VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

AIML231/DATA302 —Techniques in Machine Learning

Week 9 Neural Networks (2)

Automatic Differentiation

Dr Qi Chen

School of Engineering and Computer Science

Victoria University of Wellington

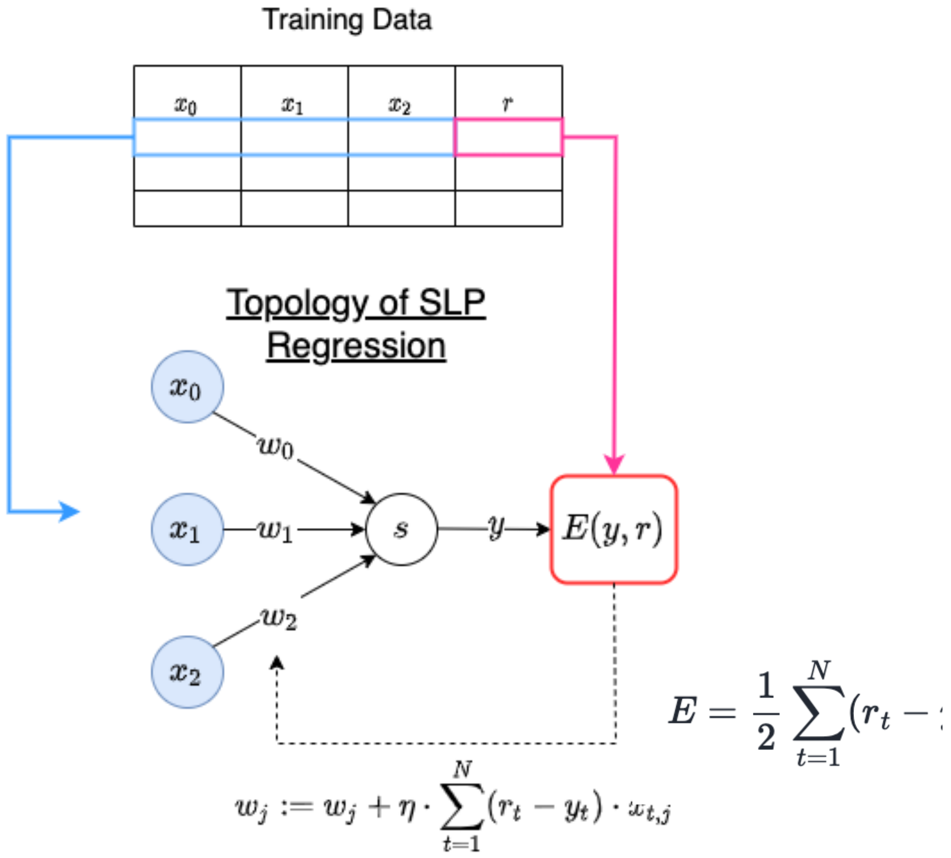
Qi.Chen@vuw.ac.nz

Outline

- Definition of Automatic Differentiation
- Basics of Automatic Differentiation
  - Computational graphs
  - Forward mode vs. Reverse mode
  - Importance of reverse mode in neural networks
- Backpropagation and Automatic Differentiation
- Automatic Differentiation Algorithm
- Tools and Libraries Supporting Automatic Differentiation

Neural Network Learning - Recap

- NN learning
  - adjust weights based on the loss between the predicted outputs and the actual outputs.
  - this adjustment is made possible through an optimization algorithm called gradient descent
  - the gradient—essentially a derivative—indicates the direction and magnitude of weight adjustments to reduce loss
  - differentiation allows us to compute these gradients.



$$E = \frac{1}{2}\sum_{t=1}^{N}(r_t -$$

$$w_j := w_j + \eta \cdot \sum_{t=1}^{N}(r_t - y_t) \cdot x_{t,j}$$

L1 Regularization

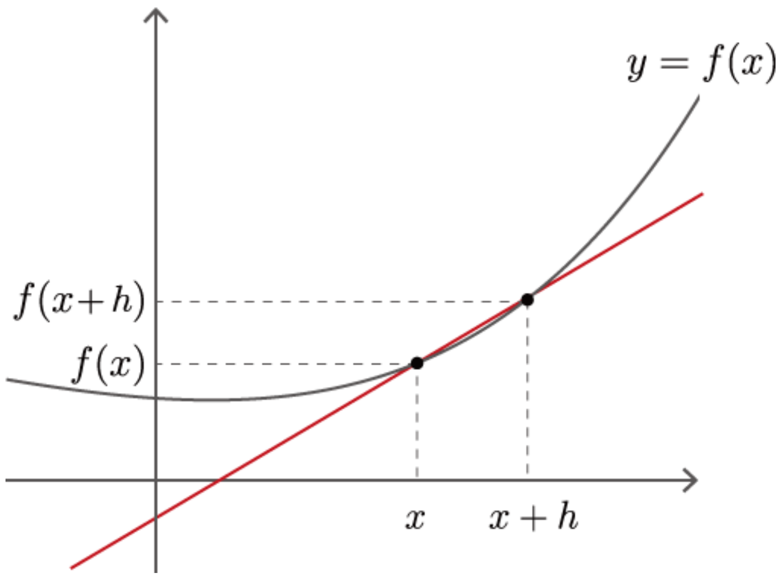
Modified loss function = Loss function + $\lambda\sum_{i=1}^{n}|W_i|$

L2 Regularization

Modified loss function = Loss function + $\lambda\sum_{i=1}^{n}W_i^2$

Traditional Differentiation Methods

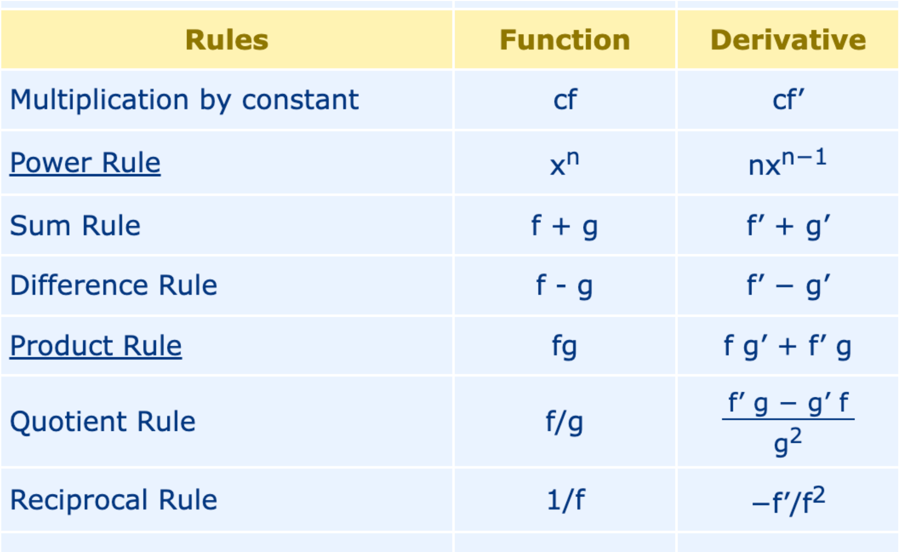
- Numerical Differentiation: approximate derivatives by finite differences
  - straightforward but can lead to significant rounding errors and inefficiencies, especially in high-dimensional space
- Symbolic Differentiation: computes derivatives symbolically or analytically (using symbols) use rules
  - can handle complex expressions
  - often lead to inefficient code and suffers from expression swell, making it impractical



$$
\begin{aligned}
\frac{df(x)}{dx} &= \frac{f(x+h)-f(x)}{h} \\
&= \frac{(x+h)^2 - x^2}{h} \\
&= \frac{x^2 + 2xh + h^2 - x^2}{h} \\
&= \frac{2xh + h^2}{h} \\
&= \frac{h(2x+h)}{h} \\
&= 2x + h
\end{aligned}
$$

