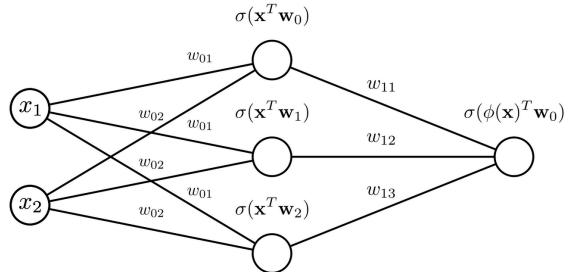


# Automatic differentiation: Motivation

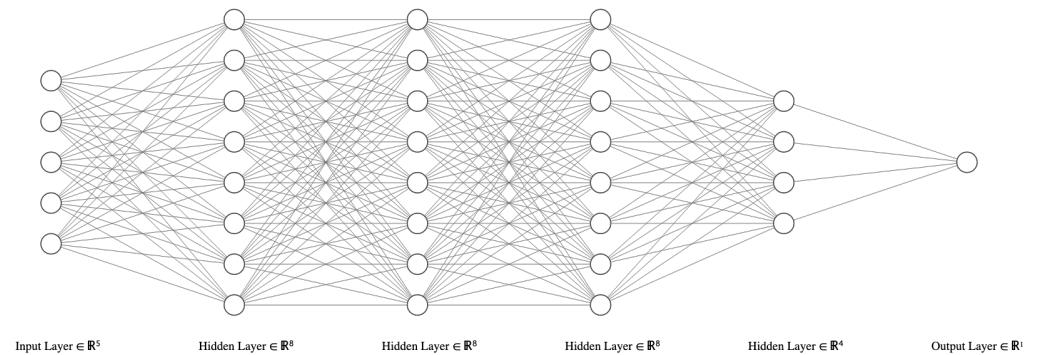
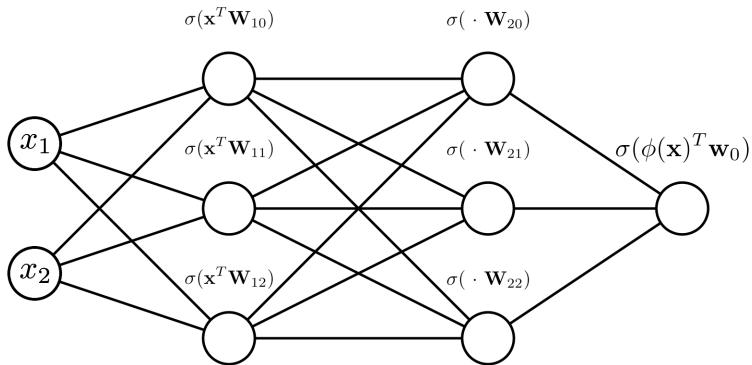


$$\mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = - \sum_{i=1}^N \log \sigma((2y_i - 1)\phi(\mathbf{x}_i)^T \mathbf{w})$$

$$= - \sum_{i=1}^N \log \sigma((2y_i - 1)\sigma(w_{01} \cdot \sigma(x_1 W_{11} + x_2 W_{12}) + w_{02} \cdot \sigma(x_1 W_{21} + x_2 W_{22}) + w_{03} \cdot \sigma(x_1 W_{31} + x_2 W_{32})))$$

# Automatic differentiation: Motivation

$$\text{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = - \sum_{i=1}^N \log \sigma((2y_i - 1)\phi(\mathbf{x}_i)^T \mathbf{w})$$



# Derivatives revisited

Given a function  $f: \mathbb{R} \rightarrow \mathbb{R}$ , a derivative is the *instantaneous rate of change*

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

## Derivatives revisited

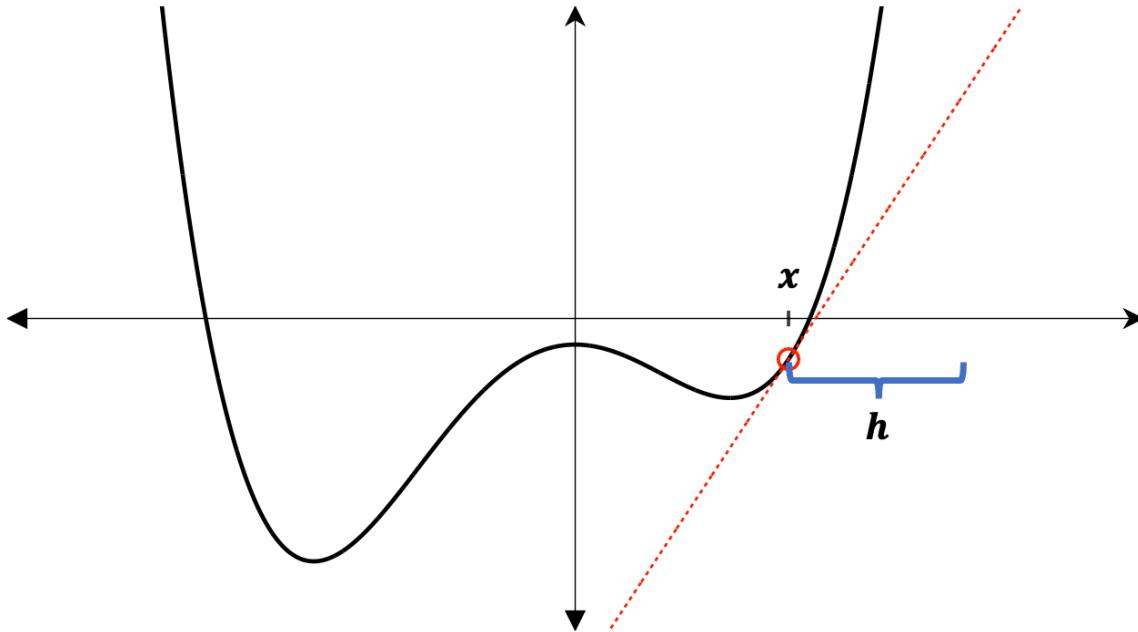
Given a function  $f: \mathbb{R} \rightarrow \mathbb{R}$ , a derivative is the *instantaneous rate of change*

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

Equivalently, it's the slope of the *optimal linear approximation* of  $f$  near  $x$ .

# Derivatives revisited

## *Optimal linear approximation*



**Consider the question:**

“If I perturb  $x$  by  $h$ , how does the output change?”

# The chain rule

**Where does this appear?**

Consider the chain rule

$$\frac{\partial}{\partial x} f(g(x)) \cdot 1 = f'(g(x)) \cdot g'(x) \cdot 1$$


If I perturb  $x$  by 1, by how  
much does  $f(g(x))$  change?

(under a first-order approx.)

# The chain rule

**Where does this appear?**

Consider the chain rule

$$\frac{\partial}{\partial x} f(g(x)) \cdot 1 = f'(g(x)) \cdot \underbrace{g'(x) \cdot 1}_{}$$

If I perturb  $x$  by 1, by how much does  $g(x)$  change?

# The chain rule

## Where does this appear?

Consider the chain rule

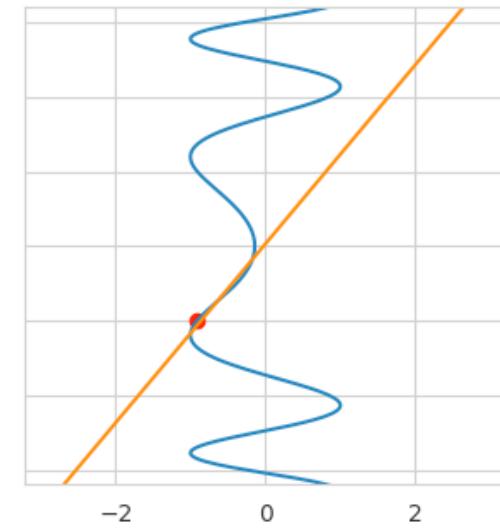
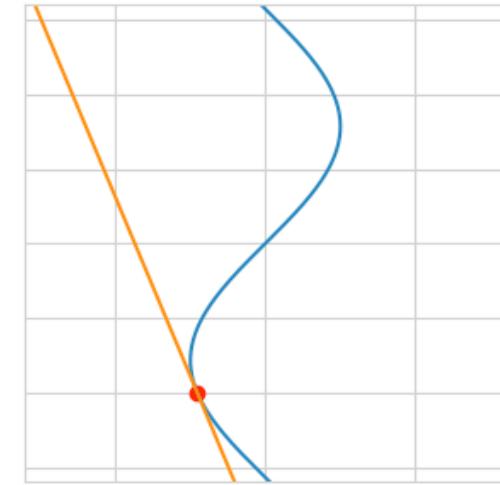
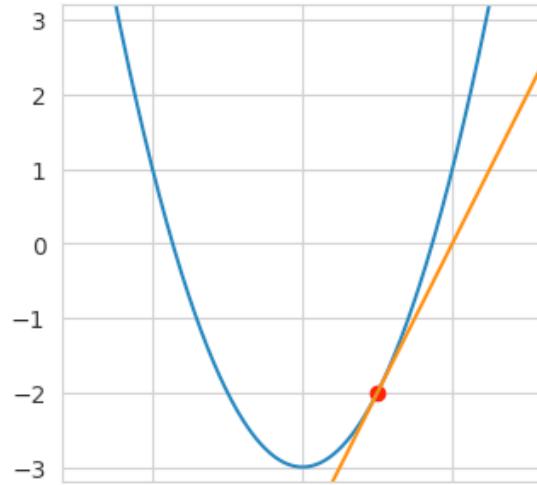
$$\frac{\partial}{\partial x} f(g(x)) \cdot 1 = \underbrace{f'(g(x))}_{\text{ }} \cdot h$$

If I perturb  $g(x)$  by  $h$ , by how much does  $f(g(x))$  change?

(to the first order)

To find the derivative of a complicated function, we iteratively map input perturbations to output perturbations for each sub-function

