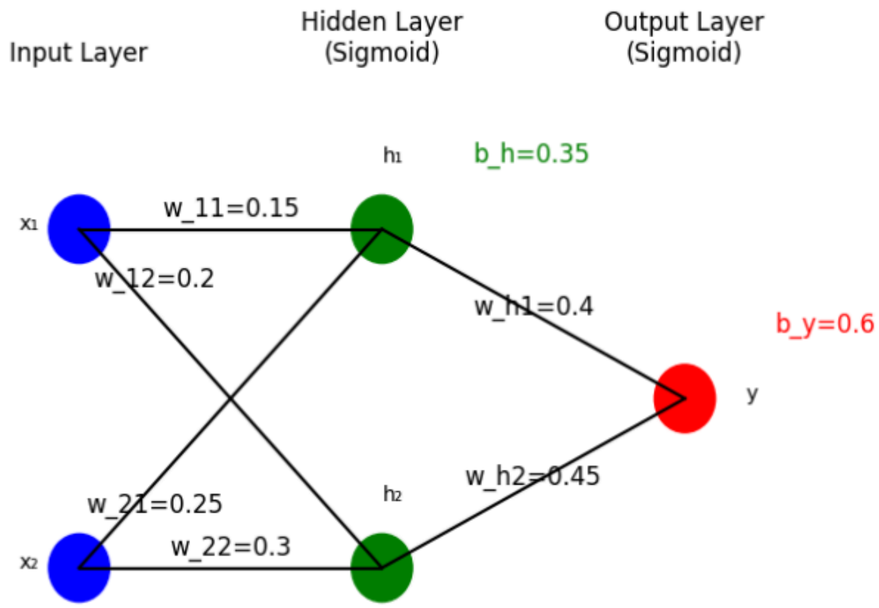
$w_{h2} \cdot \delta_y = 0.45 \cdot (-0.0084) \approx -0.00378$ Sigmoid derivative: $\sigma'(z_{h2}) = h_2(1 - h_2) = 0.6177(1 - 0.6177) \approx 0.2362$
$\delta_{h2} = -0.00378 \cdot 0.2362 \approx -0.00089$

Gradients for weights to hidden layer:

$$\frac{\partial E}{\partial w_{11}} = \delta_{h1} \cdot x_1 = -0.0008 \cdot 0.5 \approx -0.0004$$
$$\frac{\partial E}{\partial w_{21}} = \delta_{h1} \cdot x_2 = -0.0008 \cdot 0.1 \approx -0.00008$$
$$\frac{\partial E}{\partial w_{12}} = \delta_{h2} \cdot x_1 = -0.00089 \cdot 0.5 \approx -0.000445$$
$$\frac{\partial E}{\partial w_{22}} = \delta_{h2} \cdot x_2 = -0.00089 \cdot 0.1 \approx -0.000089$$

Gradient for hidden bias:

$$\frac{\partial E}{\partial b_h} = \delta_{h1} + \delta_{h2} = -0.0008 + (-0.00089) \approx -0.00169$$



2.2 Backpropagation Example

Step 3: Update Weights and Biases

Using gradient descent: $w = w - \eta \cdot \frac{\partial E}{\partial w}$. Output Layer:

$$w_{h1} = 0.4 - 0.5 \cdot (-0.00513) = 0.4 + 0.002565 \approx 0.4026$$

$$w_{h2} = 0.45 - 0.5 \cdot (-0.00519) = 0.45 + 0.002595 \approx 0.4526$$

$$b_y = 0.6 - 0.5 \cdot (-0.0084) = 0.6 + 0.0042 \approx 0.6042$$

Hidden Layer:

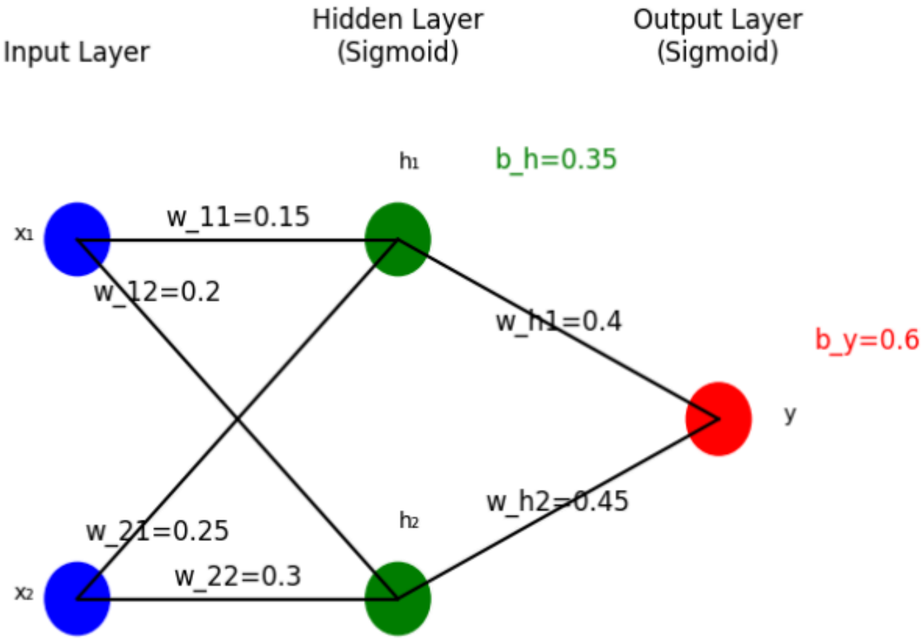
$$w_{11} = 0.15 - 0.5 \cdot (-0.0004) = 0.15 + 0.0002 \approx 0.1502$$

$$w_{21} = 0.25 - 0.5 \cdot (-0.00008) = 0.25 + 0.00004 \approx 0.25004$$

$$w_{12} = 0.2 - 0.5 \cdot (-0.000445) = 0.2 + 0.0002225 \approx 0.2002$$

$$w_{22} = 0.3 - 0.5 \cdot (-0.000089) = 0.3 + 0.0000445 \approx 0.30004$$

$$b_h = 0.35 - 0.5 \cdot (-0.00169) = 0.35 + 0.000845 \approx 0.3508$$



2.3 Backpropagation Applications

Backpropagation has many application in real life. In the scope of this section, we only separate them into 2 field: inside and outside of Machine Learning (ML).

Machine Learning (ML)

- Backpropagation is the backbone of training artificial neural networks (ANNs) and is widely used across ML domains:
  - **Image Recognition**: Optimizes weights in convolutional neural networks (CNNs) for tasks like object detection (e.g., identifying objects in images).
  - **Natural Language Processing (NLP)**: Adjusts weights in recurrent neural networks (RNNs) and transformers for tasks like machine translation and text generation.
  - **Speech Recognition**: Fine-tunes deep neural networks to convert audio to text by minimizing prediction errors.