$$\text{If } f(x) = x^n \text{ then } \frac{df(x)}{dx} = nx^{n-1}$$

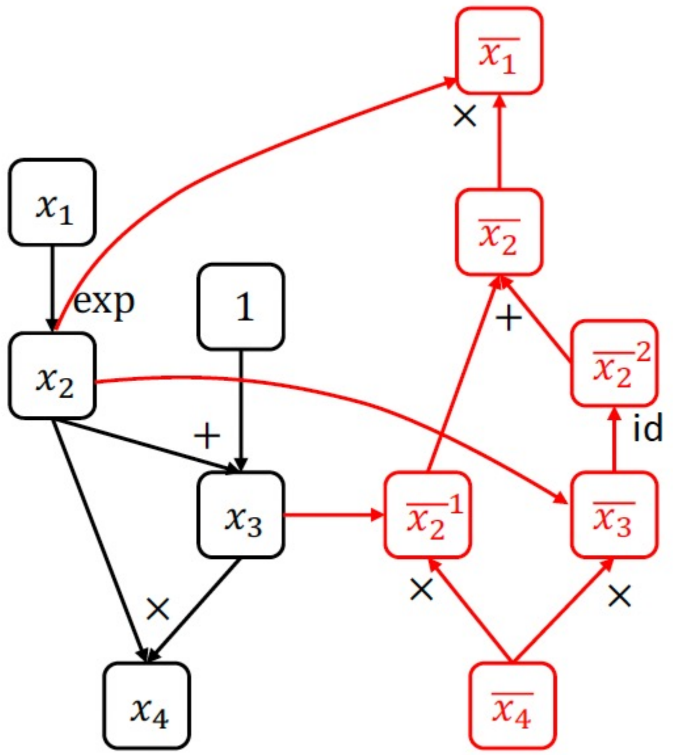
$$\text{If } f(x) = k \text{ then } \frac{df(x)}{dx} = 0$$



Rules	Function	Derivative
Multiplication by constant	cf	cf′
Power Rule	x^n	nx^{n-1}
Sum Rule	f + g	f′ + g′
Difference Rule	f - g	f′ − g′
Product Rule	fg	f g′ + f′ g
Quotient Rule	f/g	$\frac{f' g - g' f}{g^2}$
Reciprocal Rule	1/f	$-f'/f^2$

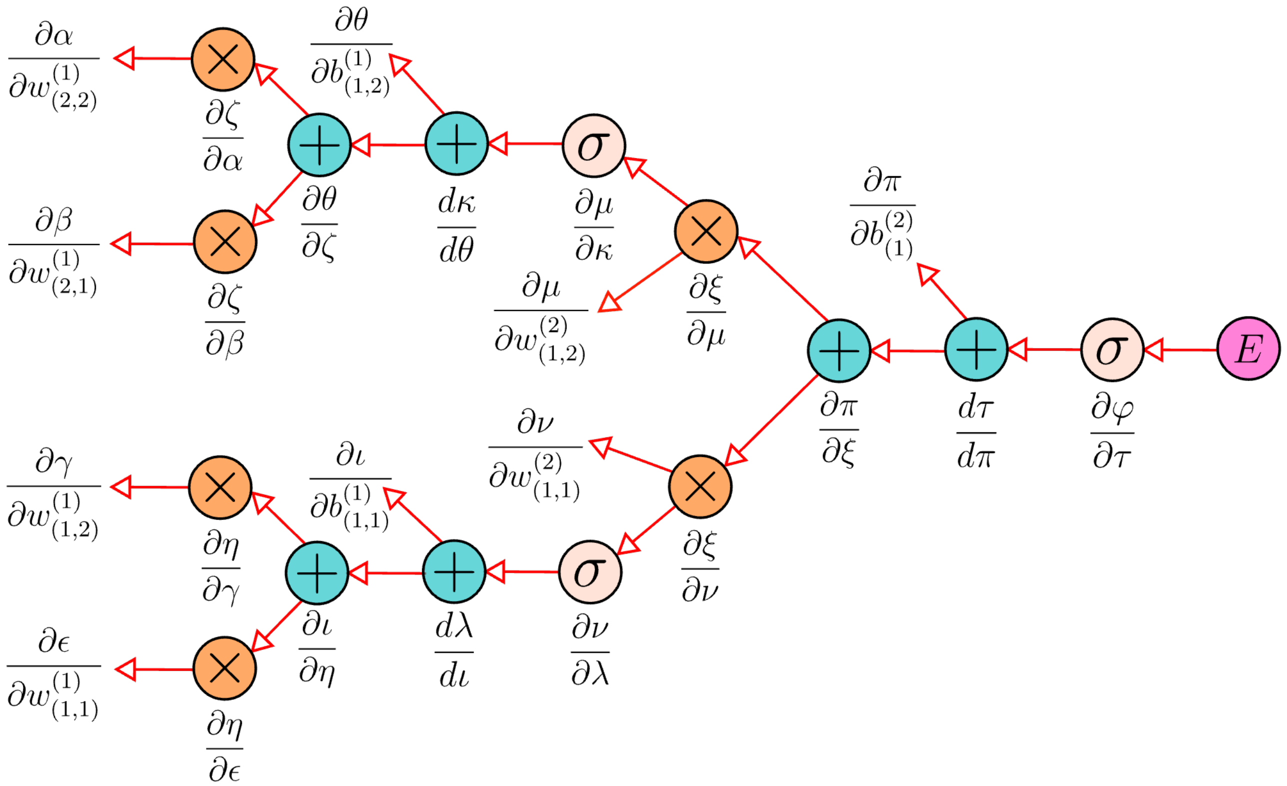
Automatic Differentiation

- AKA, *algorithmic differentiation*, *computational differentiation*, *Autodiff,* is a computational technique for efficiently and accurately evaluating derivatives of functions expressed as computer programs.
    - generate numerical derivative evaluations rather than derivative
    - build up data structures to represent derivative computations, and then can simply execute the expression to compute the derivative
    - efficient and optimizes derivative computation
    - “autograd” is the name of a particular package for “autodiff”



How Autodiff works for NN

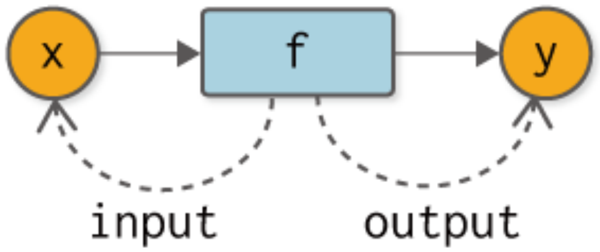
- Autodiff facilitates NN training by break down complex functions into simpler ones to compute derivatives efficiently
  - construct a computational graph
  - leverage the chain rule to compute derivatives efficiently
    - ❖ a composite function *f(x)=h(g(x)),* the derivative of f with respect to x is *df/dx=dh/dg dg/dx*
  - during backpropagation, compute derivatives through accumulation of values during code execution