# Chain rule illustrated



## Useful notation

$$\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$$

$$b = g(x)$$

$$a = f(b)$$

$$\frac{da}{dx} = \frac{da}{db} \frac{db}{dx}$$

Corresponds to code

```
b = x ** 2  
a = log(b)
```

## Useful notation

```
b = x ** 2  
a = log(b)
```

$$b = g(x)$$

$$a = f(b)$$

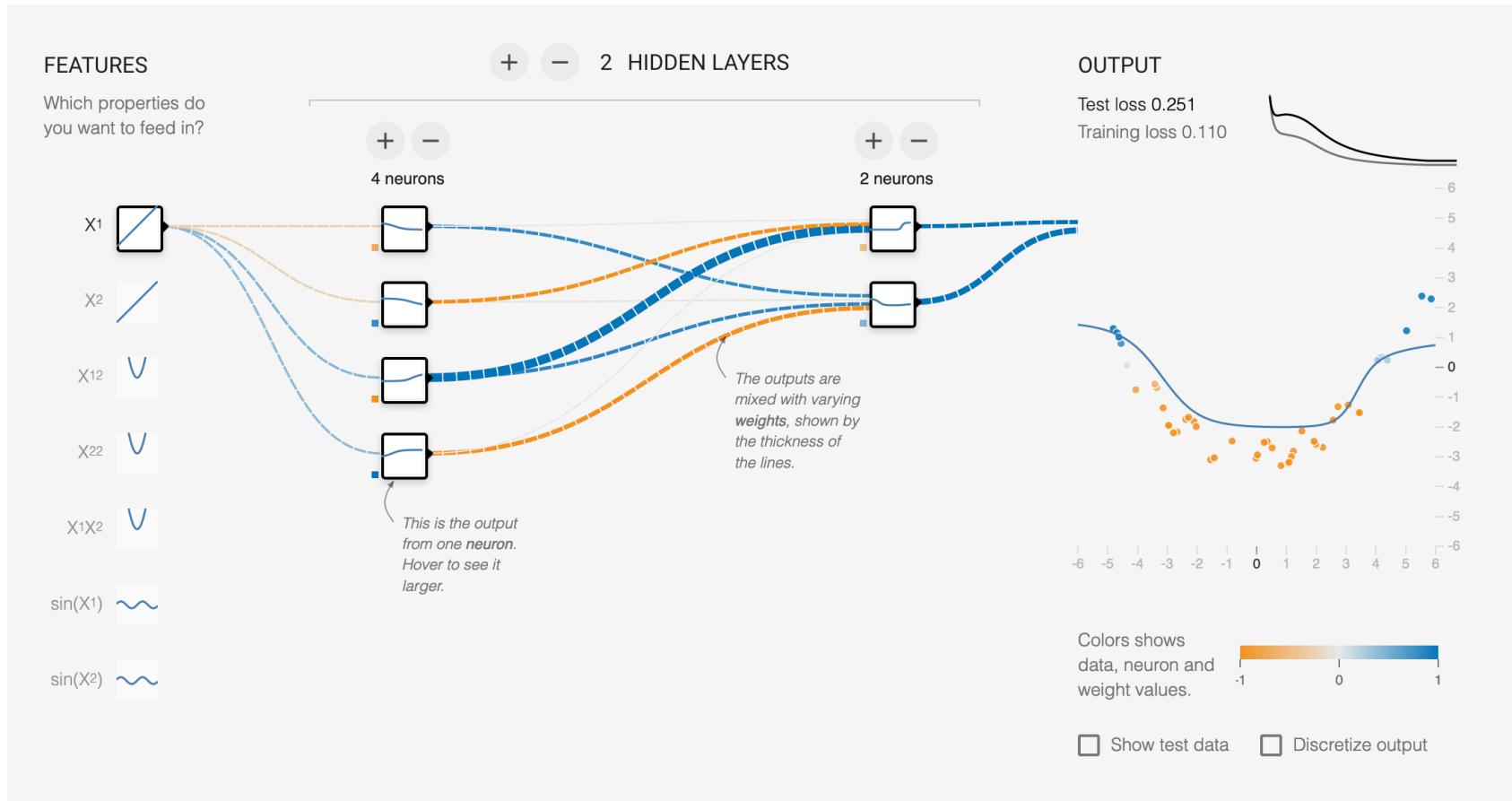
$$a = \log(b), \quad b = x^2$$

$$\frac{da}{db} = \quad \quad \quad \frac{db}{dx} =$$

$$\frac{da}{dx} = \frac{da}{db} \frac{db}{dx}$$

$$\frac{da}{dx} =$$

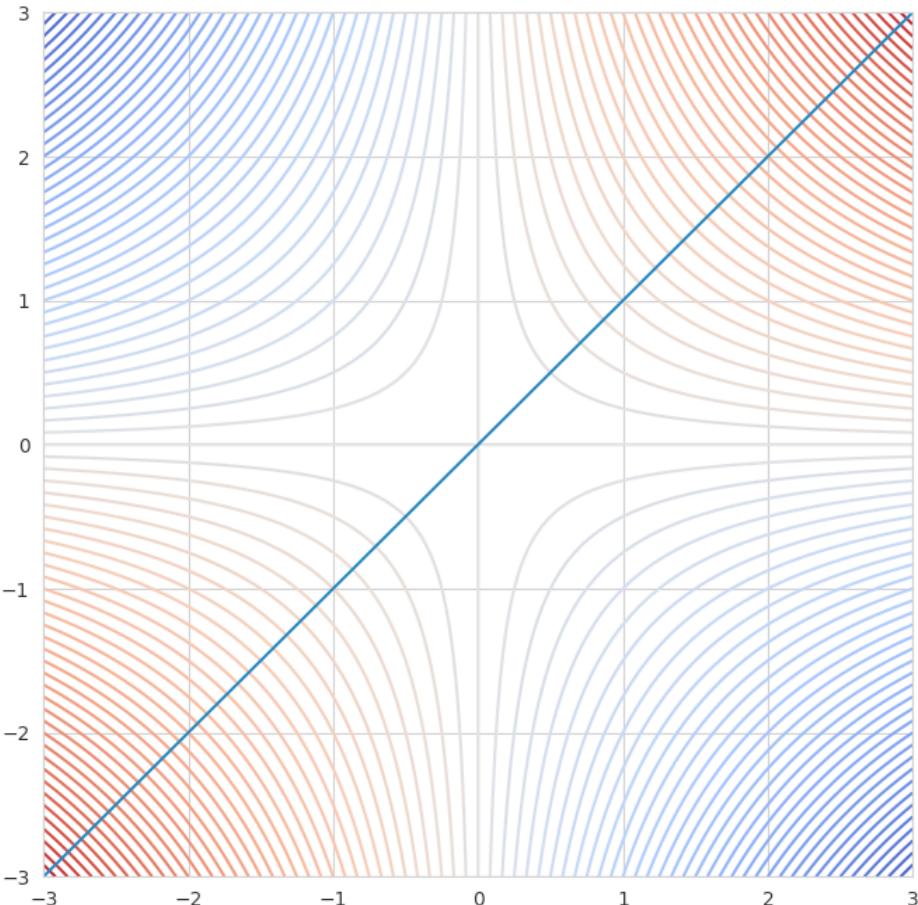
# Neural networks



## Total derivatives

$$\frac{df(y(x), z(x))}{dx} = \frac{df}{dy} \frac{dy}{dx} + \frac{df}{dz} \frac{dz}{dx}$$

## Total derivatives



$$f(x, y) = xy$$

$$\frac{df}{dx} = y \qquad \qquad \frac{df}{dy} = x$$

$$\frac{d}{dx} f(x, x) =$$

## Total derivatives

$$\frac{df(y(x), z(x))}{dx} = \frac{df}{dy} \frac{dy}{dx} + \frac{df}{dz} \frac{dz}{dx}$$

## Total derivatives

$$\frac{df(y(x), x)}{dx} = \frac{df}{dy} \frac{dy}{dx} + \frac{df}{dx}$$

## Partial derivatives

$$\frac{df(y(x), x)}{dx} = \frac{\partial f}{\partial y} \frac{\partial y}{\partial x} + \frac{\partial f}{\partial x}$$

## Composing the chain rule

$$L = -\log \sigma(wx^2)$$

$$a = x^2$$

$$b = wa$$

$$c = \sigma(b)$$

$$g = \log c$$

$$L = -g$$

## Composing the chain rule

$$L = -\log \sigma(wx^2)$$

$$a = x^2$$

$$b = wa$$

$$c = \sigma(b)$$

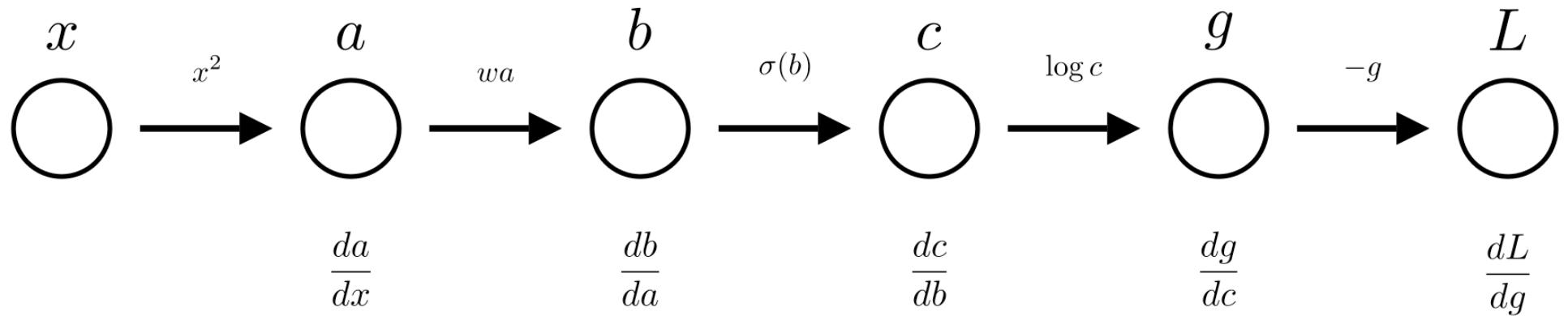
$$g = \log c$$

$$L = -g$$

$$\frac{dL}{dx} = \frac{dL}{dg} \frac{dg}{dc} \frac{dc}{db} \frac{db}{da} \frac{da}{dx}$$

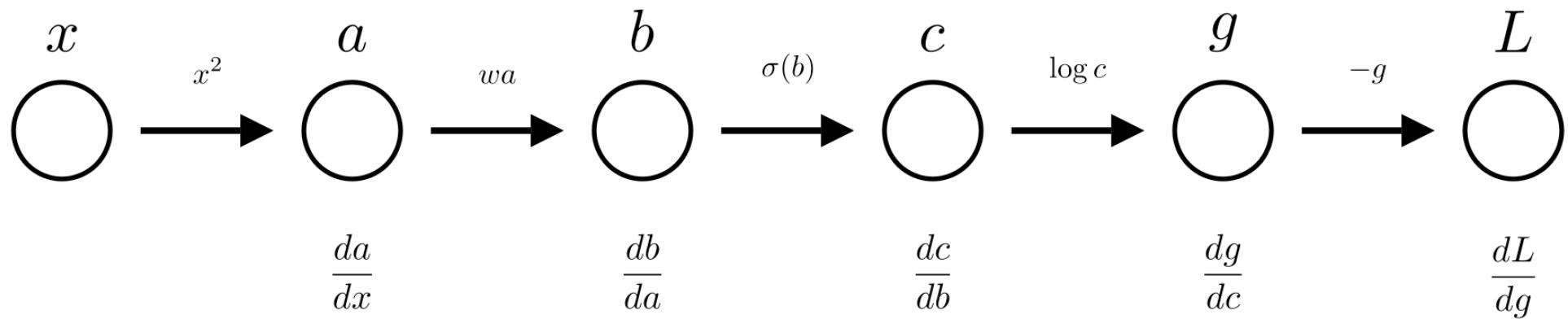
# Computational graphs

$$L = -\log \sigma(wx^2)$$



# Forward-mode automatic differentiation

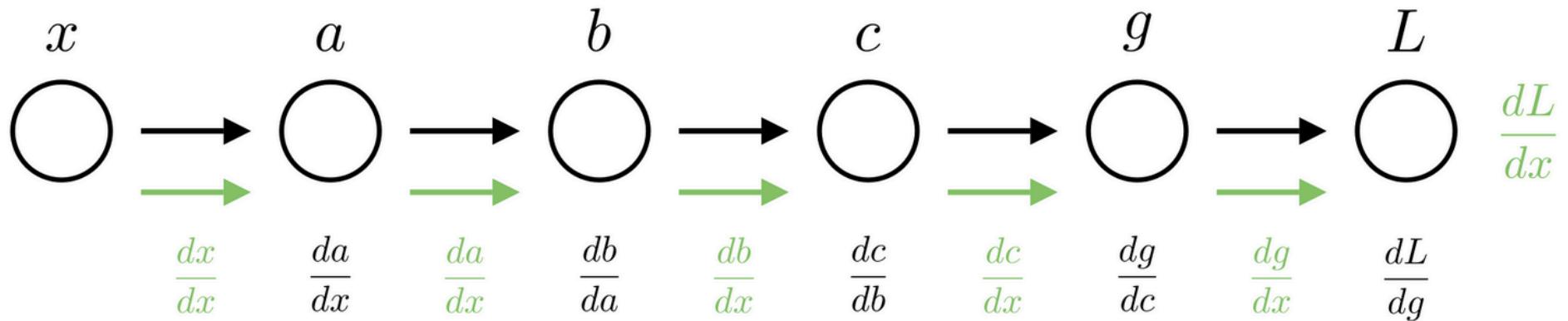
$$L = -\log \sigma(wx^2)$$



# Forward-mode automatic differentiation

$$L = -\log \sigma(wx^2)$$

## Forward mode



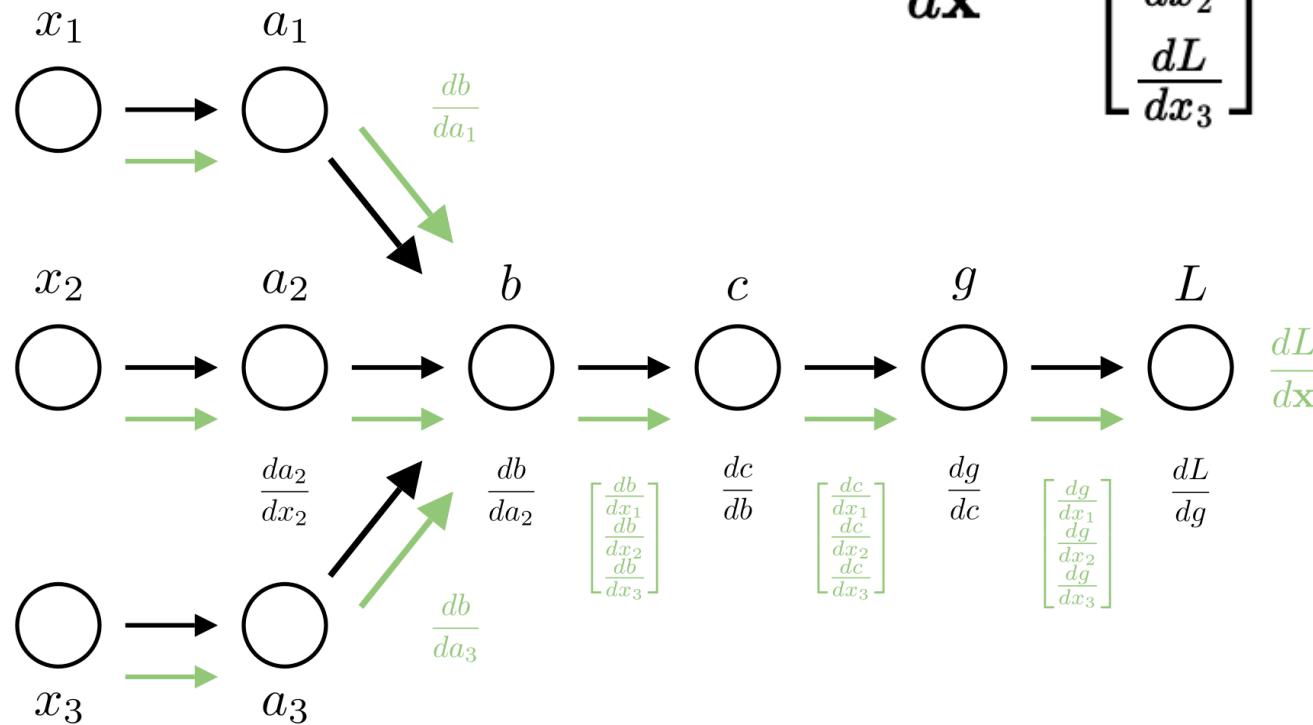
## Automatic differentiation with multiple inputs

$$-\log \sigma(w_1x_1 + w_2x_2 + w_3x_3) \quad \frac{dL}{d\mathbf{x}} = \begin{bmatrix} \frac{dL}{dx_1} \\ \frac{dL}{dx_2} \\ \frac{dL}{dx_3} \end{bmatrix}$$

## Automatic differentiation with multiple inputs

$$-\log \sigma(w_1x_1 + w_2x_2 + w_3x_3)$$

$$\frac{dL}{d\mathbf{x}} = \begin{bmatrix} \frac{dL}{dx_1} \\ \frac{dL}{dx_2} \\ \frac{dL}{dx_3} \end{bmatrix}$$



