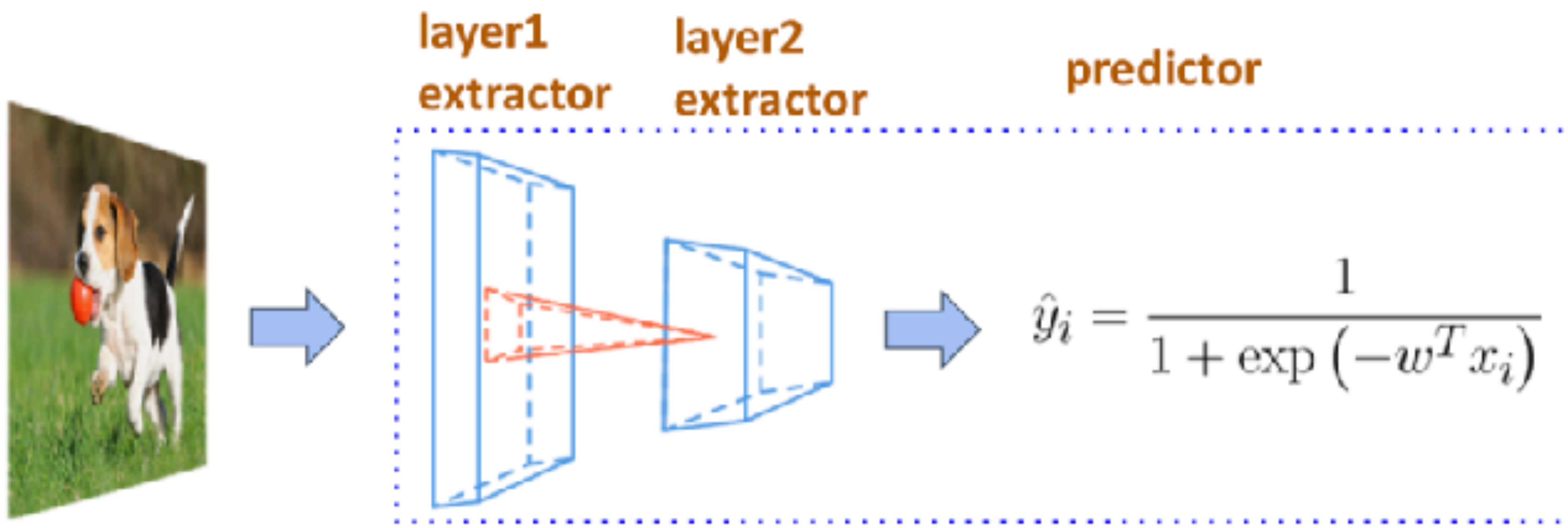
Machine Learning Systems

Lecture 8: Backpropagation and Automatic Differentiation

Pooyan Jamshidi

Model Training Overview



Objective $$L(w) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \lambda \|w\|^2$$

Training $$w \leftarrow w - \eta \nabla_w L(w)$$

Symbolic Differentiation

- Input formulae is a symbolic expression tree (computation graph).
- Implement differentiation rules, e.g., sum rule, product rule, chain rule

$$\frac{d(f+g)}{dx} = \frac{df}{dx} + \frac{dg}{dx} \qquad \frac{d(fg)}{dx} = \frac{df}{dx}g + f\frac{dg}{dx} \qquad \frac{d\big(h(x)\big)}{dx} = \frac{df\big(g(x)\big)}{dx} \cdot \frac{dg(x)}{x}$$

✗ For complicated functions, the resultant expression can be exponentially large.

✗ Wasteful to keep around intermediate symbolic expressions if we only need a numeric value of the gradient in the end

✗ Prone to error

Numerical Differentiation

- We can approximate the gradient using

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{e_i}) - f(\boldsymbol{x})}{h}$$

$$f(W, x) = \quad W \quad \cdot \quad x$$

$$\begin{bmatrix} -0.8 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ -0.2 \end{bmatrix}$$

Numerical Differentiation

- We can approximate the gradient using

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{e}_i) - f(\boldsymbol{x})}{h}$$

$$f(W, x) = \quad W \quad \cdot \quad x$$

$$\begin{bmatrix} -0.8 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ -0.2 \end{bmatrix}$$

$$f(W, x) = \quad W \quad \cdot \quad x$$

$$\begin{bmatrix} -0.8 + \varepsilon & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ -0.2 \end{bmatrix}$$

Numerical Differentiation

- We can approximate the gradient using

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{e_i}) - f(\boldsymbol{x})}{h}$$