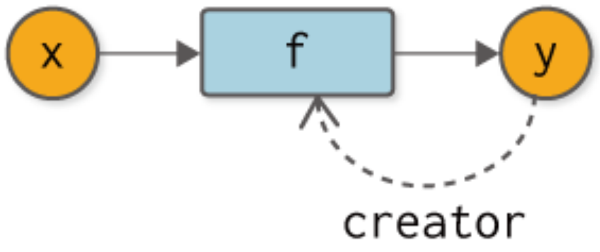

$\frac{\partial \alpha}{\partial w^{(1)}_{(2,2)}}$
$\frac{\partial \theta}{\partial b^{(1)}_{(1,2)}}$
$\frac{\partial \zeta}{\partial \alpha}$
$\frac{\partial \theta}{\partial \zeta}$
$\frac{d\kappa}{d\theta}$
σ
$\frac{\partial \mu}{\partial \kappa}$
$\frac{\partial \beta}{\partial w^{(1)}_{(2,1)}}$
$\frac{\partial \zeta}{\partial \beta}$
$\frac{\partial \mu}{\partial w^{(2)}_{(1,2)}}$
$\frac{\partial \xi}{\partial \mu}$
$\frac{\partial \pi}{\partial b^{(2)}_{(1)}}$
$\frac{\partial \pi}{\partial \xi}$
$\frac{d\tau}{d\pi}$
σ
$\frac{\partial \varphi}{\partial \tau}$
E
$\frac{\partial \nu}{\partial w^{(2)}_{(1,1)}}$
$\frac{\partial \gamma}{\partial w^{(1)}_{(1,2)}}$
$\frac{\partial \iota}{\partial b^{(1)}_{(1,1)}}$
$\frac{\partial \eta}{\partial \gamma}$
σ
$\frac{\partial \xi}{\partial \nu}$
$\frac{\partial \iota}{\partial \eta}$
$\frac{d\lambda}{d\iota}$
$\frac{\partial \nu}{\partial \lambda}$
$\frac{\partial \epsilon}{\partial w^{(1)}_{(1,1)}}$
$\frac{\partial \eta}{\partial \epsilon}$

https://mehta-rohan.com/writings/blog_posts/autodiff.html

Computational Graph

- a conceptual representation to break down calculations into individual operations that are easier to analyze and manipulate
- **Nodes**: each node represents an operation or a variable.
- **Edges**: directed arrows connecting nodes, indicating the flow of data
  - ➢ represent the dependencies between operations, specifying which operations must be completed before others can begin.



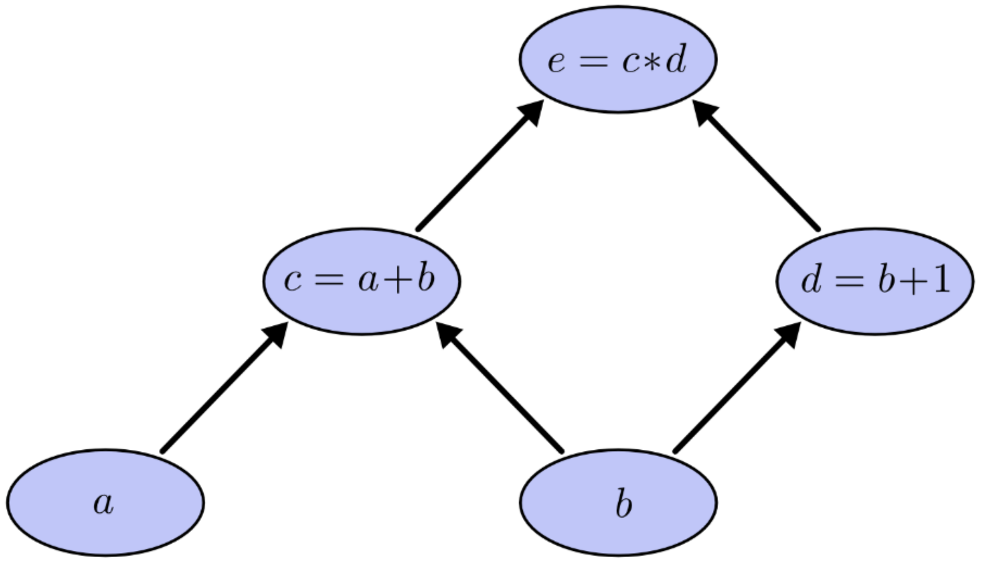


e=(a + b) * (b + 1)

c=(a + b)

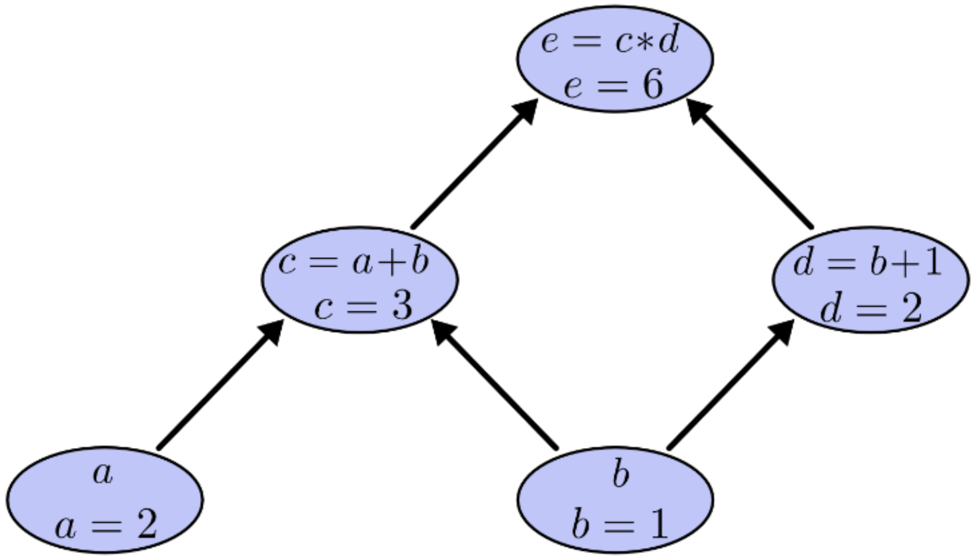
d=(b + 1)

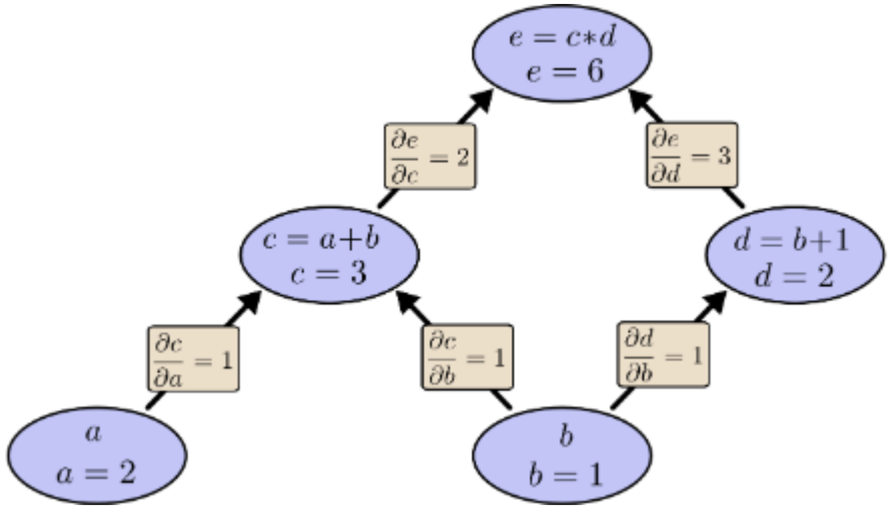
e=(c*d)



The role of Computational Graph

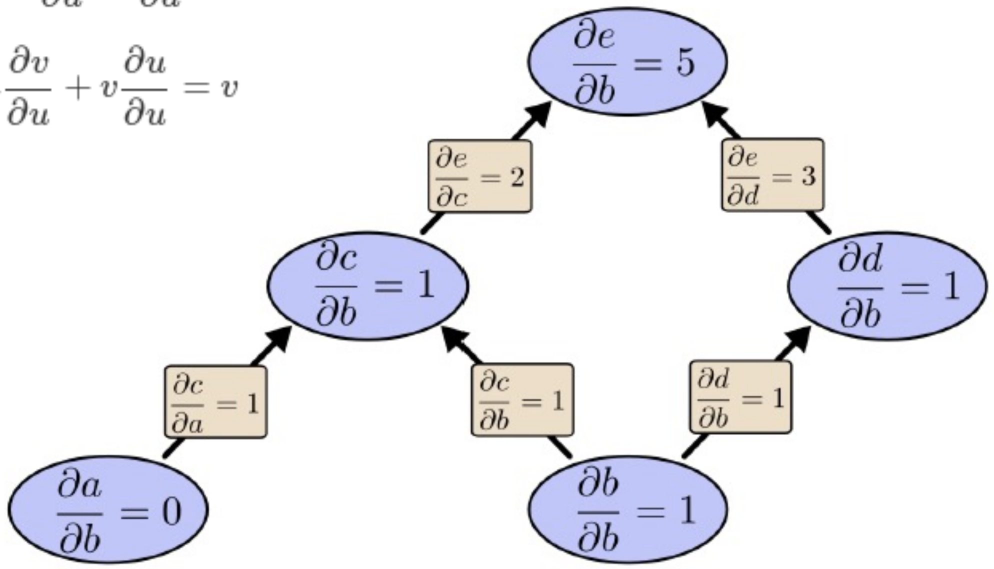
- can easily determine which partial-derivative-factors must be multiplied and for which paths the products must be added