## Reusing values

```
def loss(x):
    a = x ** 2
    b = 5 * a
    c = log(a)
    g = b * c
    L = -g
    return L
```

$$a = x^2$$

$$b = 5a$$

$$c = \log a$$

$$g = bc$$

$$L = -b$$

## Reusing values

$$a = x^2$$

$$b = 5a$$

$$c = \log a$$

$$g = bc$$

$$L = -b$$

## Reusing values

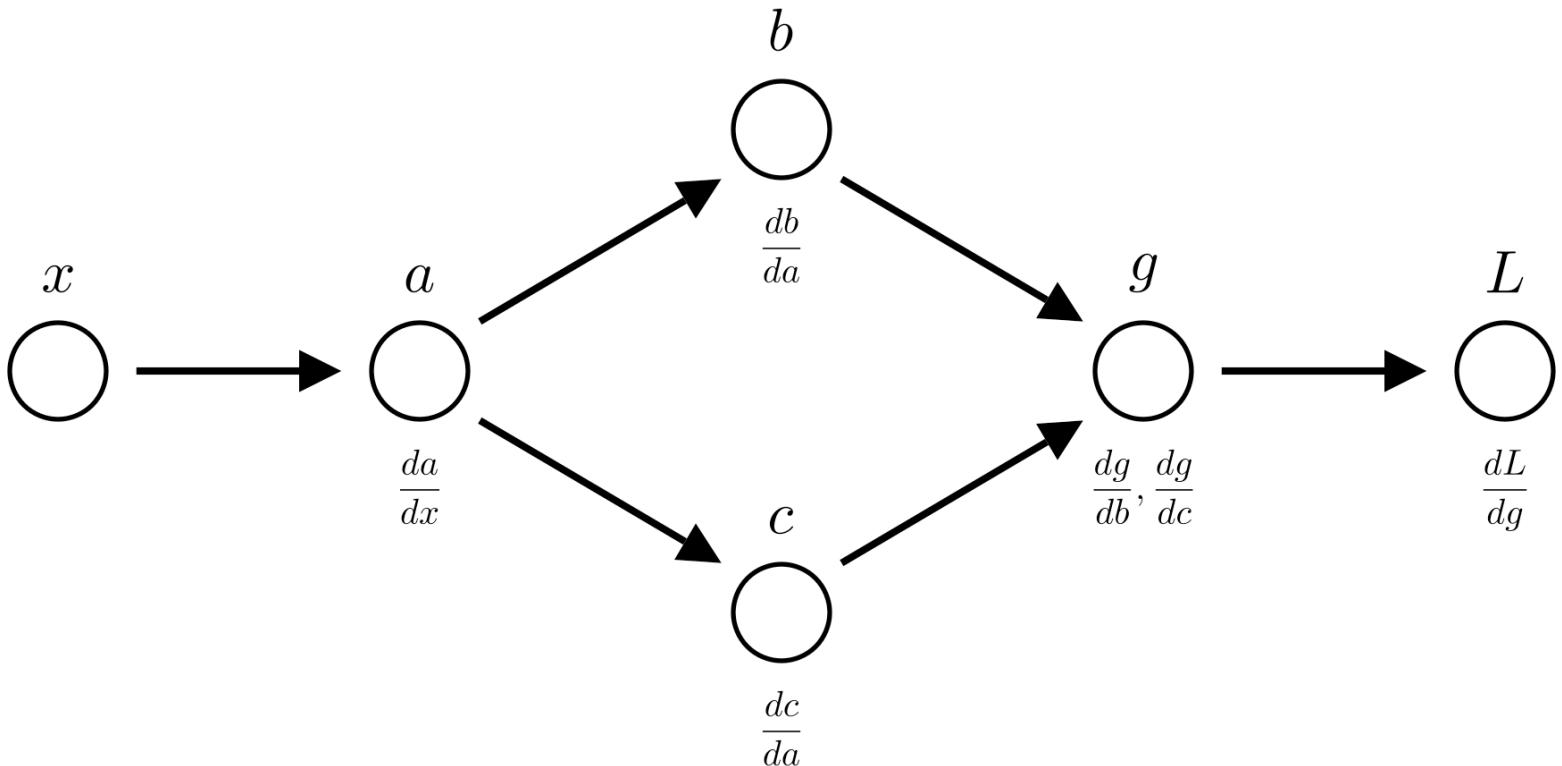
$$a = x^2$$

$$b = 5a$$

$$c = \log a$$

$$g = bc$$

$$L = -b$$



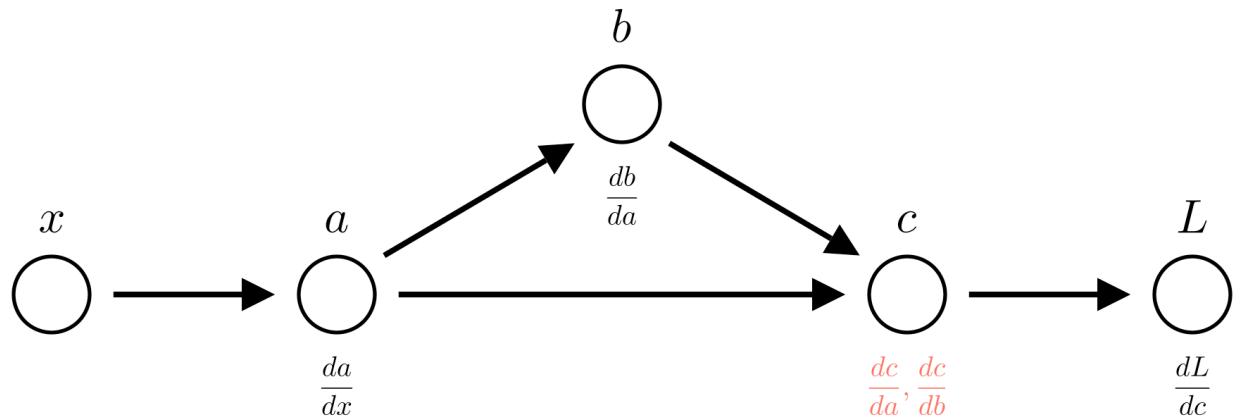
# Partial derivatives revisited

$$a = x^2$$

$$b = 5a$$

$$c = ab$$

$$L = -c$$



$$\frac{dc}{da} =$$

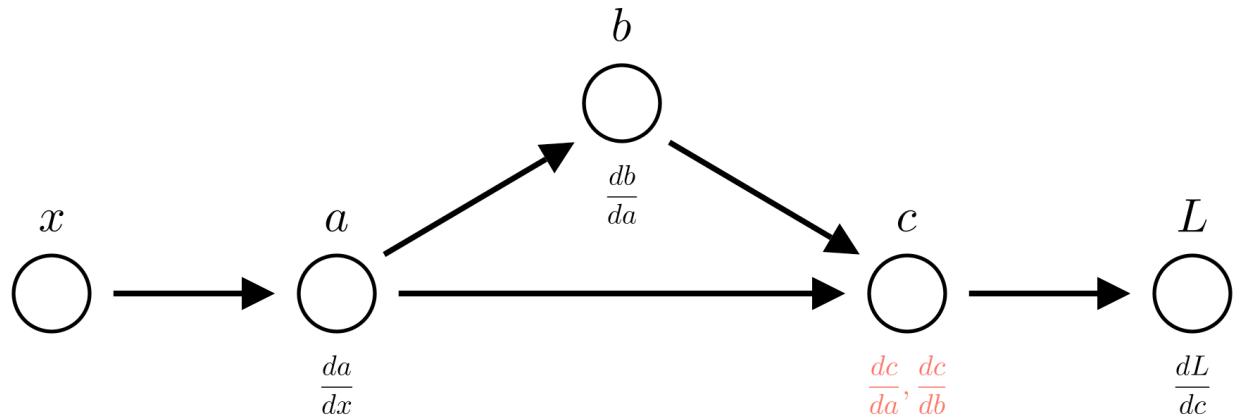
## Partial derivatives revisited

$$a = x^2$$

$$b = 5a$$

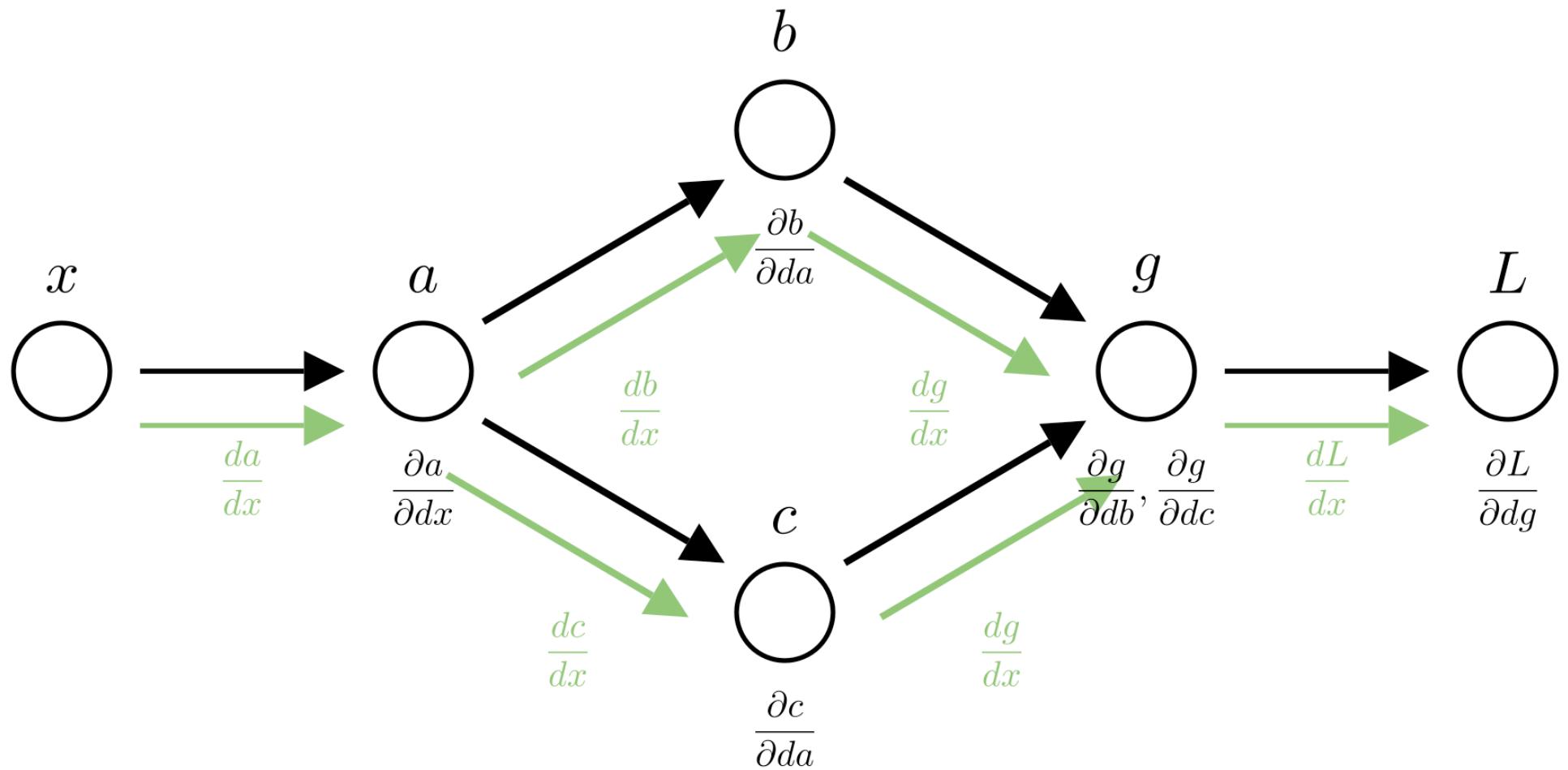
$$c = ab$$

$$L = -c$$

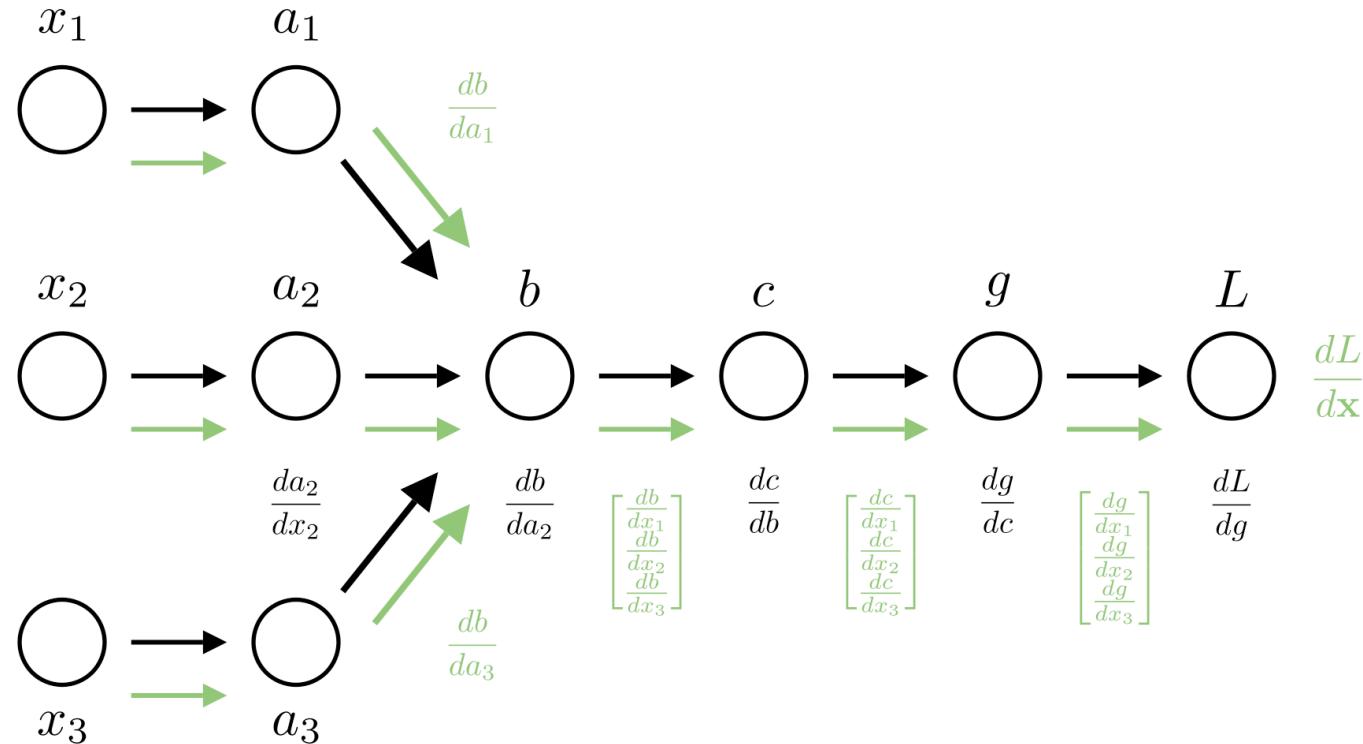


$$\frac{dc}{da} = \frac{\partial c}{\partial a} + \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} = 5a + 5a = 10a$$