Others

- **Control Systems**: Optimizes parameters in control algorithms for robotics or autonomous systems, adjusting to minimize error in system dynamics.

  $$E = \frac{1}{2}(y_{\text{desired}} - y_{\text{actual}})^2$$

- **Physics Simulations**: Used to fit models to experimental data, such as optimizing parameters in computational fluid dynamics.
- **Economics**: Adjusts weights in econometric models to predict economic trends or optimize resource allocation.
- **Signal Processing**: Fine-tunes filters in adaptive signal processing to minimize noise in communication systems.

COMPARISON & SUMMARY

2.4 Comparison & Summary — Compare to other Differentiation methods

Numerical Differentiation:

- Accuracy: Automatic differentiation (AD) computes exact derivatives using the chain rule, while numerical differentiation (ND) approximates them, introducing errors.
- Efficiency: AD is faster for complex or high-dimensional functions, requiring fewer evaluations than ND's multiple function calls.
- Use Case: AD needs specialized tools, ideal for machine learning; ND is simpler, better for quick approximations.

Symbolic Differentiation:

- Precision: AD gives numerical derivatives with machine precision; SD produces exact analytical expressions.
- Efficiency: AD is faster for numerical tasks, especially in high dimensions; SD is slower due to symbolic manipulation.
- Use Case: AD suits iterative computations (e.g., machine learning); SD is better for deriving reusable formulas.

Insight:

- Numerical Differentiation is simple but inaccurate and inefficient for large systems.
- Symbolic Differentiation is precise but impractical for dynamic, large-scale problems.
- Automatic Differentiation strikes a balance, offering exactness and efficiency, making it the preferred choice

2.4 Comparison & Summary

Comparison between Forward & Backward mode

Forward Mode AD computes derivatives alongside function evaluation, propagating tangents from inputs to outputs. It's efficient for functions with few inputs and many outputs ($n \ll m$), uses less memory for small input dimensions, and is simpler to implement. Best for real-time systems, sensitivity analysis, and computing Jacobians for small systems, but scales poorly with large input dimensions.

Backward Mode AD computes derivatives by backpropagating gradients from outputs to inputs. It's efficient for functions with many inputs and few outputs ($n \gg m$), but requires more memory to store intermediate activations. It's more complex to implement and suits neural network training, optimization with scalar loss, and gradient computation in deep learning, scaling well with large input dimensions.

Insight:

Forward Mode good for problems with a small number of inputs (e.g., 2–3 parameters) while Backward Mode are optimized for dealing with many-to-one condition like traning ML model.

VISUALIZATION, CODE AND IMPLEMENTATION

Autograd is a library for automatic differentiation, used to compute gradients via two methods: forward mode and reverse mode. Autograd is effective with functions that take arrays as input.

```
[ ]  !pip install autograd
```

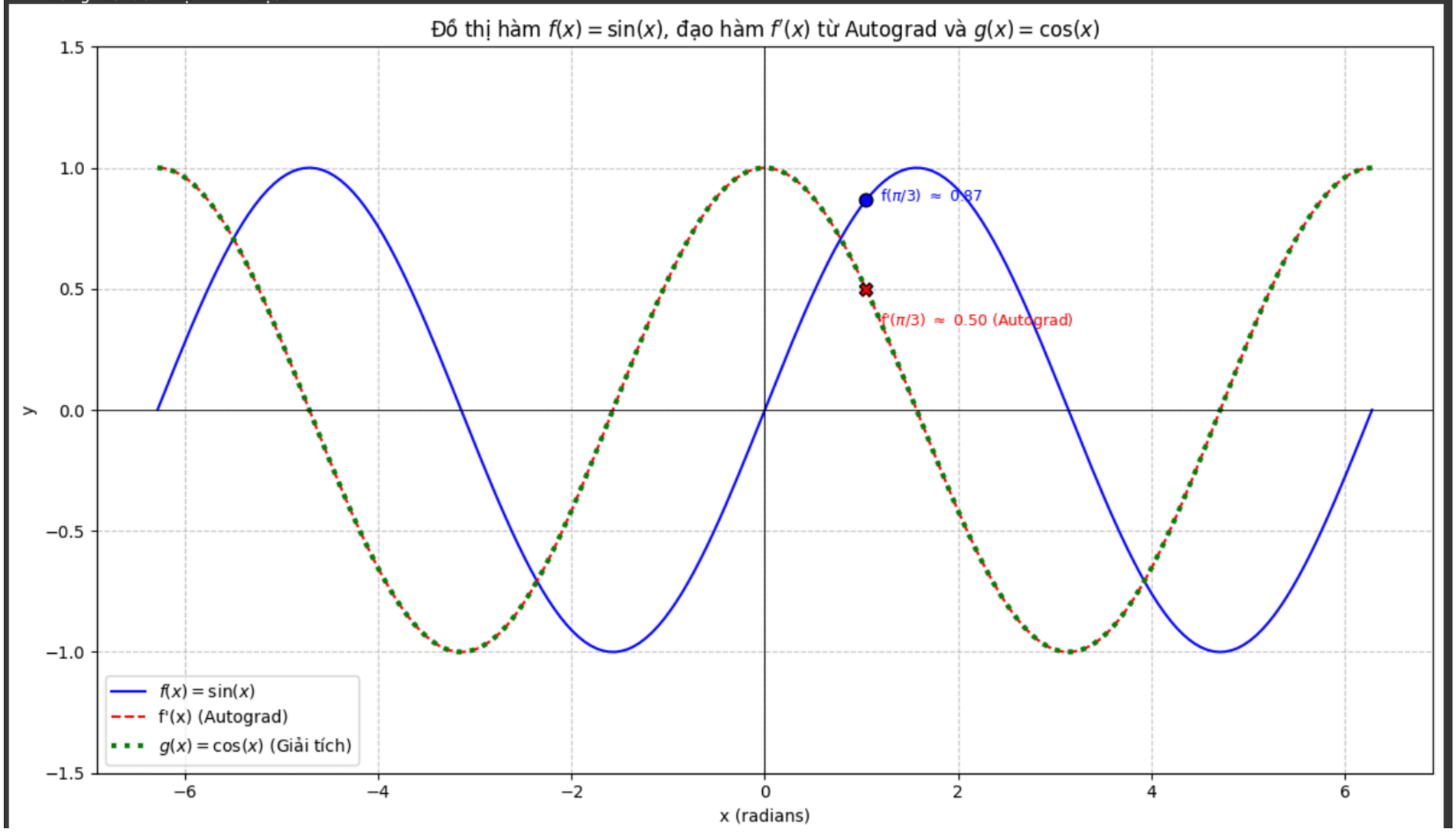
```
Requirement already satisfied: autograd in /usr/local/lib/python3.11/dist-packages (1.7.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from autograd) (2.0.2)
```

```
import autograd.numpy as np
from autograd import grad
import matplotlib.pyplot as plt
```

```
import autograd.numpy as np
from autograd import grad
import matplotlib.pyplot as plt
```