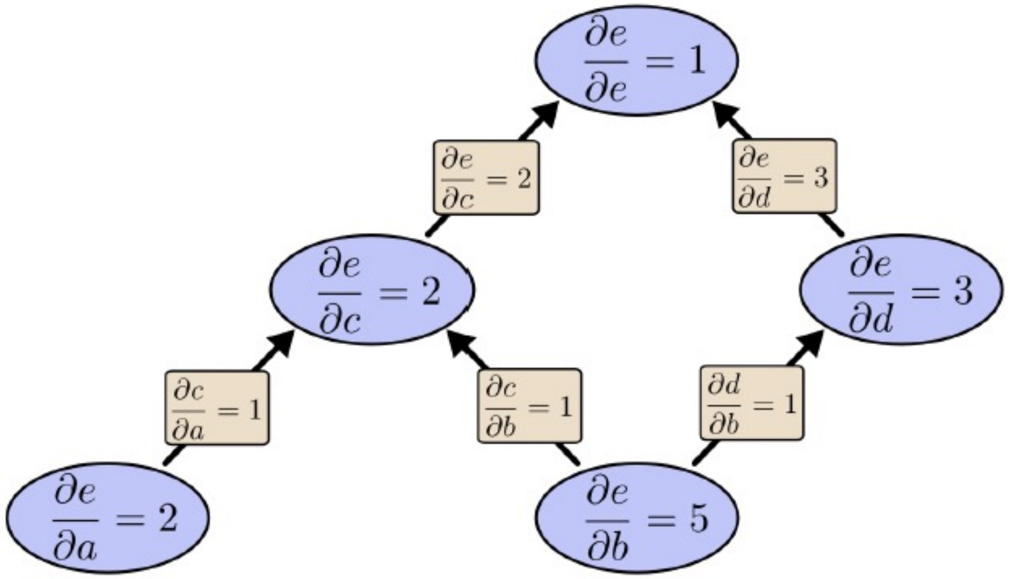



$e = c*d$
$e = 6$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$c = a+b$
$c = 3$
$d = b+1$
$d = 2$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
a
$a = 2$
b
$b = 1$

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$

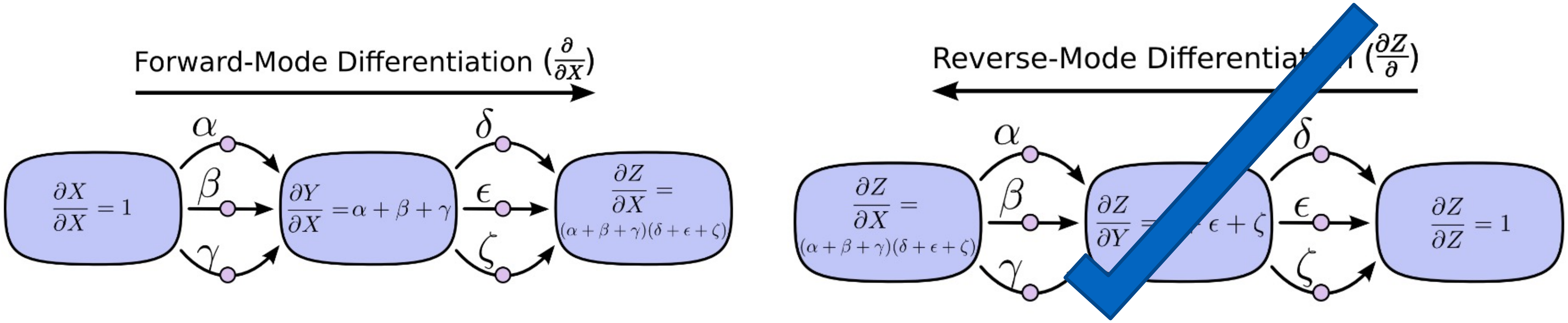
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$


$\frac{\partial e}{\partial b} = 5$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial a}{\partial b} = 0$
$\frac{\partial b}{\partial b} = 1$


$\frac{\partial e}{\partial e} = 1$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial e}{\partial a} = 2$
$\frac{\partial e}{\partial b} = 5$

Two Modes

- Two primary modes: forward mode and reverse mode differentiation
  - forward-mode starts at an input to the graph and moves towards the end, gives us the derivatives of all outputs with respect to one input

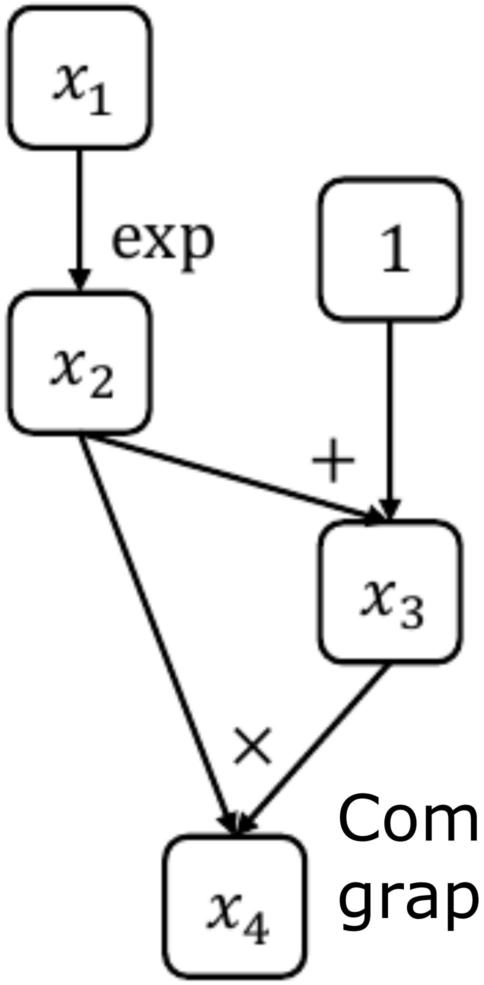

Forward-Mode Differentiation ($\frac{\partial}{\partial X}$)
$\frac{\partial X}{\partial X} = 1$
α
β
γ
$\frac{\partial Y}{\partial X} = \alpha + \beta + \gamma$
δ
ϵ
ζ
$\frac{\partial Z}{\partial X} = (\alpha+\beta+\gamma)(\delta+\epsilon+\zeta)$
Reverse-Mode Differentiat ($\frac{\partial Z}{\partial}$)
$\frac{\partial Z}{\partial X} = (\alpha+\beta+\gamma)(\delta+\epsilon+\zeta)$
α
β
γ
$\frac{\partial Z}{\partial Y} =$ $\epsilon + \zeta$
δ
ϵ
ζ
$\frac{\partial Z}{\partial Z} = 1$