## Partial derivatives revisited



# Implementing automatic differentiation



```

class ForwardValue:
    ...
    Base class for automatic differentiation operations. Represents variable declaration.
    Subclasses will overwrite func and grads to define new operations.

Properties:
    parent_values (list): A list of raw values of each input (as floats)
    value (float): The value of the result of this operation
    forward_grads (dict): A dictionary mapping inputs to gradients
    ...

def __init__(self, *args):
    self.parent_values = [arg.value if isinstance(arg, ForwardValue) else arg for arg in args]
    self.value = self.forward_pass(args)

    if len(self.forward_grads.keys()) == 0:
        self.forward_grads = {self: 1}

def func(self, input):
    ...
    Compute the value of the operation given the inputs.
    For declaring a variable, this is just the identity function (return the input).

    Args:
        input (float): The input to the operation
    Returns:
        value (float): The result of the operation
    ...
    return input

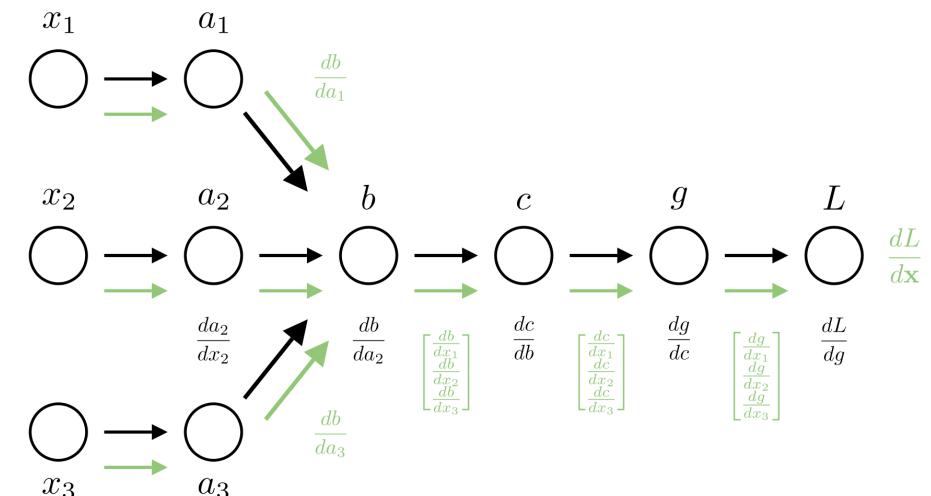
def grads(self, *args):
    ...
    Compute the derivative of the operation with respect to each input.
    In the base case the derivative of the identity function is just 1. (da/dx = 1).

    Args:
        input (float): The input to the operation
    Returns:
        grads (tuple): The derivative of the operation with respect to each input
            Here there is only a single input, so we return a length-1 tuple.
    ...
    return (1,)

def forward_pass(self, args):
    # Calls func to compute the value of this operation
    self.forward_grads = {}
    return self.func(*self.parent_values)

```

# Implementing automatic differentiation



# Implementing automatic differentiation

```

class _add(ForwardValue):
    # Addition operator (a + b)
    def func(self, a, b):
        return a + b

    def grads(self, a, b):
        return 1., 1.

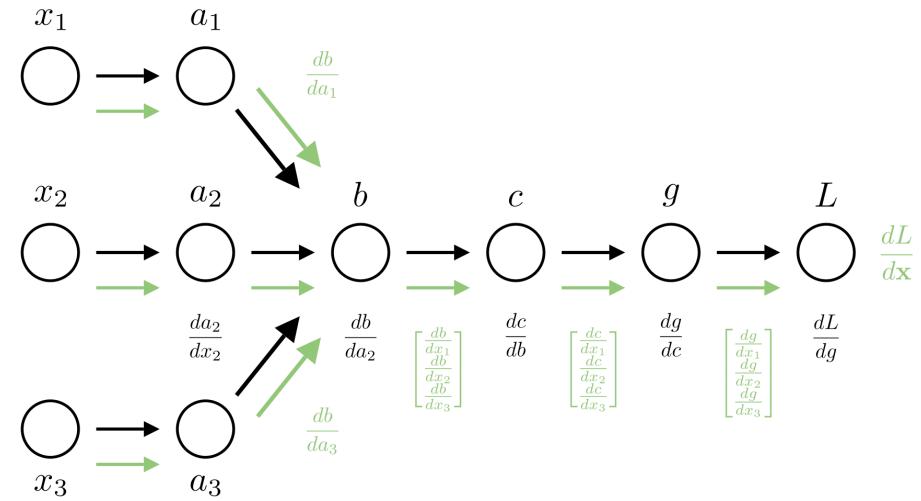
class _neg(ForwardValue):
    # Negation operator (-a)
    def func(self, a):
        return -a

    def grads(self, a):
        return (-1.,)

class _sub(ForwardValue):
    # Subtraction operator (a - b)
    def func(self, a, b):
        # Your code here

    def grads(self, a, b):
        # Your code here

```



# Calculus for vectors (Jacobian)

Now let's consider a vector-valued function  $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$

e.g.  $f(x_1, x_2, \dots, x_n) = (y_1, y_2, \dots, y_m)$

We define the instantaneous rate of change via the **Jacobian matrix**, whose entries are **partial derivatives**

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

The Jacobian is also called the **derivative** of  $f$ . It is a  $m \times n$  matrix

Each partial derivative fixes the other inputs and treats  $f$  as scalar-input scalar-output

## Calculus for vectors (Jacobian)

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

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

## Calculus for vectors (Jacobian)

$$\mathbf{y} = \mathbf{x}^2$$

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

# Jacobians and gradients

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

When  $m = 1$ , (scalar-valued output) the Jacobian is called the **gradient**

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \dots & \frac{\partial y}{\partial x_n} \end{bmatrix} = \nabla_{\vec{x}} f$$

Notes:

- The Jacobian of a scalar-valued function is a row vector.
- The gradient is often described as a column vector

## Chain rule for Jacobians

An easy extension of the chain rule exists:

Given  $f: \mathbb{R}^k \rightarrow \mathbb{R}^m$  and  $g: \mathbb{R}^n \rightarrow \mathbb{R}^k$

$$y = g(x)$$

$$f(y)$$

$$\frac{\partial f(g(x))}{\partial x} = \underbrace{\frac{\partial f(y)}{\partial y}}_{m \times k} \cdot \underbrace{\frac{\partial g(x)}{\partial x}}_{k \times n}$$

Where  $y = g(x)$ .