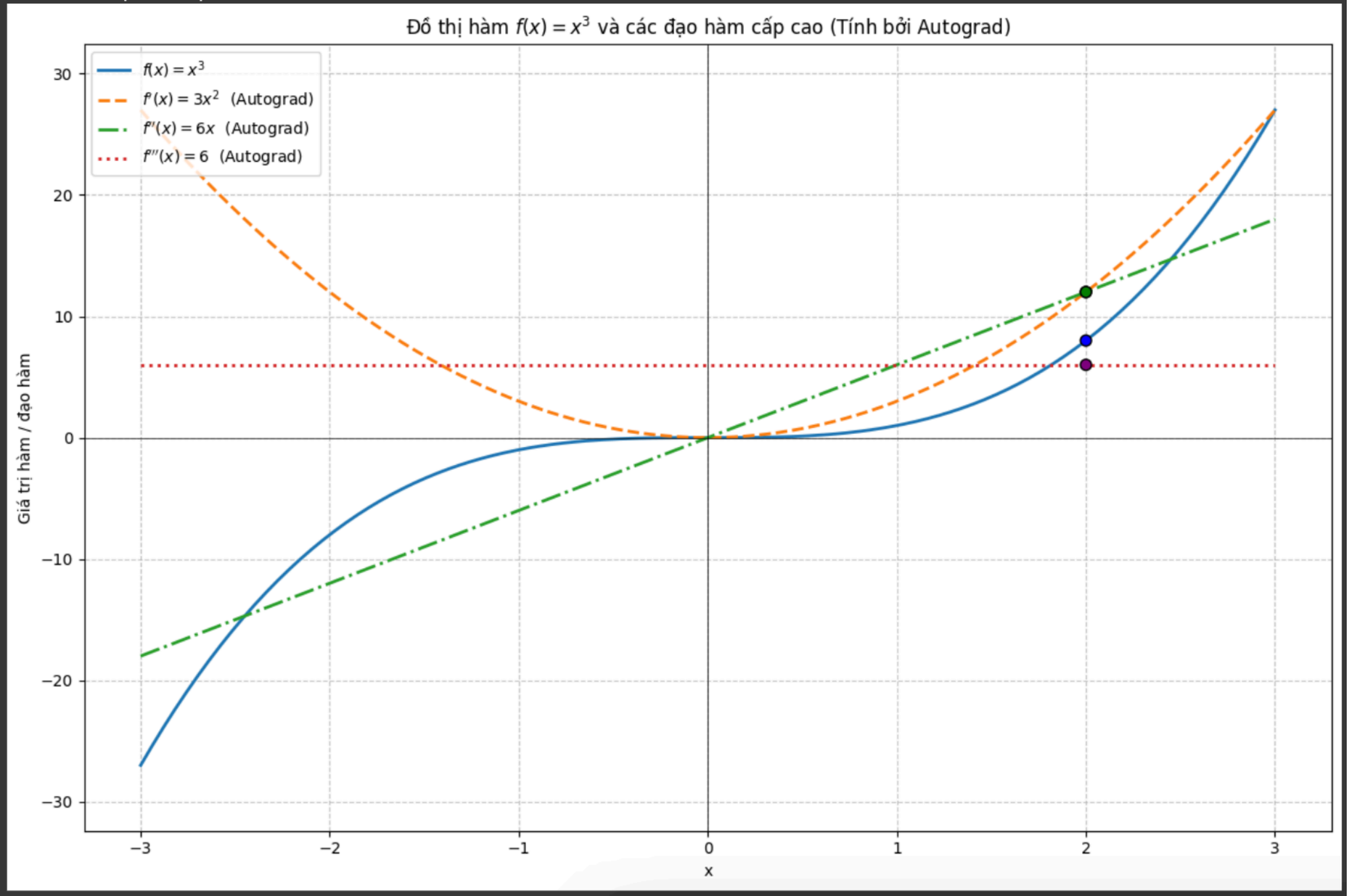
```python
import autograd.numpy as np
from autograd import grad

def my_function(x):
  return np.sin(x)

# hàm tính đạo hàm bằng grad
grad_my_function = grad(my_function)

# Tính đạo hàm tại x = pi/3
x_value = np.pi / 3
derivative_at_x = grad_my_function(x_value)
```

```
--- Tính toán tại một điểm x = pi/3 ---
Hàm số: f(x) = sin(x)
Giá trị x: 1.0472 (tức là pi/3)
Giá trị hàm f(pi/3): 0.8660
Đạo hàm f'(pi/3) (tính bằng autograd): 0.5000
Giá trị cos(pi/3) : 0.5000

--- Đang vẽ đồ thị minh họa ---
```



```python
def f_ex2(x):
    return x*x*x

# Tạo các hàm tính đạo hàm
grad_f_ex2 = elementwise_grad(f_ex2)
grad_f_order_2_ex2 = elementwise_grad(grad_f_ex2)
grad_f_order_3_ex2 = elementwise_grad(grad_f_order_2_ex2)

x_value_single_ex2 = 2.0 # Điểm cụ thể để đối chiếu
```

```
Đạo hàm cấp 1: f'(x) = 3x^2
   Giá trị tại x=2.0: f'(2.0) = 12.0
Đạo hàm cấp 2: f''(x) = 6x
   Giá trị tại x=2.0: f''(2.0) = 12.0
Đạo hàm cấp 3: f'''(x) = 6
   Giá trị tại x=2.0: f'''(2.0) = 6.0

--- Vẽ đồ thị minh họa ---
```



Xét hàm $$f(x_1, x_2) = x_1^2 \cdot x_2 + \sin(x_1)$$

Đạo hàm riêng từng phần theo x1 :