  - reverse-mode starts at an output of the graph and moves towards the beginning, gives the derivatives of one output with respect to all inputs

https://colah.github.io/posts/2015-08-Backprop/

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```

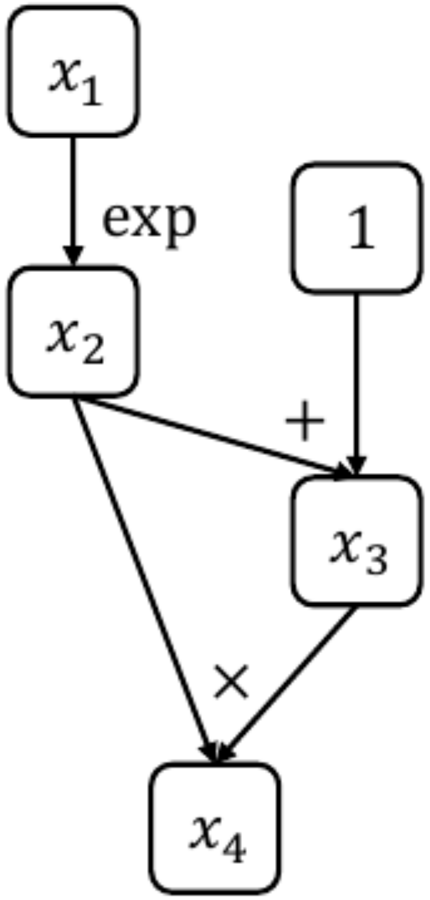


Computational graph

#Sum the partial derivative from output edges
#Compute gradients of the operation with respect to its inputs
# Accumulate gradients for each input node

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```



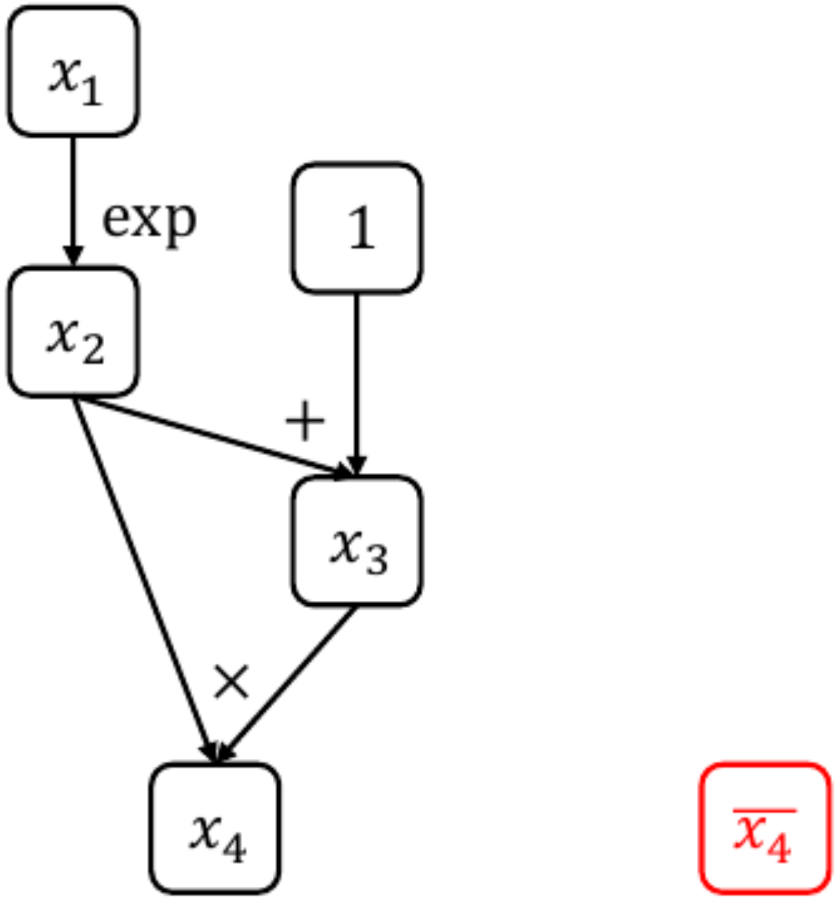
node_to_grad:
  x_4: $\overline{x_4}$

The adjoint of a node x denoted as $\bar{x}$ or $\frac{dL}{dx}$, is the derivative of the loss L with respect to x

$$\overline{x_4} = 1$$

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for
input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```

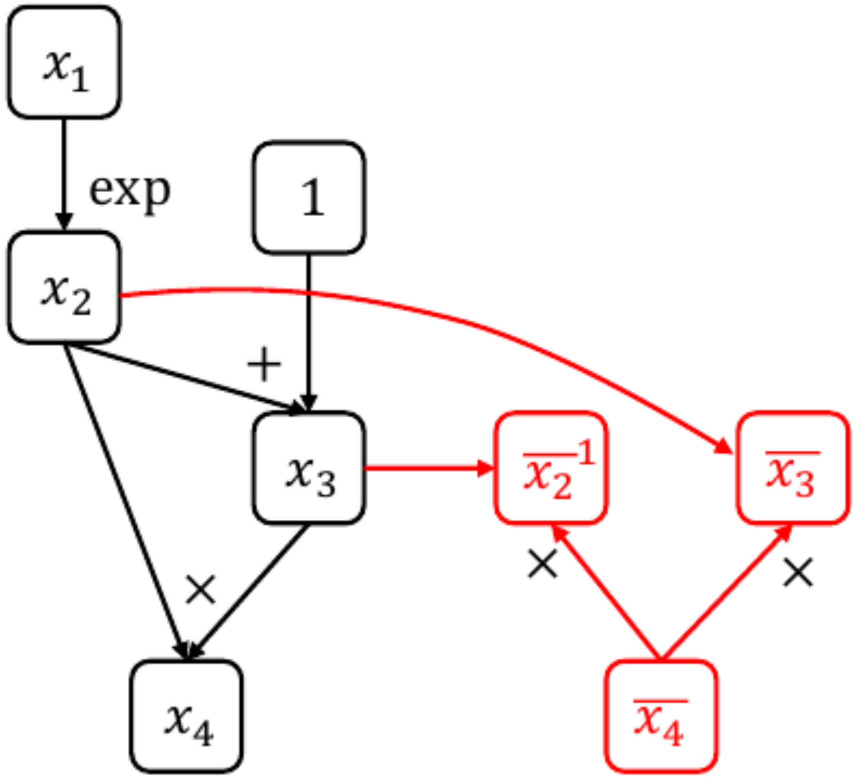


node_to_grad:
  x_4: $\overline{x_4}$

x_4	x_3	x_2	x_1
⇧			

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for
input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```



node_to_grad:
x_4 : $\overline{x_4}$

x_4	x_3	x_2	x_1

⇧

$$x_4 = x_2 \times x_3$$

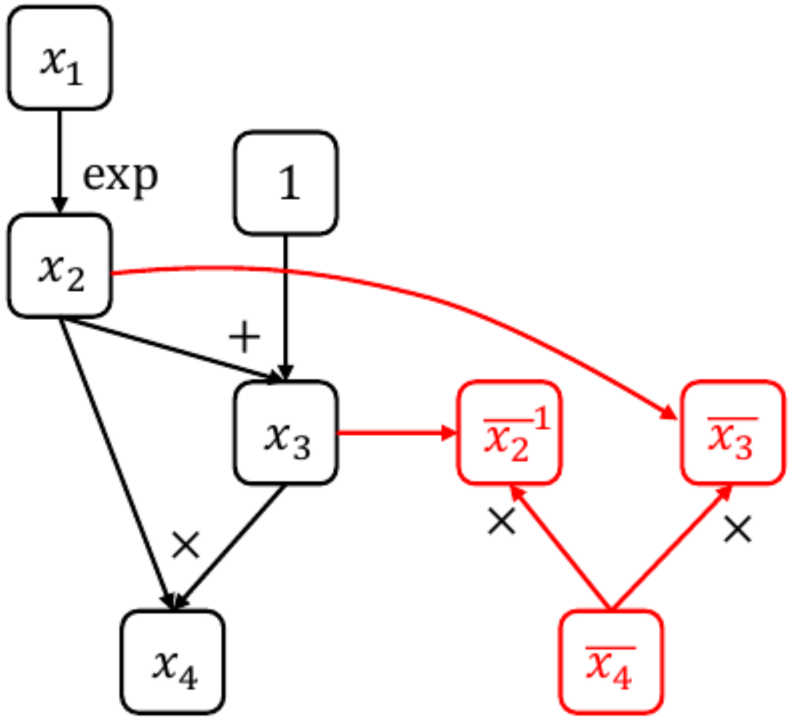
x_2 impact x_4 in two path,
here just one path