## Forward-mode with vectors

$$\mathbf{a} = \mathbf{X}\mathbf{w}$$

$$\mathbf{b} = \mathbf{y} - \mathbf{a}$$

$$\mathbf{c} = \mathbf{b}^2$$

$$g = \sum_{i=1}^N c_i$$

$$L = \frac{1}{N} g$$

# Forward-mode with vectors

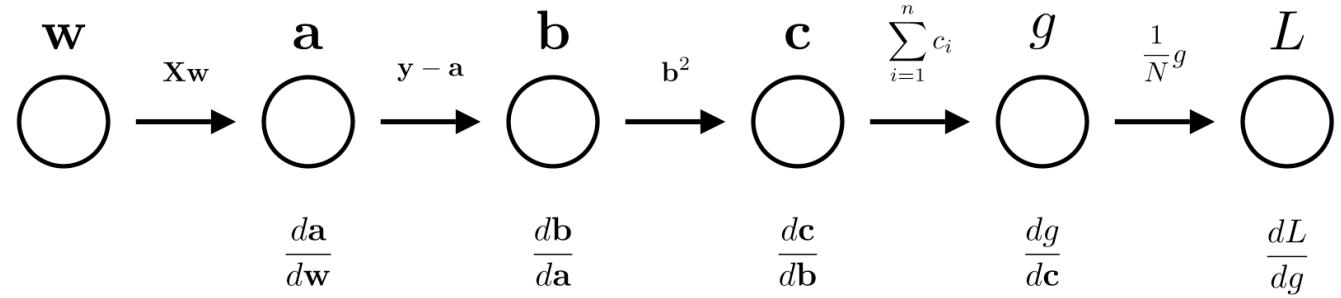
$$\mathbf{a} = \mathbf{X}\mathbf{w}$$

$$\mathbf{b} = \mathbf{y} - \mathbf{a}$$

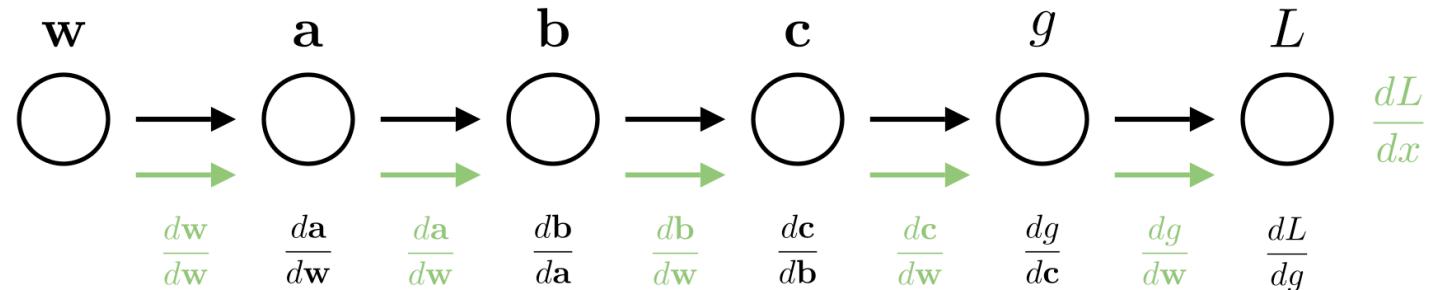
$$\mathbf{c} = \mathbf{b}^2$$

$$g = \sum_{i=1}^N c_i$$

$$L = \frac{1}{N} g$$

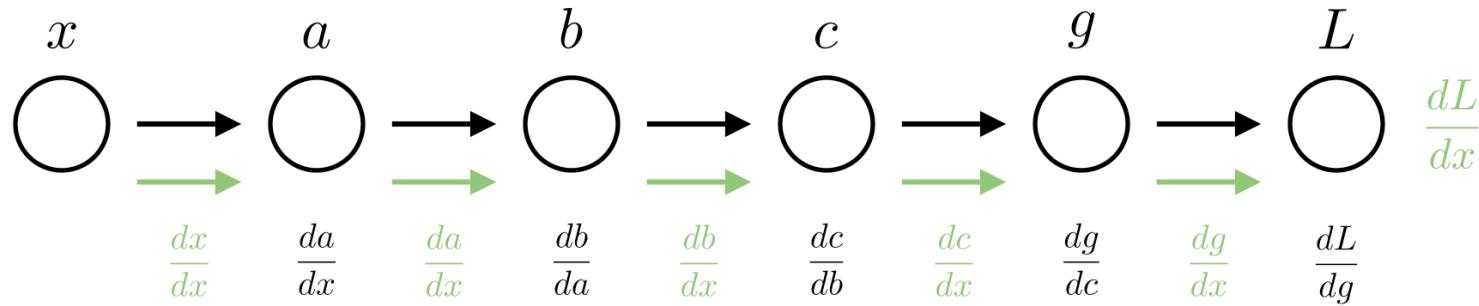


Forward mode

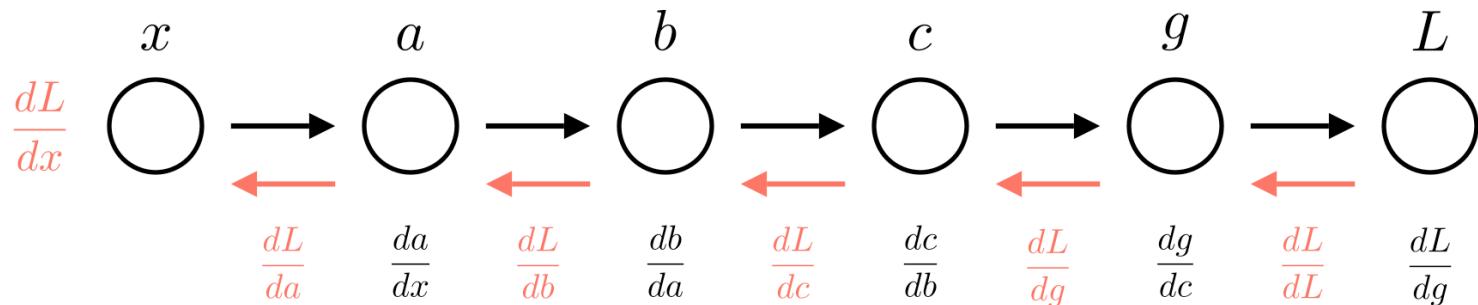


# Reverse-mode AD

## Forward mode

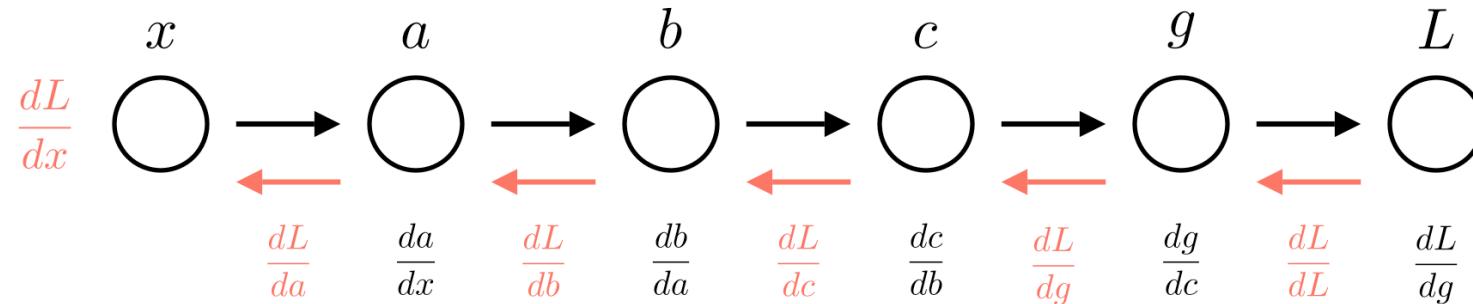


## Reverse mode



## Reverse mode

## Reverse-mode AD



Define inputs

```
a = AutogradValue(5)  
b = AutogradValue(2)
```

Compute operations

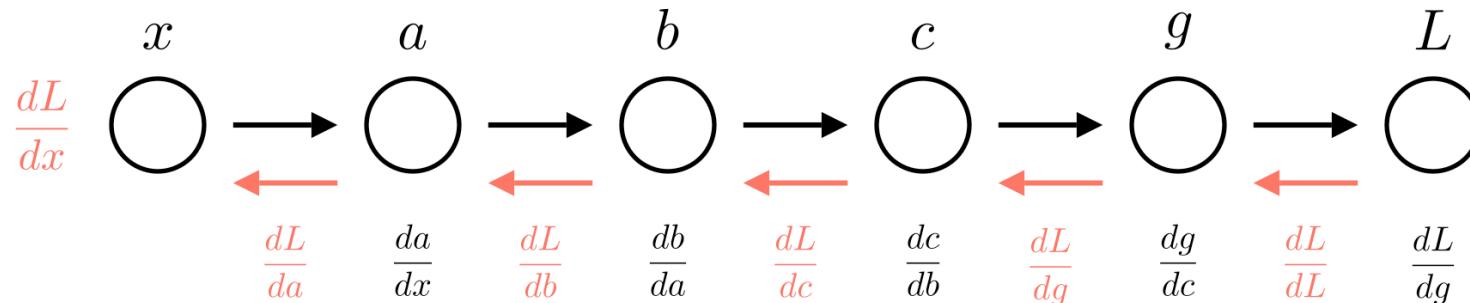
```
c = a + b  
L = log(c)
```

Get derivatives

```
L.backward()  
dL_da = a.grad
```

## Reverse mode

## Reverse-mode AD



Define inputs

```
a = AutogradValue(5)  
b = AutogradValue(2)
```

Compute operations

```
c = a + b  
L = log(c)
```

Get derivatives

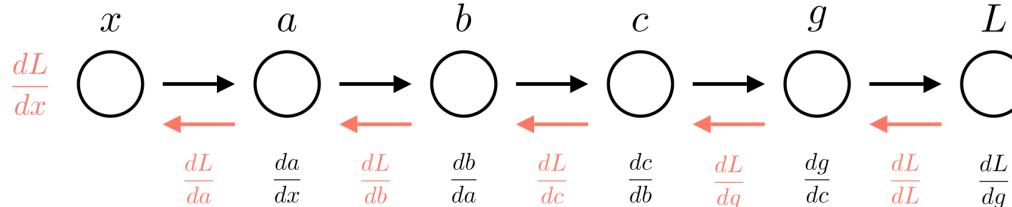
```
L.backward()  
dL_da = a.grad
```

Operations with non-Autograd values

```
s = 4  
L = s * a  
dL_da = a.grad # Will work because a is an AutogradValue  
dL_ds = s.grad # Will give an error because s is not an AutogradValue
```

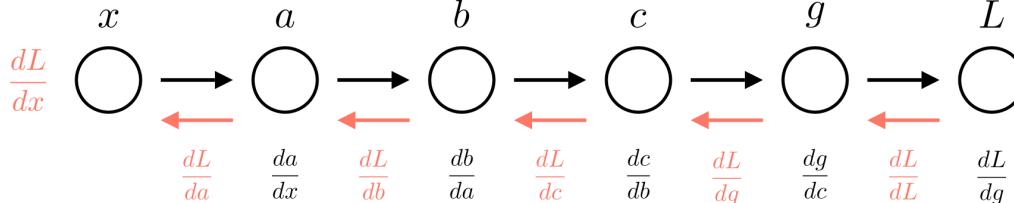
## Reverse mode

## Reverse-mode AD



```
class AutogradValue:  
    ...  
  
    Base class for automatic differentiation operations.  
    Represents variable delcaration. Subclasses will overwrite  
    func and grads to define new operations.  
  
Properties:  
    parents (list): A list of the inputs to the operation,  
        may be AutogradValue or float  
    args    (list): A list of raw values of each  
        input (as floats)  
    grad    (float): The derivative of the final loss with  
        respect to this value ( $dL/da$ )  
    value   (float): The value of the result of this operation  
    ...  
  
def __init__(self, *args):  
    self.parents = list(args)  
    self.args = [arg.value if isinstance(arg, AutogradValue)  
                else arg  
                for arg in self.parents]  
    self.grad = 0.  
    self.value = self.forward_pass()  
  
def forward_pass(self):  
    # Calls func to compute the value of this operation  
    return self.func(*self.args)
```

## Reverse mode



## Reverse-mode AD

```
class AutogradValue:
    ...
    Base class for automatic differentiation operations.
    Represents variable declaration. Subclasses will overwrite
    func and grads to define new operations.

Properties:
    parents (list): A list of the inputs to the operation,
        may be AutogradValue or float
    args (list): A list of raw values of each
        input (as floats)
    grad (float): The derivative of the final loss with
        respect to this value ( $dL/dx$ )
    value (float): The value of the result of this operation
    ...

def __init__(self, *args):
    self.parents = list(args)
    self.args = [arg.value if isinstance(arg, AutogradValue)
                else arg
                for arg in self.parents]
    self.grad = 0.
    self.value = self.forward_pass()

def forward_pass(self):
    # Calls func to compute the value of this operation
    return self.func(*self.args)
```

## Base class (input)

```
class AutogradValue:
    def func(self, input):
        ...
        Compute the value of the operation given the inputs.
        For declaring a variable, this is just the identity
        function (return the input).

    Args:
        input (float): The input to the operation
    Returns:
        value (float): The result of the operation
    ...
    return input
```

## Sub-class (operation)

```
class _square(AutogradValue):
    # Square operator ( $a ** 2$ )
    def func(self, a):
        return a ** 2
```

```
x = AutogradValue(2)  
a = x ** 2  
b = 5 * a  
c = b * a  
L = -c
```

## Forward pass

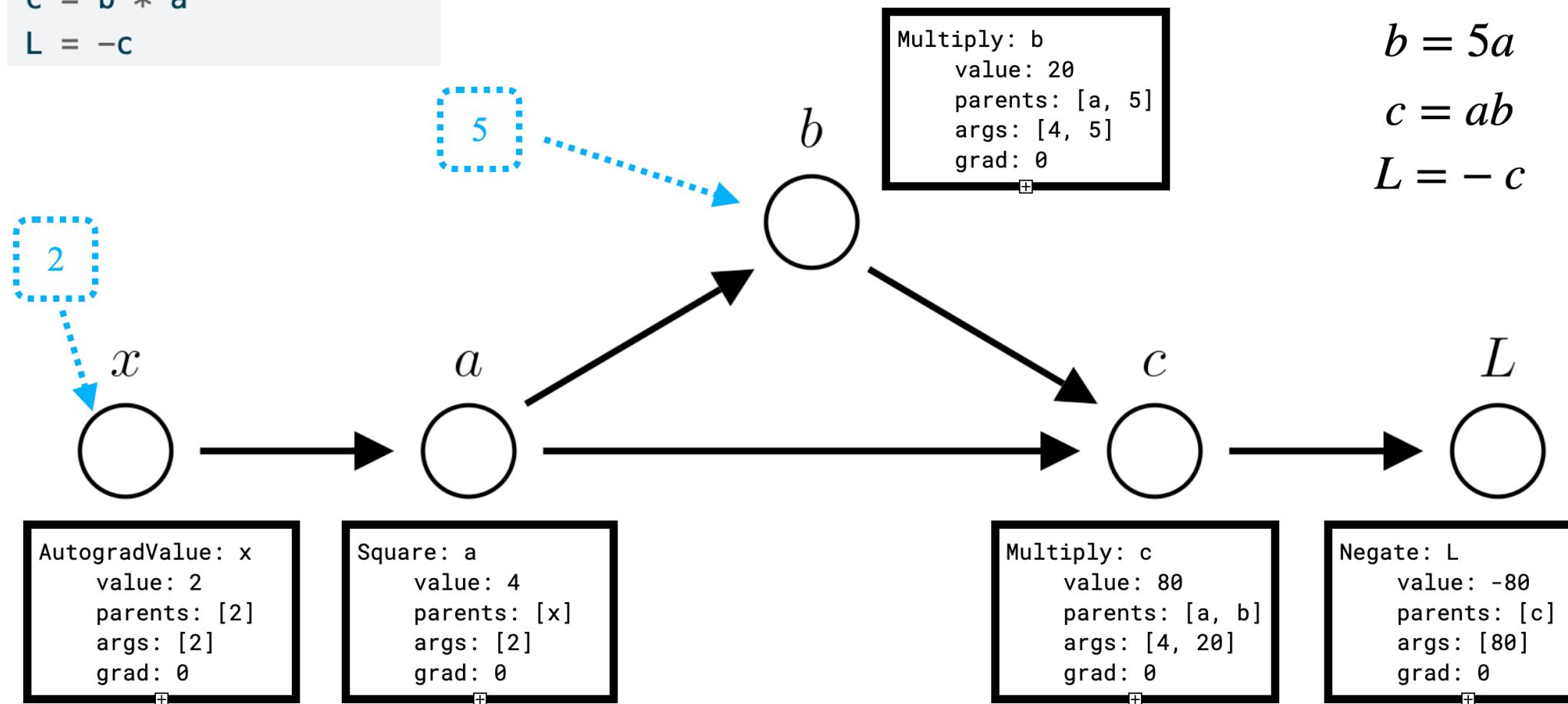
$$x = 2$$

$$a = x^2$$

$$b = 5a$$

$$c = ab$$

$$L = -c$$



```

x = AutogradValue(2)
a = x ** 2
b = 5 * a
c = b * a
L = -c

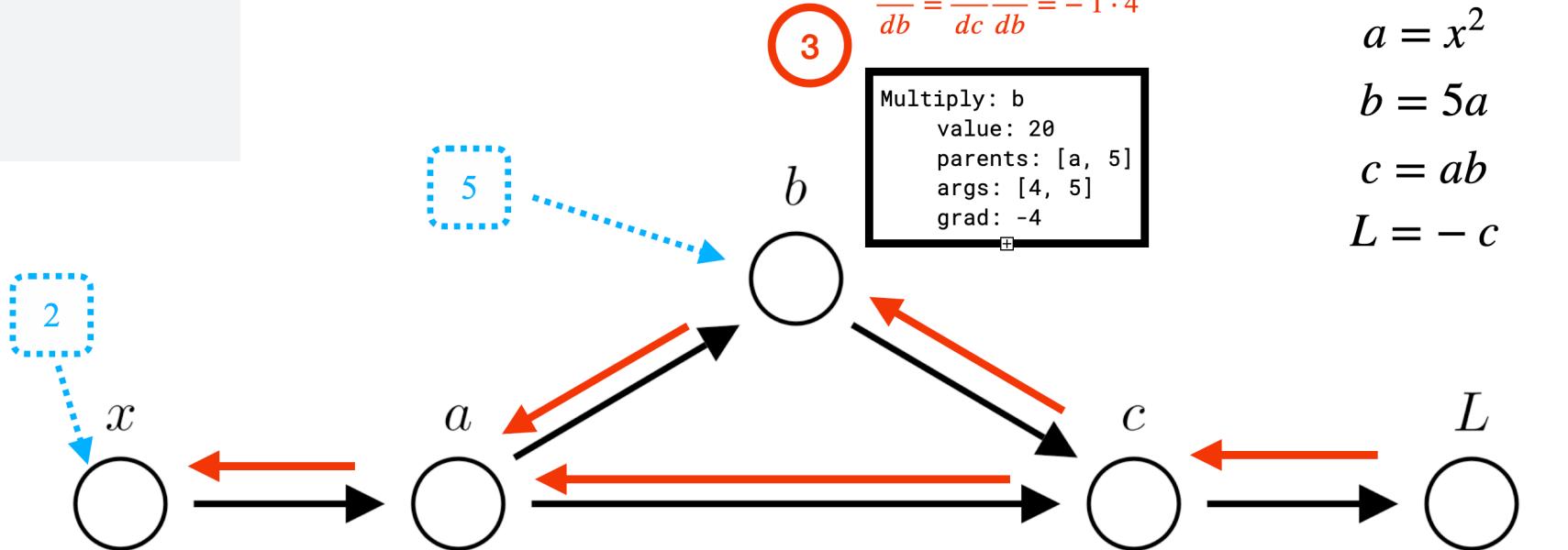
```

```

L.backward()
print('dL_dx', x.grad)