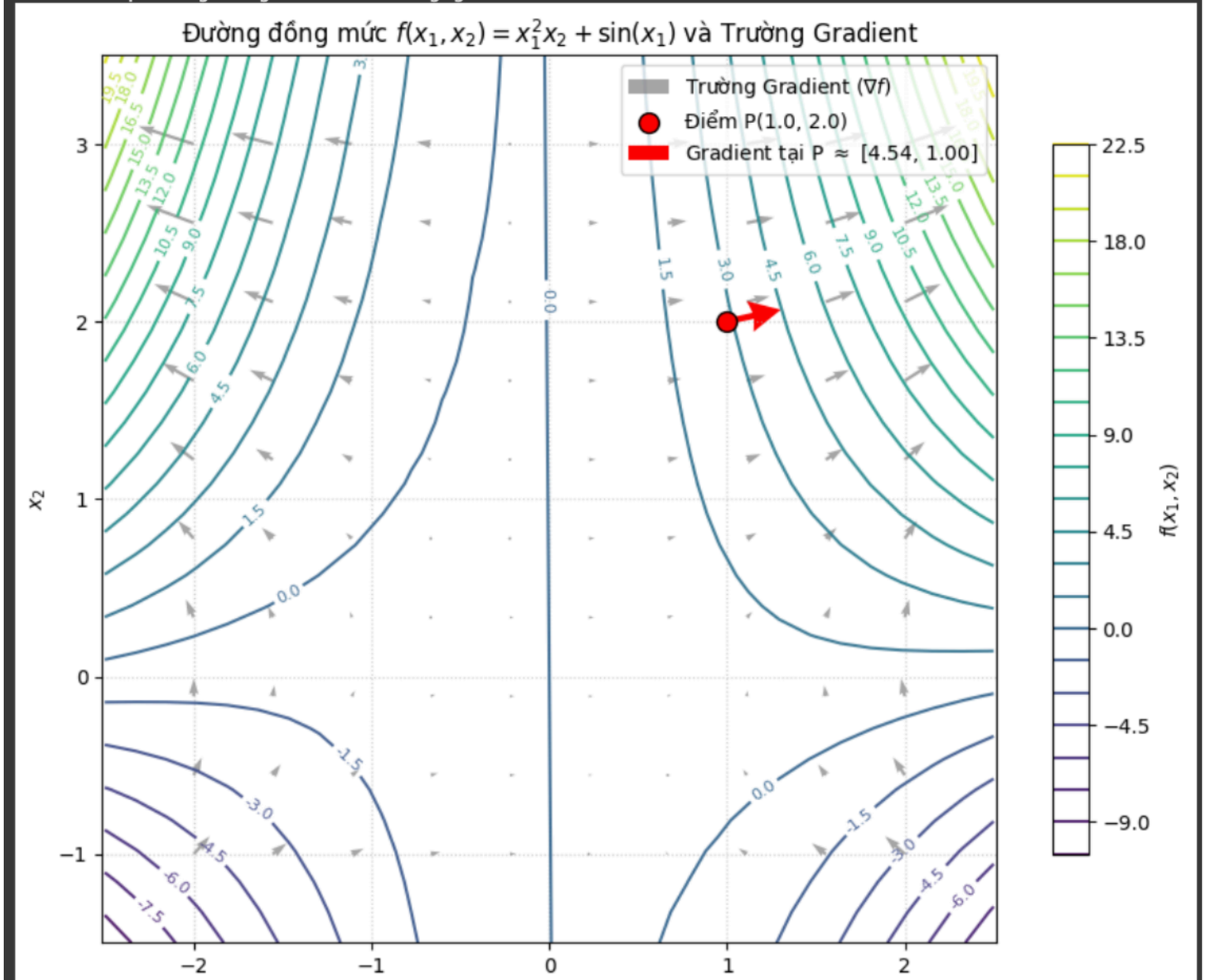
$$\frac{\partial f}{\partial x_1} = 2x_1x_2 + \cos(x_1)$$

Đạo hàm riêng từng phần theo x2 :

$$\frac{\partial f}{\partial x_2} = x_1^2$$

```python
# Hàm số gốc
def multi_var_func(params): # params là array [x1, x2]
  x1, x2 = params
  return x1**2 * x2 + np.sin(x1) # Trả về scalar

# Tính gradient của hàm
grad_multi_var_for_single_point = grad(multi_var_func)

# Điểm minh hoạ
params_values_center = np.array([1.0, 2.0]) # x1 = 1.0, x2 = 2.0
gradient_at_center = grad_multi_var_for_single_point(params_values_center)
function_value_at_center = multi_var_func(params_values_center)
```

```
--- Minh hoạ tại điểm (1.0, 2.0) ---
Hàm số: f(x1, x2) = x1^2 * x2 + sin(x1)
Giá trị hàm tại điểm: f(1.0, 2.0) = 2.8415
Gradient (df/dx1, df/dx2) tại điểm đó là: [4.5403, 1.0000]

--- Đồ thị đường đồng mức và trường gradient ---
```



- The gradient vector at a point P indicates the direction in which the value of the function f increases most rapidly
- The length of the gradient vector represents the rate of that speed increase.

CSE

EXERCISE 1 - 8

Exercise 1: Why is the second derivative much more expensive to compute than the first derivative?

The second derivative involves computing the **Hessian matrix**, which contains all possible second-order partial derivatives of a function. If a function f has n input variables:

- The **gradient** is a vector of size n.
- The **Hessian** is a matrix of size $n \times n$, i.e., it has n^2 elements.

This means the computational cost increases quadratically with the number of variables. Hence, calculating second-order derivatives is significantly more expensive. Hence, calculating second-order derivatives is significantly more expensive.

Exercise 2: After running the function for backpropagation, immediately run it again and see what happens. Investigate.

```python
import torch

x = torch.arange(4.0, requires_grad=True)
y = 2 * torch.dot(x, x)
y.backward()
print(x.grad)  # tensor([ 0.,  4.,  8., 12.])

# Second backward without clearing
y.backward()
print(x.grad)  # tensor([ 0.,  8., 16., 24.]) - accumulated

# Proper way: zero out gradient
x.grad.zero_()
y.backward()
print(x.grad)  # tensor([ 0.,  4.,  8., 12.])
```

Without retain_graph=True, `PyTorch` clears the computational graph after backward pass to save memory

The gradients accumulate when you run backward twice

The output shows [0, 8, 16, 24] because:

First backward: gradient is 4x → [0, 4, 8, 12]

Second backward: adds another 4x → [0+0, 4+4, 8+8, 12+12]

4 EXERCISES

EXERCISE 3

Exercise 3: In the control flow example where we calculate the derivative of d with respect to a, what would happen if we changed the variable a to a random vector or a matrix? At this point, the result of the calculation f(a) is no longer a scalar. What happens to the result? How do we analyze this?

```python
import torch

def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c

a = torch.randn(size=(2,3), requires_grad=True)
d = f(a)
d.backward(torch.ones_like(d))  # Need to provide gradient of output shape
print(a.grad)
```

- When output is non-scalar, must provide gradient tensor of same shape
- d.detach() works but isn't conceptually correct - should use ones/identity matrix
- The gradient scales with the number of doublings in the while loop

Exercise 4: Let f(x) = sin(x). Plot the graph of and of its derivative. Do not exploit the fact that but rather use automatic differentiation to get the result.