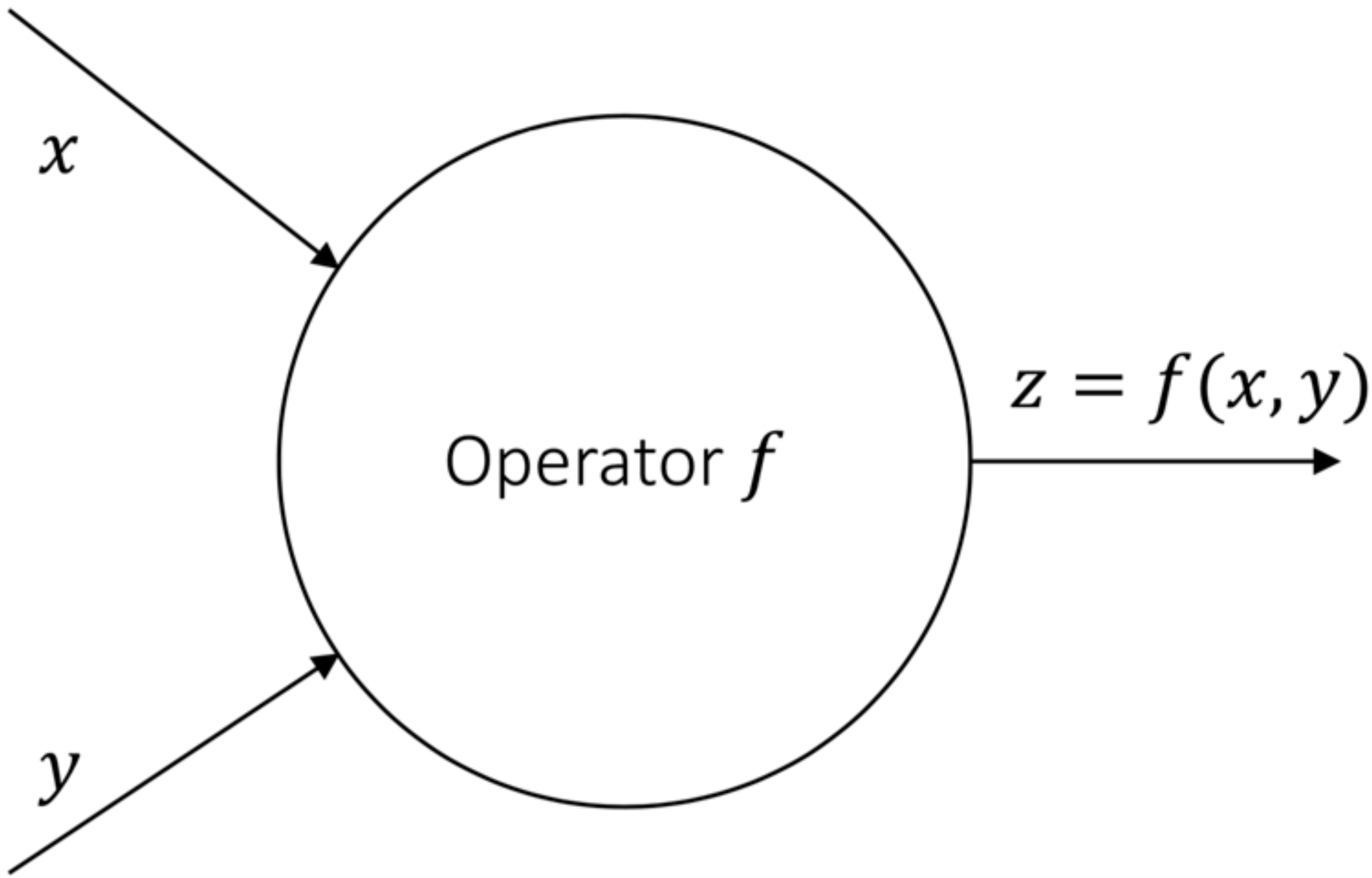
- Reduce the truncation error by using center difference

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{e_i}) - f(\boldsymbol{x} - h\boldsymbol{e_i})}{2h}$$