```

## Backward pass



$$\frac{dL}{dx} = \frac{dL}{da} \frac{da}{dx} = -40 \cdot 4$$

5

$$\frac{dL}{da} = \frac{dL}{dc} \frac{dc}{da} + \frac{dL}{db} \frac{db}{da} = -1 \cdot 20 + -4 \cdot 5$$

4

$$\frac{dL}{dc} = -1$$

2

$$\frac{dL}{dL} = 1$$

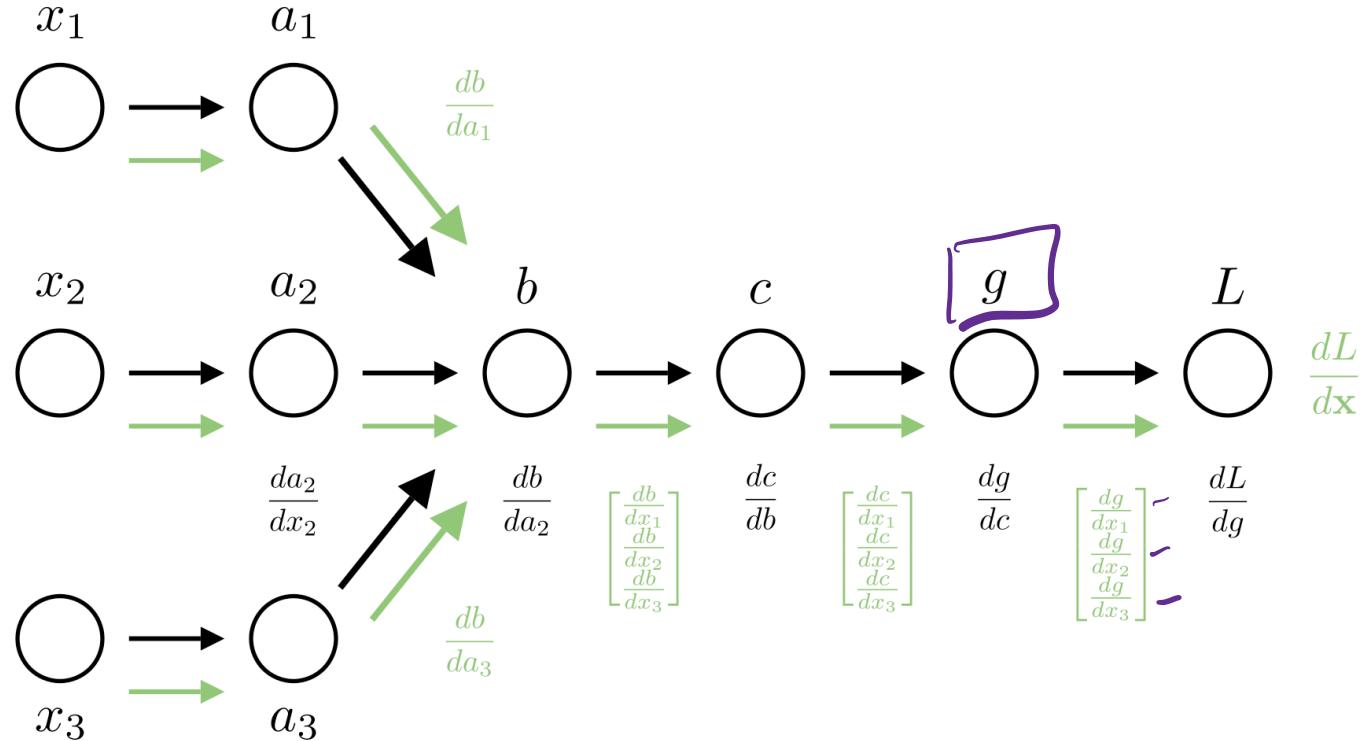
1

$x = 2$   
 $a = x^2$   
 $b = 5a$   
 $c = ab$   
 $L = -c$

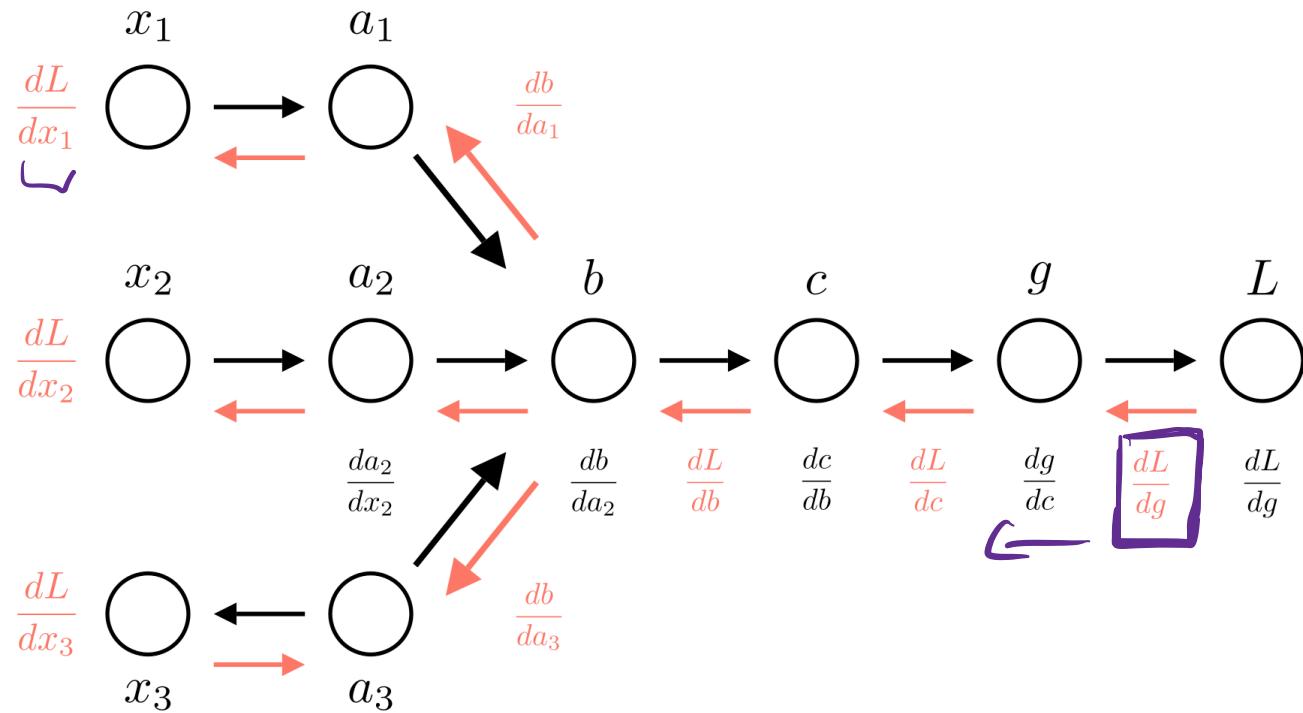
# Backward pass

```
class AutogradValue:  
    def grads(self, *args):  
        ...  
        Compute the derivative of the operation with respect to each input.  
        In the base case the derivative of the identity function is just 1. ( $da/d_a = 1$ ).  
  
        Args:  
            input (float): The input to the operation  
        Returns:  
            grads (tuple): The derivative of the operation with respect to each input  
                Here there is only a single input, so we return a length-1 tuple.  
                ...  
            return (1,)  
  
class _square(AutogradValue):  
    # Square operator ( $a ** 2$ )  
    def grads(self, a):  
        return (2 * a,)
```

# Why reverse mode?



# Why reverse mode?



## Reverse-mode with vectors

$$x_i \in \mathbb{R}^{N \times l}$$

$$\underline{\mathbf{a}} = \mathbf{X}\mathbf{w}$$

$$\mathbf{b} = \mathbf{y} - \mathbf{a}$$

*outputs*

$$\mathbf{c} = \mathbf{b}^2$$

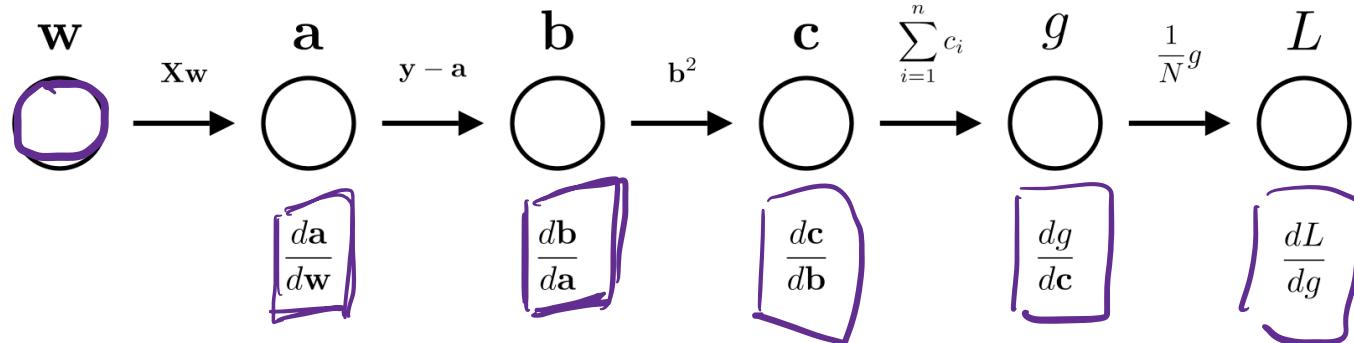
*N inputs*      *Length*  
*N vectors*

$$g = \sum_{i=1}^N c_i$$

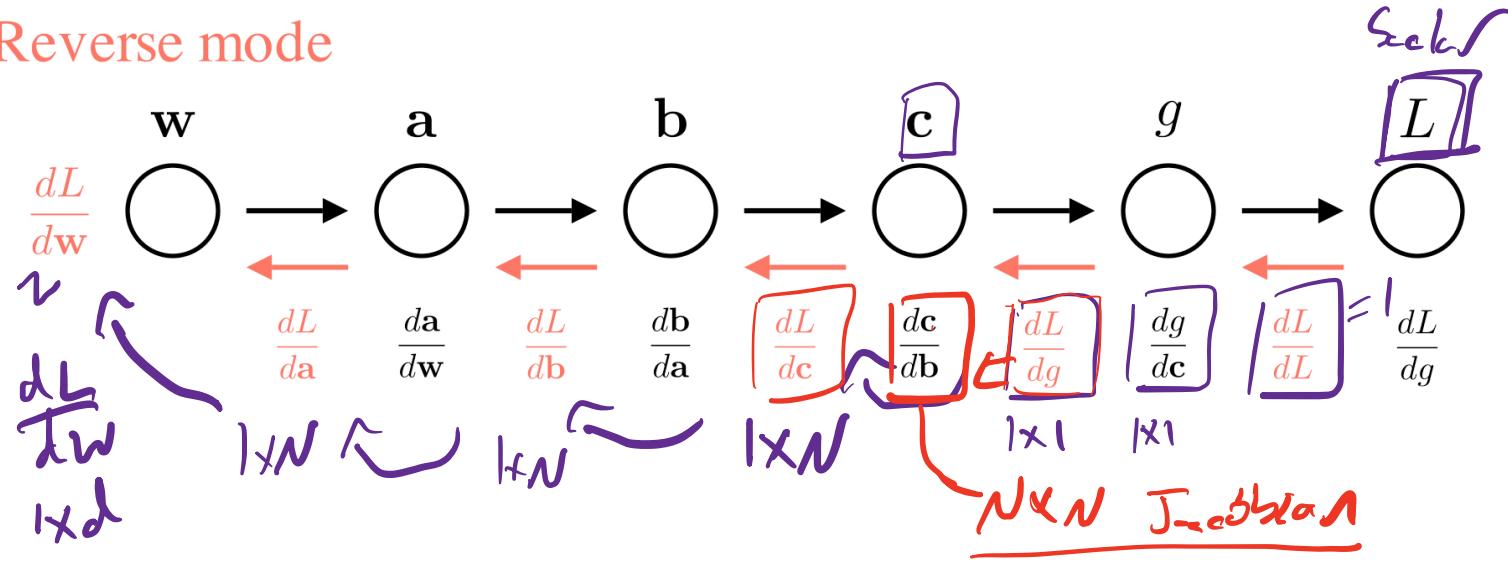
$$\frac{L}{\lambda} = \frac{1}{N} g$$

gadgets  
lengths  
variables

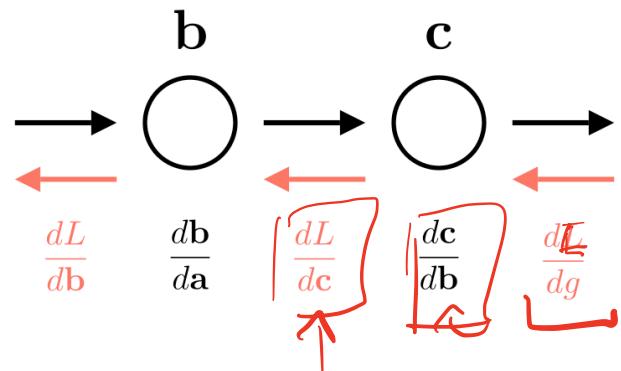
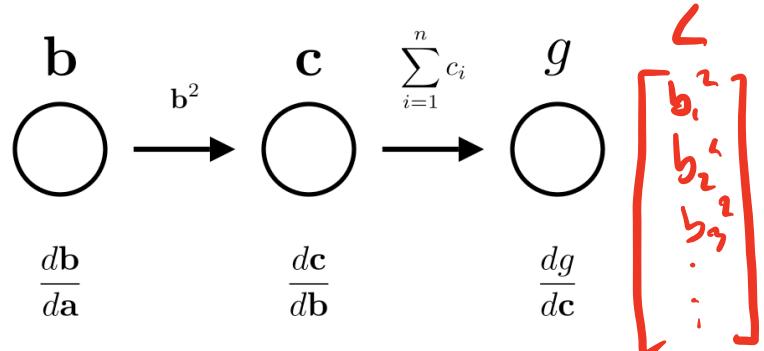
Loss