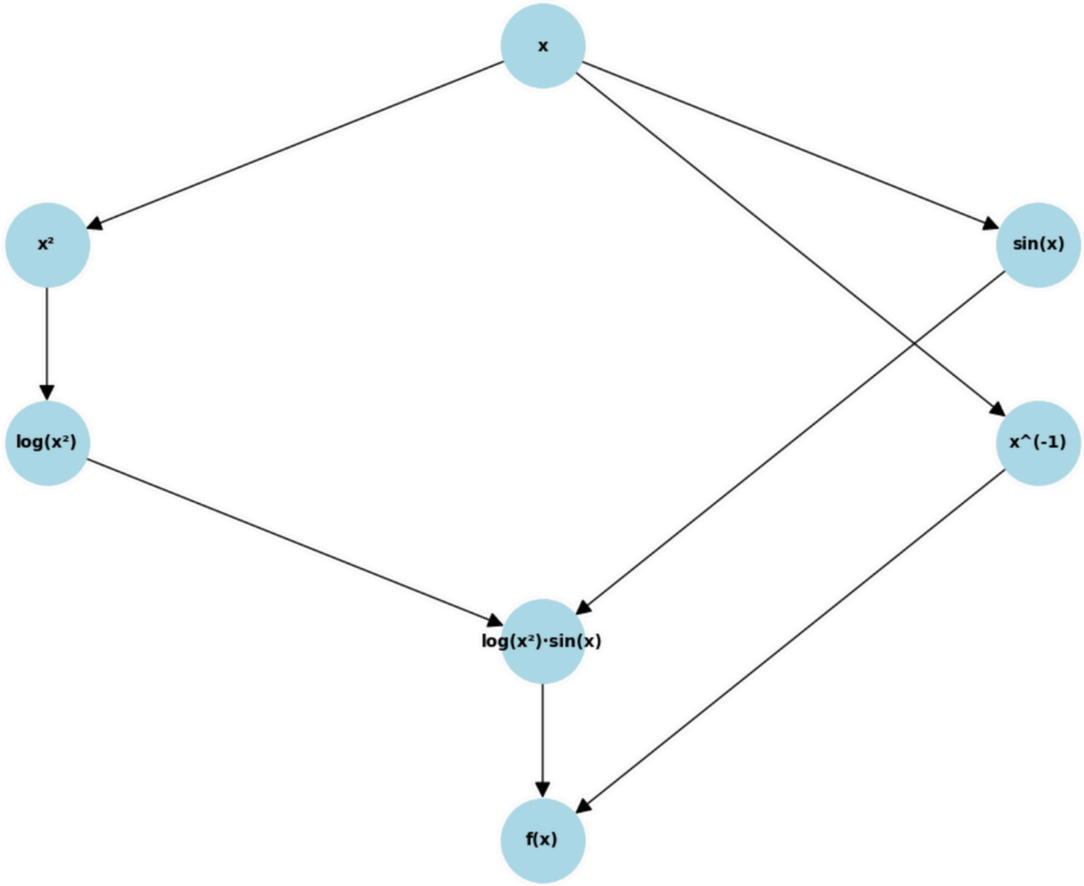
```
import torch

def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c

a = torch.randn(size=(2, 3), requires_grad=True)
d = f(a)
d.backward(torch.ones_like(d))  # Gradient must be provided
print(a.grad)
```

#result: tensor([[512., 512., 512.],[512., 512., 512.]])

Let $f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$. Write out a dependency graph tracing results from x to $f(x)$.

The computational steps are:

1. Input: x.
2. Compute x^2.
3. Compute $\log x^2$.
4. Compute $\sin x$.
5. Compute the product $(\log x^2) \cdot \sin x$.
6. Compute $x^{-1} = \frac{1}{x}$.
7. Compute $f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$.



Use the chain rule to compute the derivative $\frac{df}{dx}$ of the aforementioned function, placing each term on the dependency graph that you constructed previously.

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

Step 1: Apply the product rule to the first term

$$\frac{d}{dx}((\log x^2) \cdot \sin x) = (\log x^2) \cdot \frac{d}{dx}(\sin x) + \sin x \cdot \frac{d}{dx}(\log x^2)$$

Step 2: Calculate individual derivatives

- $\frac{d}{dx}(\sin x) = \cos x$
- $\frac{d}{dx}(\log x^2) = \frac{d}{dx}(\log x^2) = \frac{1}{x^2} \cdot \frac{d}{dx}(x^2) = \frac{1}{x^2} \cdot 2x = \frac{2}{x}$
- $\frac{d}{dx}(x^{-1}) = \frac{d}{dx}(\frac{1}{x}) = -\frac{1}{x^2}$

Step 3: Substitute back into the product rule

$$\frac{d}{dx}((\log x^2) \cdot \sin x) = (\log x^2) \cdot \cos x + \sin x \cdot \frac{2}{x}$$

Step 4: Add the derivative of the second term

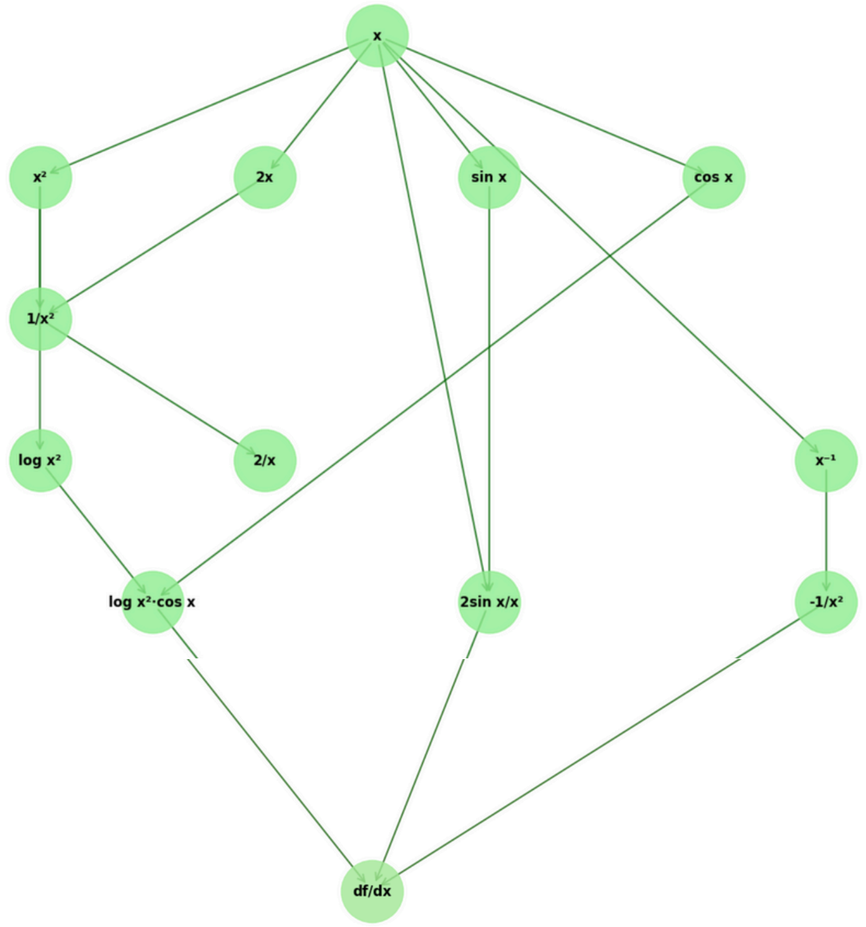
$$\frac{df}{dx} = (\log x^2) \cdot \cos x + \frac{2 \sin x}{x} - \frac{1}{x^2}$$

Final result: $\frac{df}{dx} = (\log x^2) \cdot \cos x + \frac{2 \sin x}{x} - \frac{1}{x^2}$

Use the chain rule to compute the derivative $\frac{df}{dx}$ of the aforementioned function, placing each term on the dependency graph that you constructed previously.