$$\overline{x_3} = \overline{x_4} \times x_2$$

$$\overline{x_2}^1 = \overline{x_4} \times x_3$$

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```



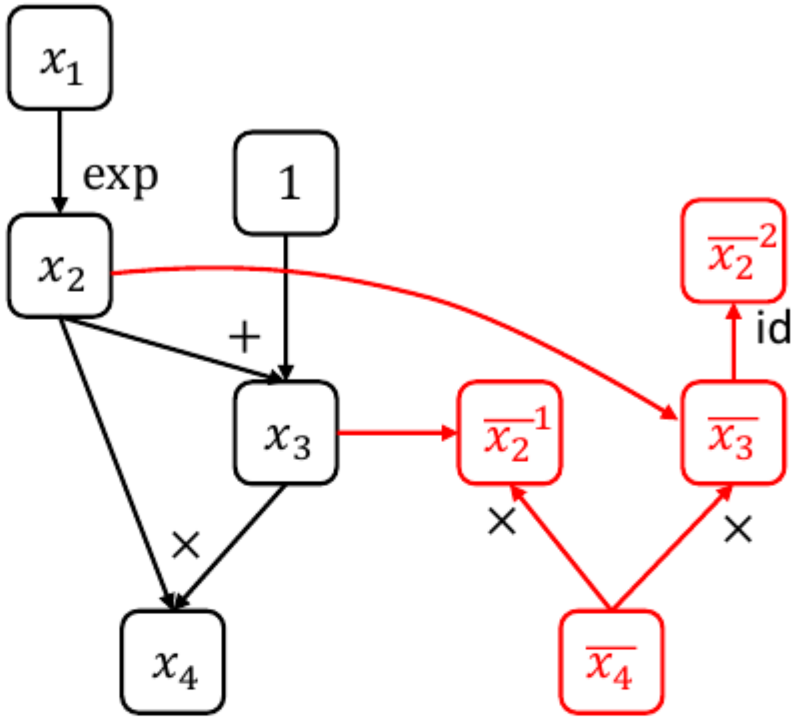
node_to_grad:
x_4: $\overline{x_4}$
x_3: $\overline{x_3}$
x_2: $\overline{x_2}^1$

x_4	x_3	x_2	x_1

⇧

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for
input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```



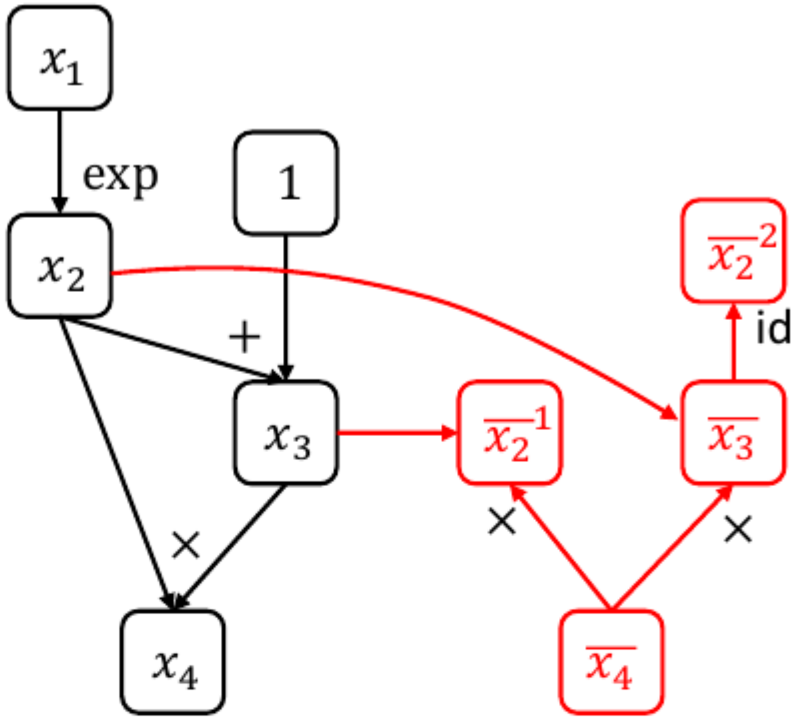
node_to_grad:
x_4: $\overline{x_4}$
x_3: $\overline{x_3}$
x_2: $\overline{x_2}^1$

x_4	x_3	x_2	x_1
	⇧		

$$x_3 = x_2 + 1 \Rightarrow \overline{x_2}^2 = \overline{x_3}\frac{\partial x_3}{\partial x_2} = \overline{x_3}$$

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```

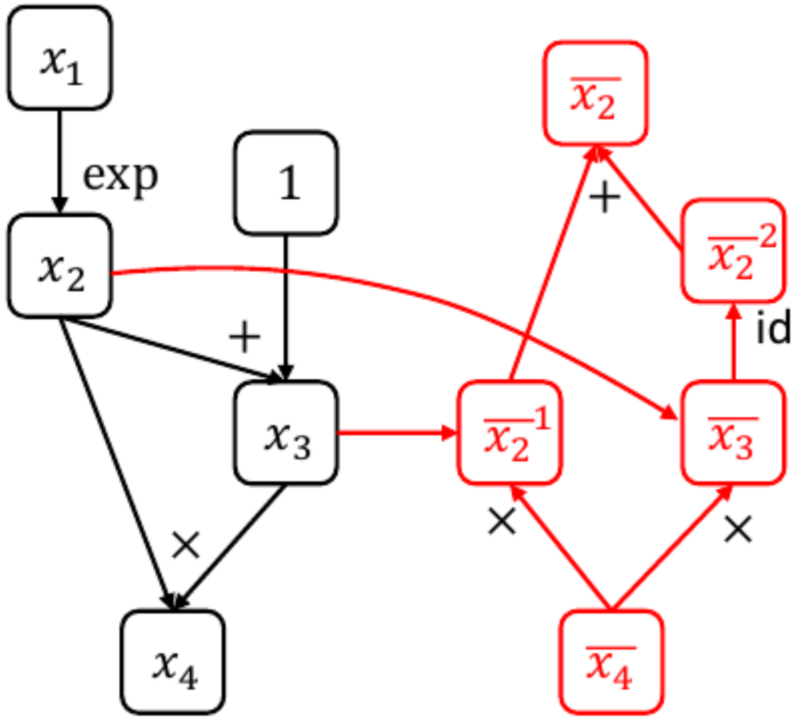


node_to_grad:
x_4: $\overline{x_4}$
x_3: $\overline{x_3}$
x_2: $\overline{x_2}^1$, $\overline{x_2}^2$

x_4	x_3	x_2	x_1
	⇧		

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```

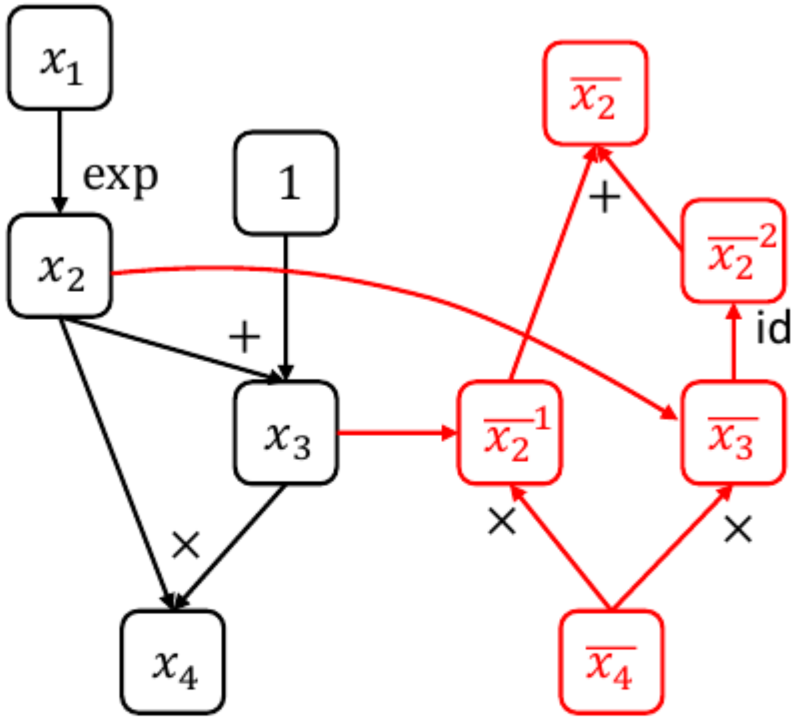


node_to_grad:
x_4: $\overline{x_4}$
x_3: $\overline{x_3}$
x_2: $\overline{x_2}^1$, $\overline{x_2}^2$