# Reverse mode



## Jacobian computation



$$\frac{dL}{db} = \frac{dL}{dc} \frac{dc}{db} = \underbrace{\frac{dL}{dc}}_{N \text{ vec.}} \underbrace{\frac{dc}{db}}_{N \times N} = O(n^2)$$

$$\frac{dc}{db} = \left[ \frac{\frac{\partial c_i}{\partial b_1}}{\vdots} \frac{\frac{\partial c_i}{\partial b_j}}{\vdots} \right] = \left[ \begin{array}{cccc} 2b_1 & 0 & 0 & \dots \\ 0 & 2b_2 & 0 & \dots \\ 0 & 0 & 2b_3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array} \right]$$

$$\left[ \frac{\frac{dL}{dc_1}}{\vdots} \frac{\frac{dL}{dc_i}}{\vdots} \frac{\frac{dL}{dc_n}}{\vdots} \right] \left[ \begin{array}{c} 2b_1 \\ 2b_2 \\ \vdots \\ 2b_n \end{array} \right] = \left[ \begin{array}{c} \frac{dL}{dc_1} \cdot 2b_1 \\ \vdots \\ \frac{dL}{dc_i} \cdot 2b_i \\ \vdots \\ \frac{dL}{dc_n} \cdot 2b_n \end{array} \right] = O(n)$$

$$c_i = b_i^2$$

$$\frac{\partial c_i}{\partial b_j} = 2b_i \quad \text{if } i=j$$

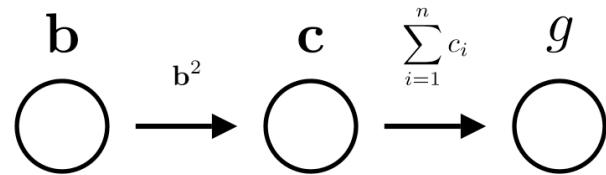
$O:$  for  $i \neq j$

$$c = b^2, \quad \mathbb{R}^n \rightarrow \mathbb{R}^n$$

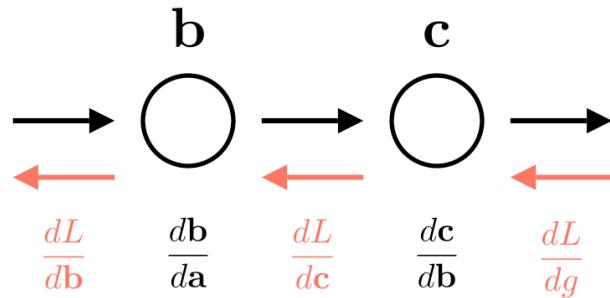
$L$  length  $n$  vectors

$n$

# Jacobian computation



$$\frac{db}{da}, \quad \frac{dc}{db}, \quad \frac{dg}{dc}$$

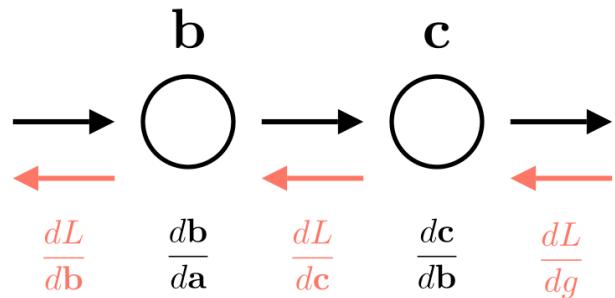
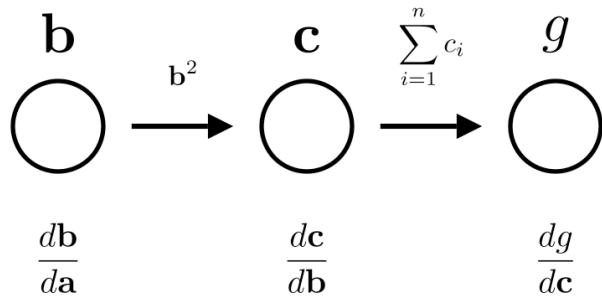


$$\mathbf{c} = \mathbf{b}^2, \quad \mathbb{R}^n \rightarrow \mathbb{R}^n$$

$$\frac{d\mathbf{c}}{d\mathbf{b}} = \begin{bmatrix} \frac{\partial c_1}{\partial b_1} & 0 & \dots & 0 \\ 0 & \frac{\partial c_2}{\partial b_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial c_N}{\partial b_d} \end{bmatrix}$$

$$\frac{dL}{d\mathbf{b}} = \frac{dL}{d\mathbf{c}} \frac{d\mathbf{c}}{d\mathbf{b}} =$$

## Vector-Jacobian Product



$$\frac{d\mathbf{c}}{d\mathbf{b}} = \begin{bmatrix} \frac{\partial c_1}{\partial b_1} & 0 & \dots & 0 \\ 0 & \frac{\partial c_2}{\partial b_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial c_N}{\partial b_d} \end{bmatrix}$$

$$\frac{dL}{d\mathbf{b}} = \frac{dL}{d\mathbf{c}} \frac{d\mathbf{c}}{d\mathbf{b}} = \begin{bmatrix} \frac{dL}{dc_1} \frac{dc_1}{db_1} \\ \frac{dL}{dc_2} \frac{dc_2}{db_2} \\ \vdots \\ \frac{dL}{dc_N} \frac{dc_N}{db_N} \end{bmatrix}$$

## Jacobian computation

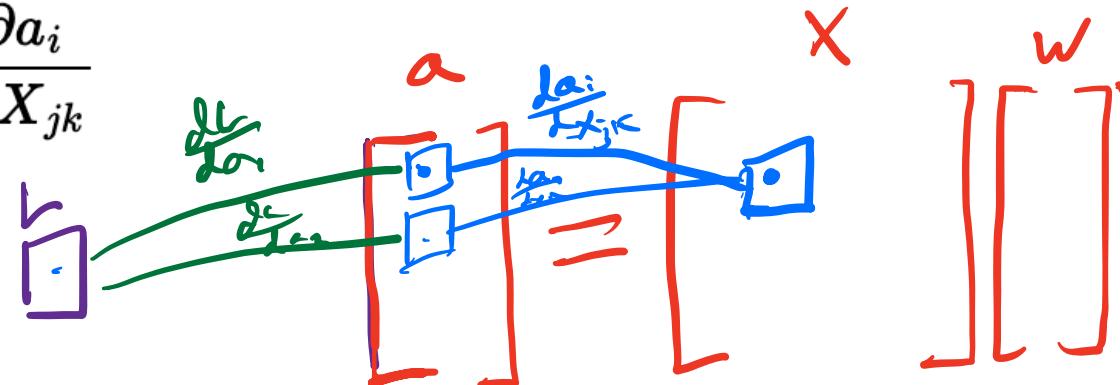
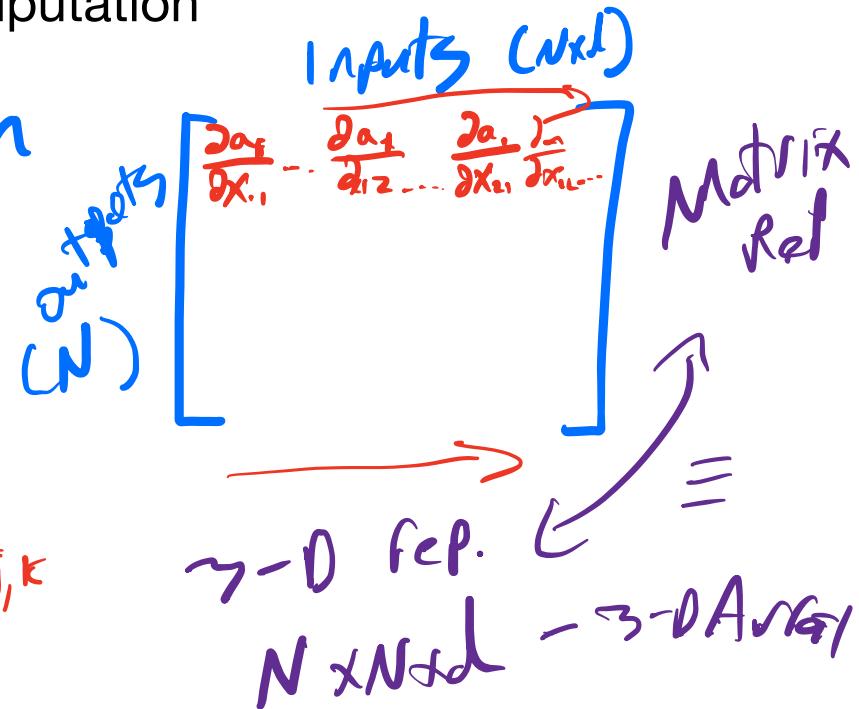
With  $\mathbf{a} = \mathbf{X}\mathbf{w}$ ,  $\frac{\partial \mathbf{a}}{\partial \mathbf{X}}$  is  $N \times d$  by  $d$ .

$$\frac{dL}{d\mathbf{X}} = \frac{dL}{da}^T \frac{da}{d\mathbf{X}}$$

$(w)$   $\leftarrow$   $1 \times N$   $N \times (N+d)$

$$\frac{dL}{dX_{jk}} = \sum_{i=1}^N \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial X_{jk}}$$

$$\frac{\partial a_i}{\partial X_{jk}} \forall i, j, k$$



$$\mathbf{a} = \mathbf{X}\mathbf{w}, \quad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{d\mathbf{X}} = \begin{bmatrix} \frac{dL}{d\mathbf{a}}^T & \frac{d\mathbf{a}}{d\mathbf{X}} \end{bmatrix}$$

$\nwarrow N \times d \quad \swarrow 1 \times N$

$$\frac{dL}{d\mathbf{X}_{jk}} = \sum_{i=1}^N \frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{X}_{jk}}$$

$\nwarrow N \times d \quad \swarrow 1 \times N$

$$(1 \times (N \times d)) \cdot (1 \times N) \cdot (N \times (N \times d))$$

## Jacobian computation

```

dL_da, da_dx # The computed gradients/jacobians
da_dx = da_dx.reshape((da_dx.shape[0], -1)) → Nx(Nxd)
dL_dx = np.dot(dL_da, da_dx) → 1 × N · (N × (Nxd)) → 1 × (Nxd)
dL_dx = dL_dx.reshape(x.shape)
    ↓
    1 × (Nxd) → Nx d
    ↙
    ↘
    1 × (Nxd)
  
```

$O(N \times N \times N \times d)$

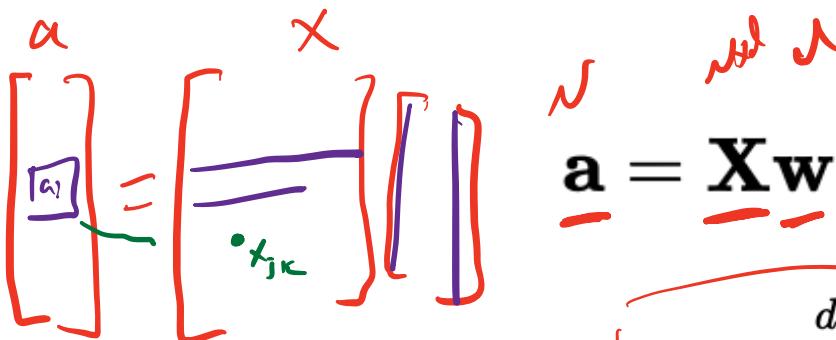
## Vector-Jacobian Product

$$\mathbf{a} = \mathbf{X}\mathbf{w}, \quad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{d\mathbf{X}} = \frac{dL}{d\mathbf{a}}^T \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{dX_{jk}} = \sum_{i=1}^N \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial X_{jk}}$$

0 if  $i \neq j$



$$\mathbf{a} = \mathbf{X}\mathbf{w}, \quad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$a_i = \sum_{k=1}^d X_{ik} w_k$$