Derivative Nodes:

- $2x$: Derivative of x^2 with respect to x
- $\frac{1}{x^2}$: Derivative component for $\log x^2$
- $\frac{2}{x}$: Derivative of $\log x^2$
- $\cos x$: Derivative of $\sin x$
- $(\log x^2) \cdot \cos x$: First term of product rule
- $\frac{2 \sin x}{x}$: Second term of product rule
- $-\frac{1}{x^2}$: Derivative of x^{-1}
- $\frac{df}{dx}$: Final derivative $(\log x^2) \cdot \cos x + \frac{2 \sin x}{x} - \frac{1}{x^2}$



Given the graph and the intermediate derivative results, you have a number of options when computing the gradient. Evaluate the result once starting from x to f and once from f tracing back to x. The path from x to f is commonly known as *forward differentiation*, whereas the path from f to x is known as backward differentiation.

The function is: $f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$

We compute the derivative $f'(x)$ using:

1. **Forward Differentiation**: Propagating derivatives from x to $f(x)$.
2. **Backward Differentiation**: Propagating sensitivities from $f(x)$ to x.

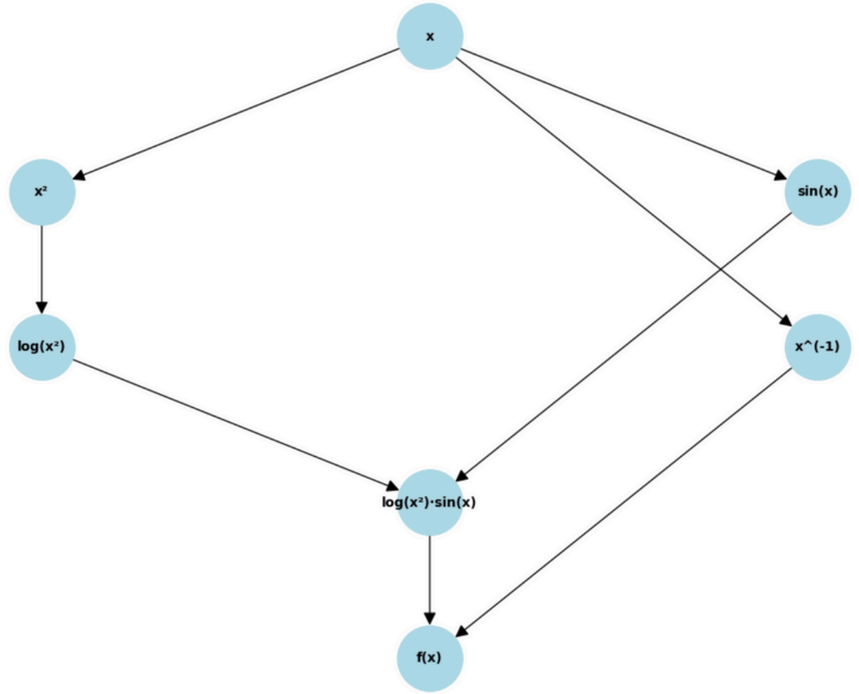
Both methods follow the dependency graph:

- Nodes: x, x^2, $\log x^2$, $\sin x$, $\text{product} = (\log x^2) \cdot \sin x$, x^{-1}, $f(x)$.
- Edges define dependencies as shown in the previous section.

Given the graph and the intermediate derivative results, you have a number of options when computing the gradient. Evaluate the result once starting from x to f and once from f tracing back to x. The path from x to f is commonly known as *forward differentiation*, whereas the path from f to x is known as backward differentiation.

Forward Differentiation

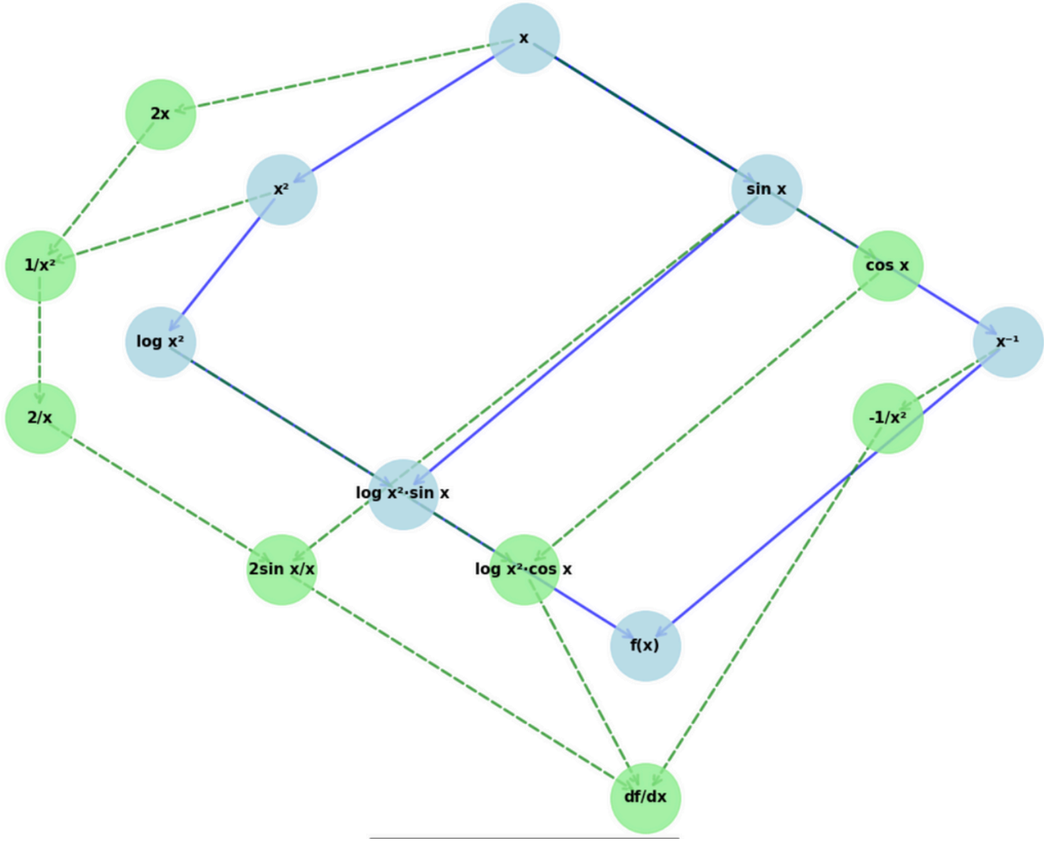
1. Start with $\frac{dx}{dx} = 1$
2. Compute $\frac{d(x^2)}{dx} = 2x$
3. Compute $\frac{d(\log x^2)}{dx} = \frac{1}{x^2} \cdot \frac{d(x^2)}{dx} = \frac{1}{x^2} \cdot 2x = \frac{2}{x}$
4. Compute $\frac{d(\sin x)}{dx} = \cos x$
5. Compute $\frac{d((\log x^2) \cdot \sin x)}{dx}$ using the product rule: $\frac{d(\log x^2)}{dx} \cdot \sin x + (\log x^2) \cdot \frac{d(\sin x)}{dx}$
   $= \frac{2}{x} \cdot \sin x + (\log x^2) \cdot \cos x = \frac{2 \sin x}{x} + (\log x^2) \cdot \cos x$
6. Compute $\frac{d(x^{-1})}{dx} = \frac{d(\frac{1}{x})}{dx} = -\frac{1}{x^2}$
7. Compute $\frac{df}{dx}$ by combining results from steps 5 and 6: $\frac{df}{dx} = \frac{d((\log x^2) \cdot \sin x)}{dx} + \frac{d(x^{-1})}{dx}$
   $= \frac{2 \sin x}{x} + (\log x^2) \cdot \cos x - \frac{1}{x^2}$



Given the graph and the intermediate derivative results, you have a number of options when computing the gradient. Evaluate the result once starting from x to f and once from f tracing back to x. The path from x to f is commonly known as *forward differentiation*, whereas the path from f to x is known as backward differentiation.