x_4	x_3	x_2	x_1
		⇧	

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for
input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```



node_to_grad:
x_4: $\overline{x_4}$
x_3: $\overline{x_3}$
x_2: $\overline{x_2}^1$, $\overline{x_2}^2$

x_4	x_3	x_2	x_1
		⇧	

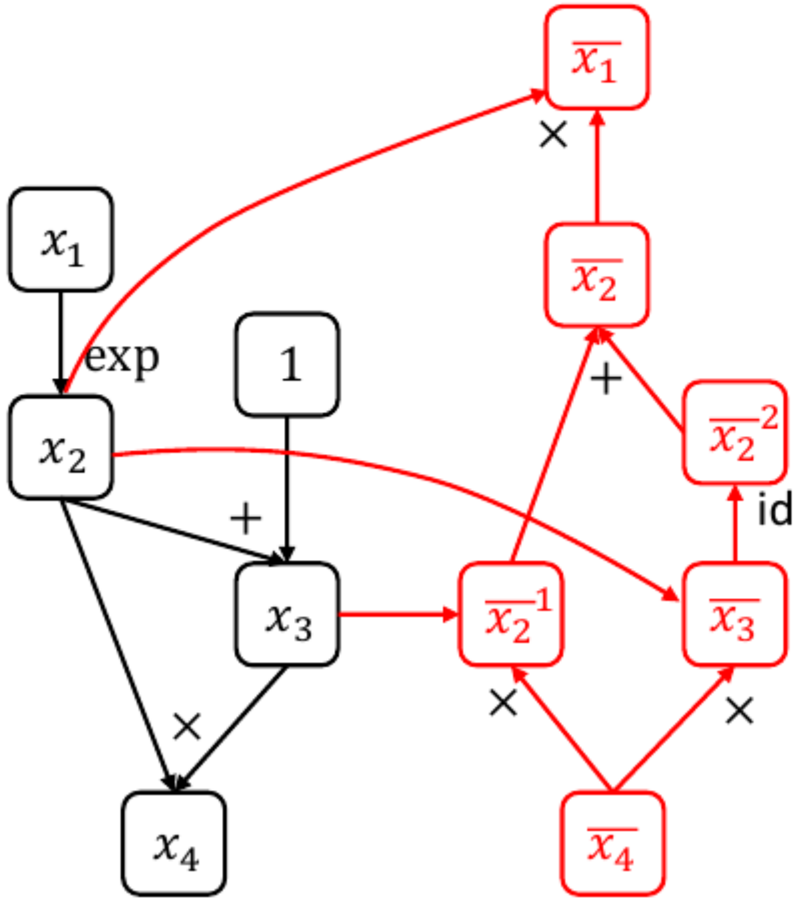
AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```

node_to_grad:
x_4: $\overline{x_4}$
x_3: $\overline{x_3}$
x_2: $\overline{x_2}^1$, $\overline{x_2}^2$

x_4	x_3	x_2	x_1

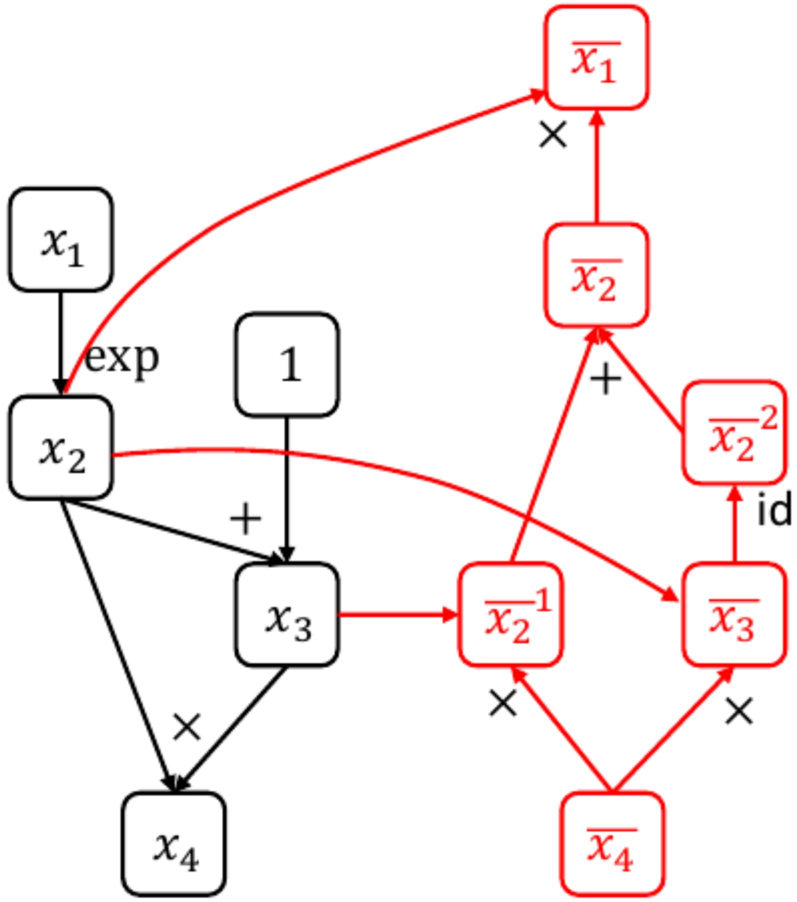
⇧



$$x_2 = \exp(x_1) \Longrightarrow \overline{x_1} = \overline{x_2}\exp(x_1)$$

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```

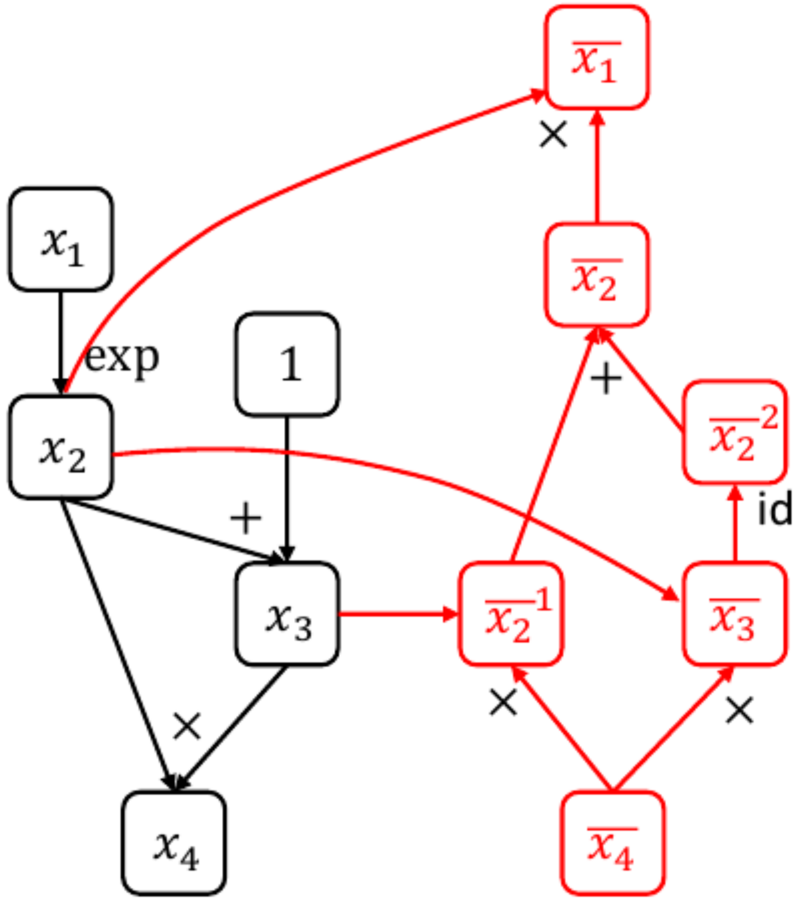


node_to_grad:
x_4: $\overline{x_4}$
x_3: $\overline{x_3}$
x_2: $\overline{x_2}^1$, $\overline{x_2}^2$
x_1: $\overline{x_1}$