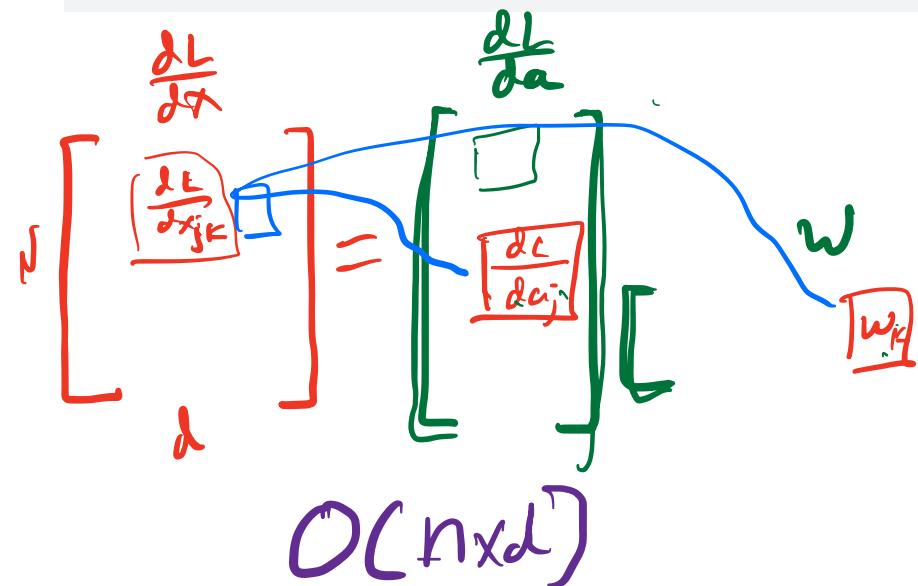
$$\underbrace{\frac{\partial a_i}{\partial X_{jk}}}_{\text{if } i=j} = \underbrace{\frac{\partial}{\partial X_{jk}}}_{\text{if } i=j} \sum_{k=1}^d X_{ik} w_k = \underbrace{\mathbb{I}(i=j) w_k}_{\text{if } i=j}$$

$$\underbrace{\frac{\partial L}{\partial X_{ik}}}_{\text{if } i=j} = \underbrace{\frac{\partial L}{\partial a_i}}_{\text{if } i=j} \underbrace{\frac{\partial a_i}{\partial X_{ik}}}_{\text{if } i=j} = \underbrace{\frac{\partial L}{\partial a_i}}_{\text{if } i=j} w_k$$

# Vector-Jacobian Product

```
w_mat = w.reshape((1, -1)) # reshape w to 1 x d
dL_da_mat = dL_da.reshape((-1, 1)) # reshape dL_da to N x 1
dL_dx = w_mat * dL_da_mat # dL_dx becomes N x d, entry ik = dL_da_i * w_k
```

```
dL_da, da_dx # The computed gradients/jacobians
da_dx = da_dx.reshape((da_dx.shape[0], -1))
dL_dx = np.dot(dL_da, da_dx)
dL_dx = dL_dx.reshape(x.shape)
```



$$\mathbf{a} = \mathbf{X}\mathbf{w}, \quad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$a_i = \sum_{k=1}^d X_{ik} w_k$$

$$\frac{\partial a_i}{\partial X_{jk}} = \frac{\partial}{\partial X_{jk}} \sum_{k=1}^d X_{ik} w_k = \mathbb{I}(i=j) w_k$$

$$\frac{\partial L}{\partial X_{ik}} = \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial X_{ik}} = \frac{\partial L}{\partial a_i} w_k$$

## Vector-Jacobian products

$$\mathbb{R}^n \rightarrow \mathbb{R}^n$$

$$f(\mathbf{x}) = \mathbf{x} + \mathbf{a}$$

$$\frac{dL}{dx} = \frac{dL^T}{df} \left[ \frac{df}{dx} \right] = \frac{dL}{df}$$

$$f = x + a$$

$$\frac{\partial f_i}{\partial x_i} = \frac{\partial}{\partial x_i} [x_i + a_i] = 1$$

Element-wise operation

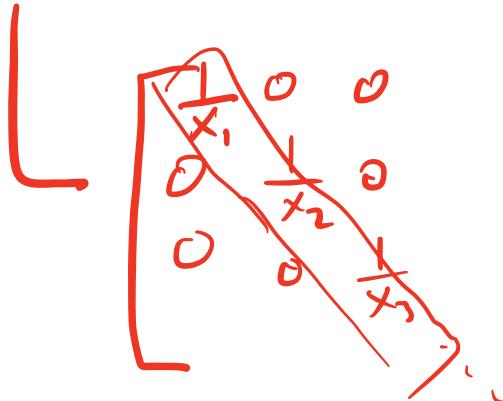
→ Jacobian will be diagonal!

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Vector-Jacobian products

$$f(\mathbf{x}) = \log \mathbf{x} \quad , \text{element-wise} \quad f = \log(\mathbf{x}) \quad f = \begin{bmatrix} \log x_1 \\ \log x_2 \\ \vdots \end{bmatrix}$$

$$\frac{dL^T}{d\mathbf{f}} \frac{d\mathbf{f}}{d\mathbf{x}} = \frac{\frac{dL}{df}}{x}$$



```
x = AutogradValue(2)
```

```
a = x ** 2
```

```
b = 5 * a
```

```
c = b * a
```

```
L = -c
```

```
L.backward()
```

```
print('dL_dx', x.grad)
```

## Backward pass

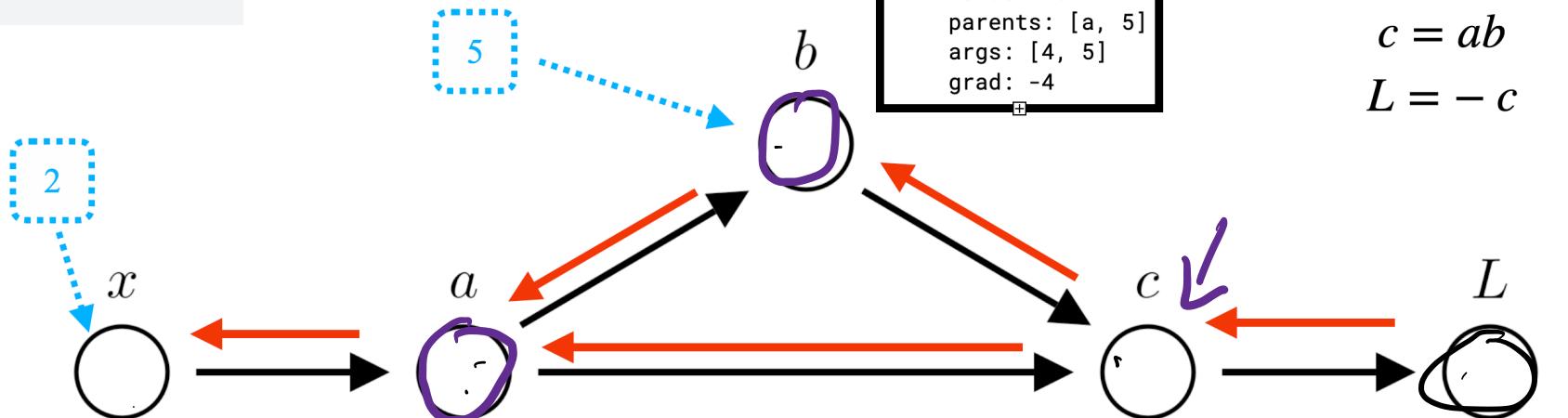
$$x = 2$$

$$a = x^2$$

$$b = 5a$$

$$c = ab$$

$$L = -c$$



$$\frac{dL}{dx} = \frac{dL}{da} \frac{da}{dx} = -40 \cdot 4$$

5

$$\frac{dL}{da} = \frac{dL}{dc} \frac{dc}{da} + \frac{dL}{db} \frac{db}{da} = -1 \cdot 20 + -4 \cdot 5$$

4

$$\frac{dL}{db} = \frac{dL}{dc} \frac{dc}{db} = -1 \cdot 4$$

3

2

1

update for node  $n$

$$\frac{dL}{dn} = \frac{dL}{dc_1} \cdot \frac{dc_1}{dn} + \frac{dL}{dc_2} \cdot \frac{dc_2}{dn} \dots$$

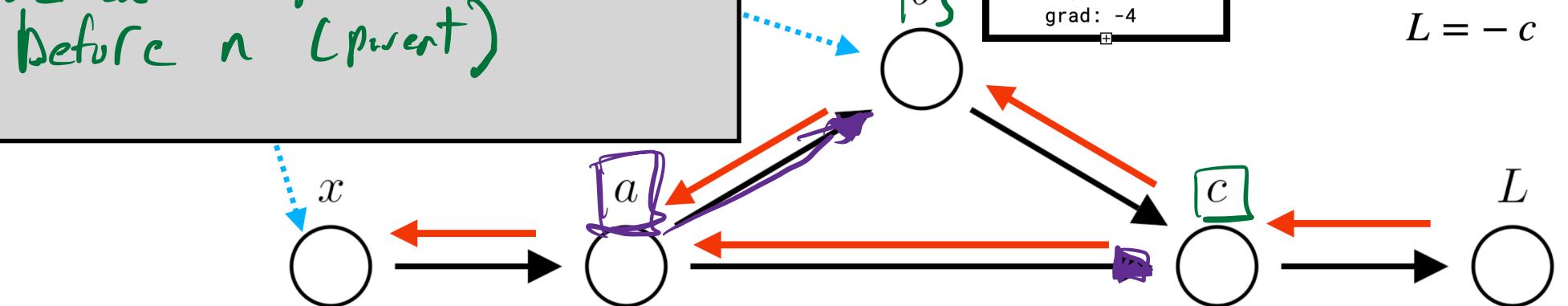
for  $c_i$  in the children of  $n$   
we need to update all children  
before  $n$  (parent)

Backward pass

$$\frac{dL}{db} = \frac{dL}{dc} \frac{dc}{db} = -1 \cdot 4$$

Multiply: b  
value: 20  
parents: [a, 5]  
args: [4, 5]  
grad: -4

$$x = 2 \\ a = x^2 \\ b = 5a \\ c = ab \\ L = -c$$



AutogradValue: x  
value: 2  
parents: [2]  
args: [2]  
grad: -160

Square: a  
value: 4  
parents: [x]  
args: [2]  
grad: -40

Multiply: c  
value: 80  
parents: [a, b]  
args: [4, 20]  
grad: -1

Negate: L  
value: -80  
parents: [c]  
args: [80]  
grad: 1

$$\frac{dL}{dx} = \frac{dL}{da} \frac{da}{dx} = -40 \cdot 4$$

$$\frac{dL}{da} = \frac{dL}{dc} \frac{dc}{da} + \frac{dL}{db} \frac{db}{da} = -1 \cdot 20 + -4 \cdot 5$$

$$\frac{dL}{dc} = -1$$

$$\frac{dL}{dL} = 1$$

5

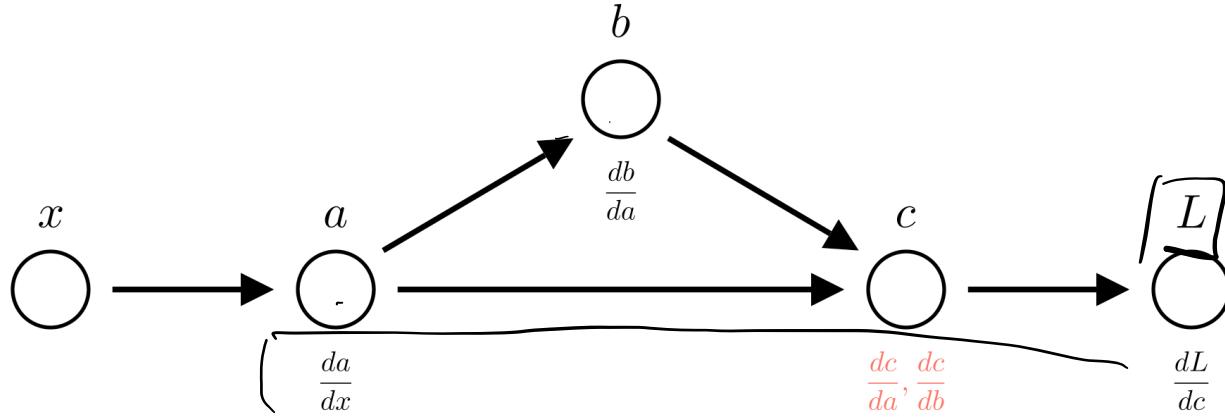
4

2

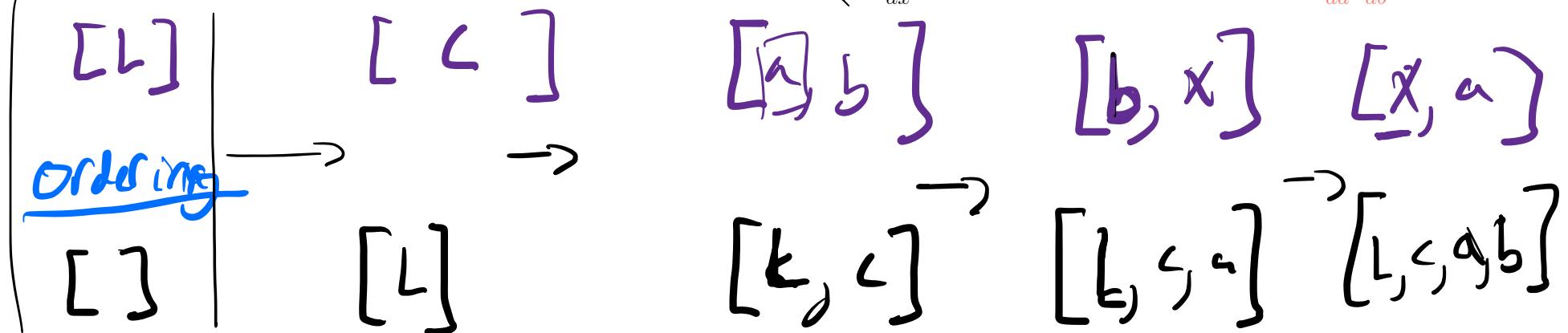
1

# A simple topological sort

Goal: an ordering of nodes s.t. if A is a child of B then A is before B in the ordering  $\delta(A, B)$



queue



$[a]$

$[x]$

$[]$

$$\xrightarrow{[L, c, a, b, x]} \xrightarrow{[L, c, \cancel{b}, x, a]} \xrightarrow{[L, c, b, \cancel{a}, x]} \rightarrow \text{Final order: } [L, c, b, a, x]$$

Traverse the graph backwards, replace entries we've already seen