Forward Differentiation

1. Start with $\frac{dx}{dx} = 1$
2. Compute $\frac{d(x^2)}{dx} = 2x$
3. Compute $\frac{d(\log x^2)}{dx} = \frac{1}{x^2} \cdot \frac{d(x^2)}{dx} = \frac{1}{x^2} \cdot 2x = \frac{2}{x}$
4. Compute $\frac{d(\sin x)}{dx} = \cos x$
5. Compute $\frac{d((\log x^2) \cdot \sin x)}{dx}$ using the product rule: $\frac{d(\log x^2)}{dx} \cdot \sin x + (\log x^2) \cdot \frac{d(\sin x)}{dx}$
   $= \frac{2}{x} \cdot \sin x + (\log x^2) \cdot \cos x = \frac{2 \sin x}{x} + (\log x^2) \cdot \cos x$
6. Compute $\frac{d(x^{-1})}{dx} = \frac{d(\frac{1}{x})}{dx} = -\frac{1}{x^2}$
7. Compute $\frac{df}{dx}$ by combining results from steps 5 and 6: $\frac{df}{dx} = \frac{d((\log x^2) \cdot \sin x)}{dx} + \frac{d(x^{-1})}{dx}$
   $= \frac{2 \sin x}{x} + (\log x^2) \cdot \cos x - \frac{1}{x^2}$



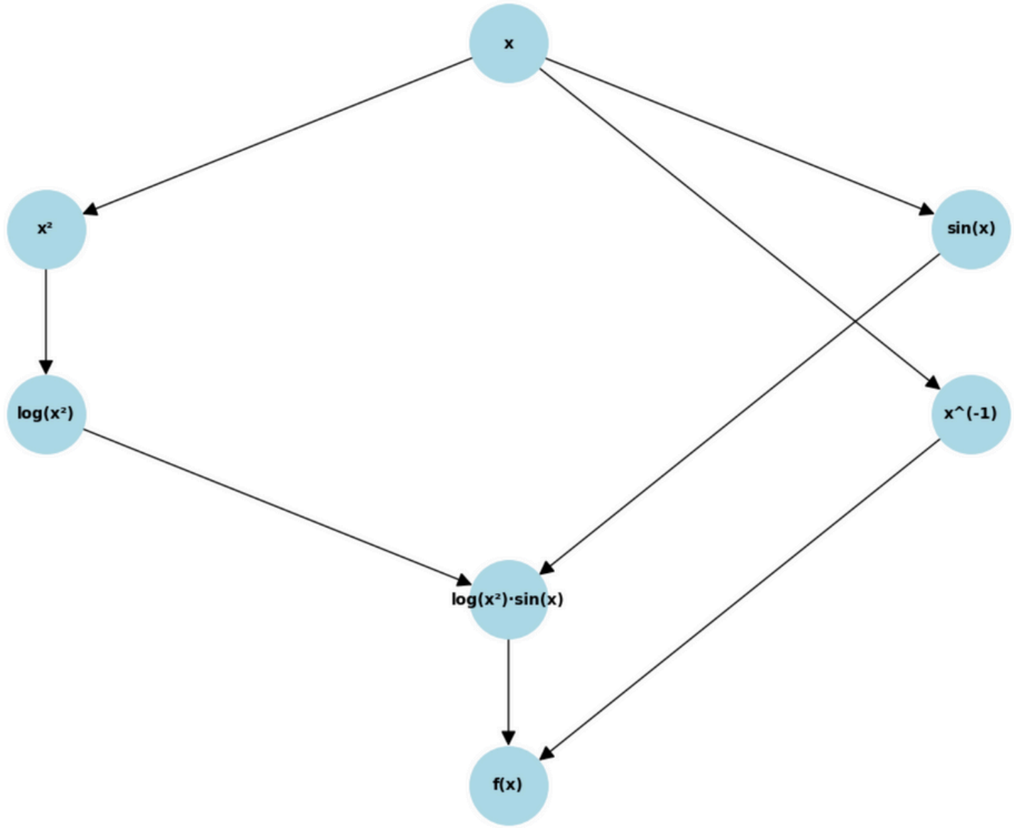
Backward Differentiation

1. Start with $\frac{\partial f}{\partial f} = 1$
2. For the addition node ($f = \text{product} + x^{-1}$): $\frac{\partial f}{\partial \text{product}} = 1$ $\frac{\partial f}{\partial x^{-1}} = 1$
3. For the product node ($\text{product} = (\log x^2) \cdot \sin x$): $\frac{\partial \text{product}}{\partial (\log x^2)} = \sin x$ $\frac{\partial \text{product}}{\partial (\sin x)} = \log x^2$
4. For the $\sin x$ node: $\frac{\partial (\sin x)}{\partial x} = \cos x$
5. For the $\log x^2$ node: $\frac{\partial (\log x^2)}{\partial (x^2)} = \frac{1}{x^2}$
6. For the x^2 node: $\frac{\partial (x^2)}{\partial x} = 2x$
7. For the x^{-1} node: $\frac{\partial (x^{-1})}{\partial x} = -\frac{1}{x^2}$
8. Apply the chain rule to compute $\frac{df}{dx}$:

   Through the first path (via product and log): $\frac{\partial f}{\partial \text{product}} \cdot \frac{\partial \text{product}}{\partial (\log x^2)} \cdot \frac{\partial (\log x^2)}{\partial (x^2)} \cdot \frac{\partial (x^2)}{\partial x}$
   $= 1 \cdot \sin x \cdot \frac{1}{x^2} \cdot 2x = \sin x \cdot \frac{2x}{x^2} = \frac{2 \sin x}{x}$

   Through the second path (via product and sin): $\frac{\partial f}{\partial \text{product}} \cdot \frac{\partial \text{product}}{\partial (\sin x)} \cdot \frac{\partial (\sin x)}{\partial x} = 1 \cdot (\log x^2) \cdot \cos x$
   $= (\log x^2) \cdot \cos x$

   Through the third path (via x^{-1}): $\frac{\partial f}{\partial x^{-1}} \cdot \frac{\partial (x^{-1})}{\partial x} = 1 \cdot (-\frac{1}{x^2}) = -\frac{1}{x^2}$
9. Sum all partial derivatives to get the total derivative: $\frac{df}{dx} = \frac{2 \sin x}{x} + (\log x^2) \cdot \cos x - \frac{1}{x^2}$