x_4	x_3	x_2	x_1
		⇧	

AutoDiff Algorithm

```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for
input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```

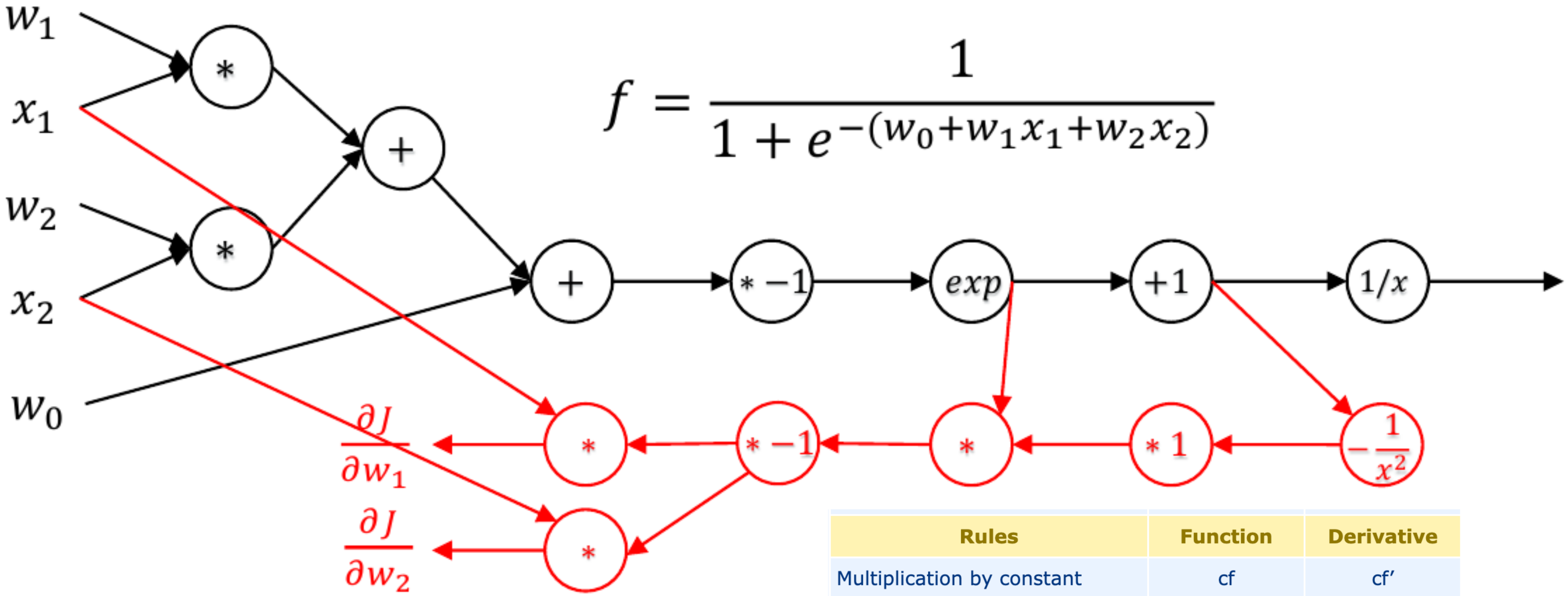


node_to_grad:
x_4: $\overline{x_4}$
x_3: $\overline{x_3}$
x_2: $\overline{x_2}^1$, $\overline{x_2}^2$
x_1: $\overline{x_1}$

x_4	x_3	x_2	x_1

⇧

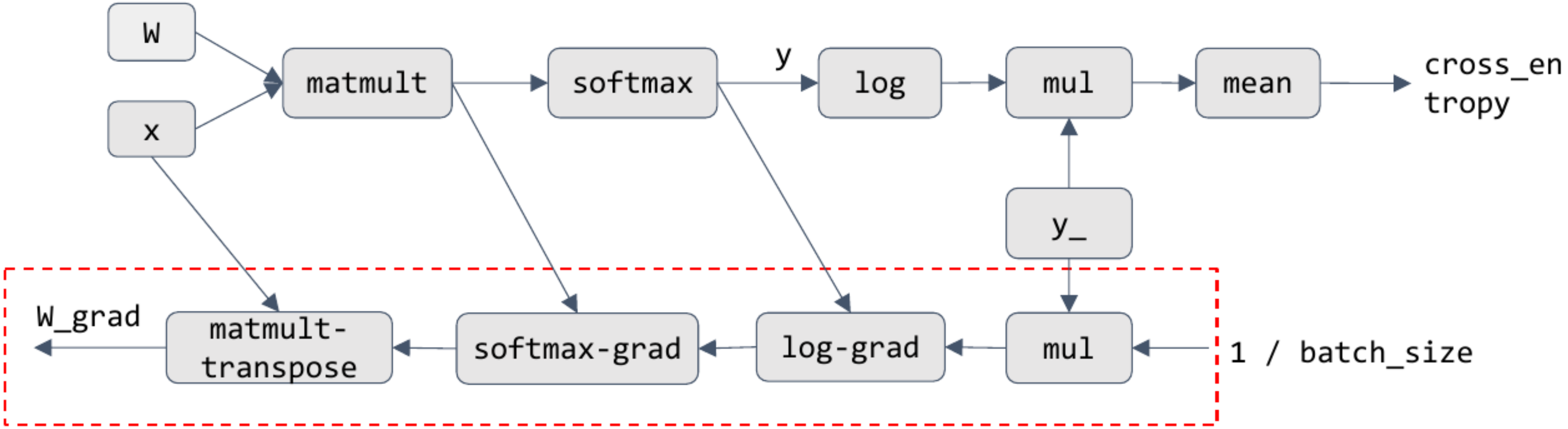
More complicated functions



$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

Rules	Function	Derivative
Multiplication by constant	cf	cf’
Power Rule	x^n	nx^{n-1}
Sum Rule	f + g	f’ + g’
Difference Rule	f - g	f’ − g’
Product Rule	fg	f g’ + f’ g
Quotient Rule	f/g	$\frac{f' g - g' f}{g^2}$
Reciprocal Rule	1/f	$-f'/f^2$

More complicated functions



Autograd in Pytorch

- torch.autograd is PyTorch’s automatic differentiation engine
- tensors have an attribute “requires_grad” that indicates whether gradient should be tracked
- computation graph is created dynamically during the forward pass
- call “backward()” on the output tensor to compute gradients

  e.g., “loss.backward()”
- gradients are accumulated in the .grad attribute, must be zeroed out before new gradients are computed

  e.g., “optimizer.zero_grad()”

Summary

- Automatic Differentiation is a technique to compute derivatives of functions
- AD has two modes:Forward Mode and Reverse Mode
  - Reverse Mode is Efficient for functions with many inputs and few outputs
- constructs a computation graph during the forward pass and computes gradients via backpropagation.
- autograd is an auto differentiation module in PyTorch
- call backward() on loss to compute gradients for all tensors in the computation graph.

Reference

- https://dlsys.cs.washington.edu/materials
- Automatic differentiation in machine learning: a survey

  https://arxiv.org/abs/1502.05767