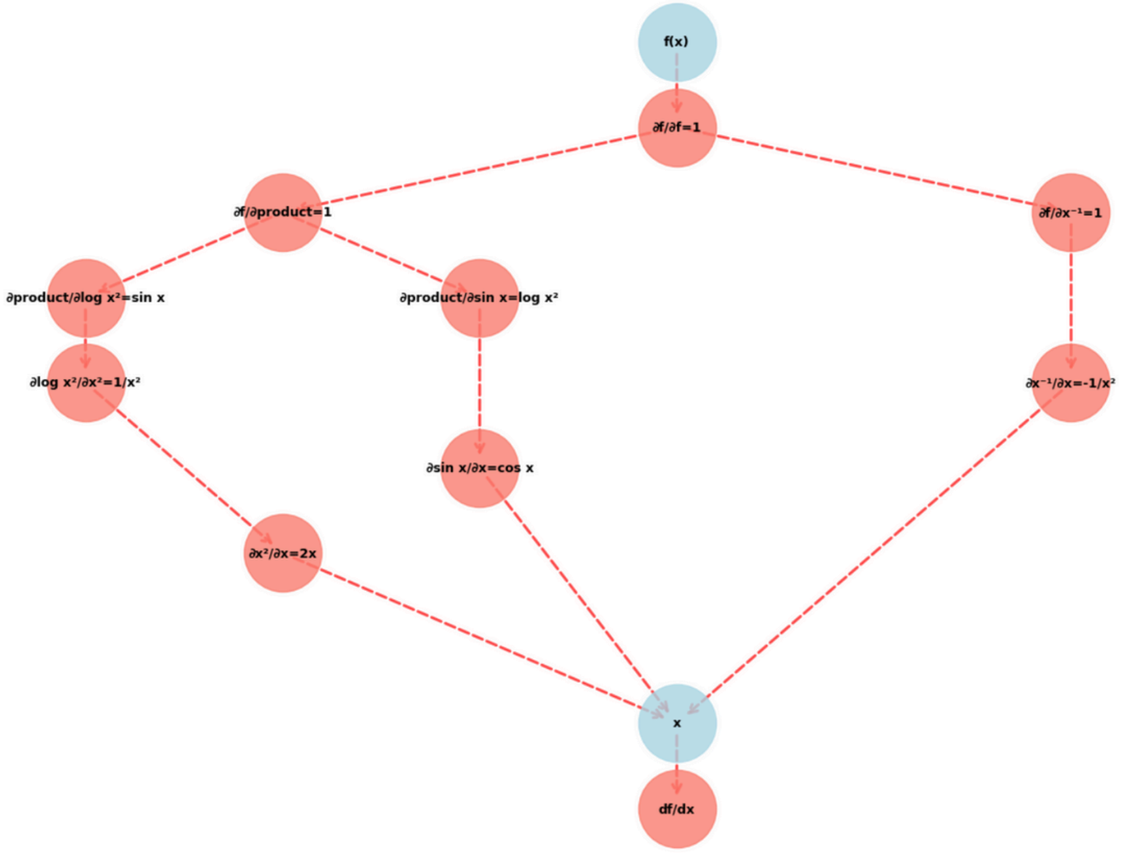
Backward Differentiation

1. Start with $\frac{\partial f}{\partial f} = 1$
2. For the addition node ($f = \text{product} + x^{-1}$): $\frac{\partial f}{\partial \text{product}} = 1$ $\frac{\partial f}{\partial x^{-1}} = 1$
3. For the product node ($\text{product} = (\log x^2) \cdot \sin x$): $\frac{\partial \text{product}}{\partial (\log x^2)} = \sin x$ $\frac{\partial \text{product}}{\partial (\sin x)} = \log x^2$
4. For the $\sin x$ node: $\frac{\partial (\sin x)}{\partial x} = \cos x$
5. For the $\log x^2$ node: $\frac{\partial (\log x^2)}{\partial (x^2)} = \frac{1}{x^2}$
6. For the x^2 node: $\frac{\partial (x^2)}{\partial x} = 2x$
7. For the x^{-1} node: $\frac{\partial (x^{-1})}{\partial x} = -\frac{1}{x^2}$
8. Apply the chain rule to compute $\frac{df}{dx}$:

   Through the first path (via product and log): $\frac{\partial f}{\partial \text{product}} \cdot \frac{\partial \text{product}}{\partial (\log x^2)} \cdot \frac{\partial (\log x^2)}{\partial (x^2)} \cdot \frac{\partial (x^2)}{\partial x}$
   $= 1 \cdot \sin x \cdot \frac{1}{x^2} \cdot 2x = \sin x \cdot \frac{2x}{x^2} = \frac{2 \sin x}{x}$

   Through the second path (via product and sin): $\frac{\partial f}{\partial \text{product}} \cdot \frac{\partial \text{product}}{\partial (\sin x)} \cdot \frac{\partial (\sin x)}{\partial x} = 1 \cdot (\log x^2) \cdot \cos x$
   $= (\log x^2) \cdot \cos x$

   Through the third path (via x^{-1}): $\frac{\partial f}{\partial x^{-1}} \cdot \frac{\partial (x^{-1})}{\partial x} = 1 \cdot (-\frac{1}{x^2}) = -\frac{1}{x^2}$
9. Sum all partial derivatives to get the total derivative: $\frac{df}{dx} = \frac{2 \sin x}{x} + (\log x^2) \cdot \cos x - \frac{1}{x^2}$



When might you want to use forward, and when backward, differentiation? Hint: consider the amount of intermediate data needed, the ability to parallelize steps, and the size of matrices and vectors involved.

Amount of Intermediate Data Needed

- **Forward**: Prefer when n (number of inputs) is small, as memory scales with n. Ideal for functions with few inputs and many outputs ($n \ll m$).
- **Backward**: Prefer when m (number of outputs) is small, as memory scales with m. Ideal for functions with many inputs and few outputs ($m \ll n$), common in machine learning (e.g., scalar loss functions).

Ability to Parallelize Steps

- **Forward**: Prefer when you have many inputs (n) and parallel hardware (e.g., GPUs) to compute derivatives for each input simultaneously. Best for $n \gg m$.
- **Backward**: Prefer when you have many outputs (m) and can parallelize across outputs, or when m is small (e.g., $m = 1$), and parallelization within the backward pass (e.g., across nodes) is feasible. Common in neural networks where $m = 1$.

Size of Matrices and Vectors Involved

- **Forward**: Use when $n \ll m$, as it requires fewer passes (proportional to n). For example, in simulations with few parameters (n) and many outputs (m), like computational fluid dynamics with multiple observables.
- **Backward**: Use when $m \ll n$, as it requires fewer passes (proportional to m). Common in machine learning, where the loss is a scalar ($m = 1$) and there are many parameters (n), like neural network weights.

CẢM ƠN THẦY VÀ CÁC BẠN ĐÃ LẮNG NGHE