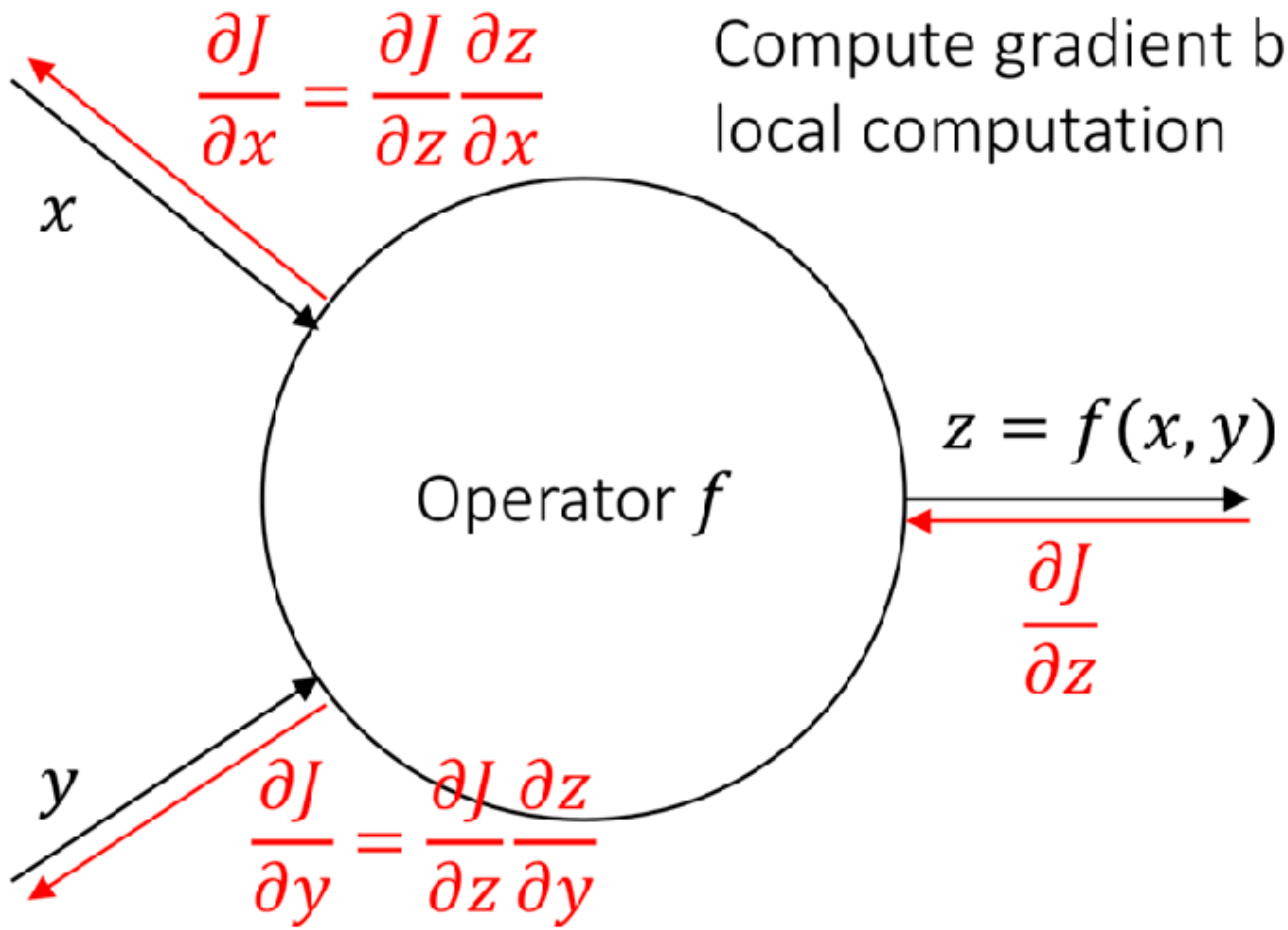
✗ Bad: rounding error, and slow to compute

✓ A powerful tool to check the correctness of implementation, usually use h = 1e-6.

Backpropagation



Backpropagation



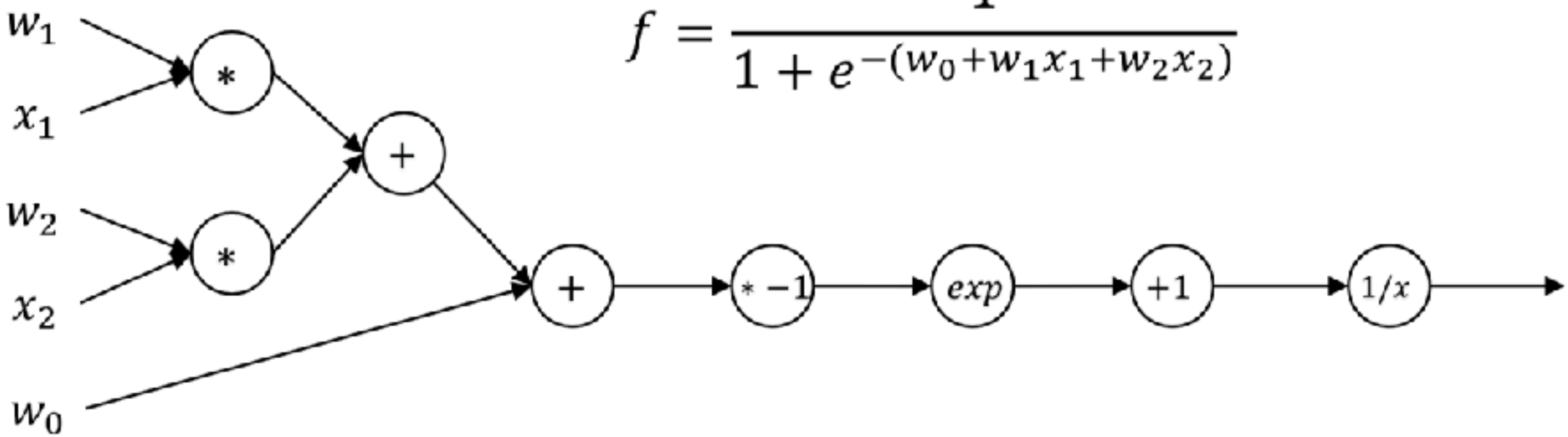
Compute gradient becomes local computation

Backpropagation simple example

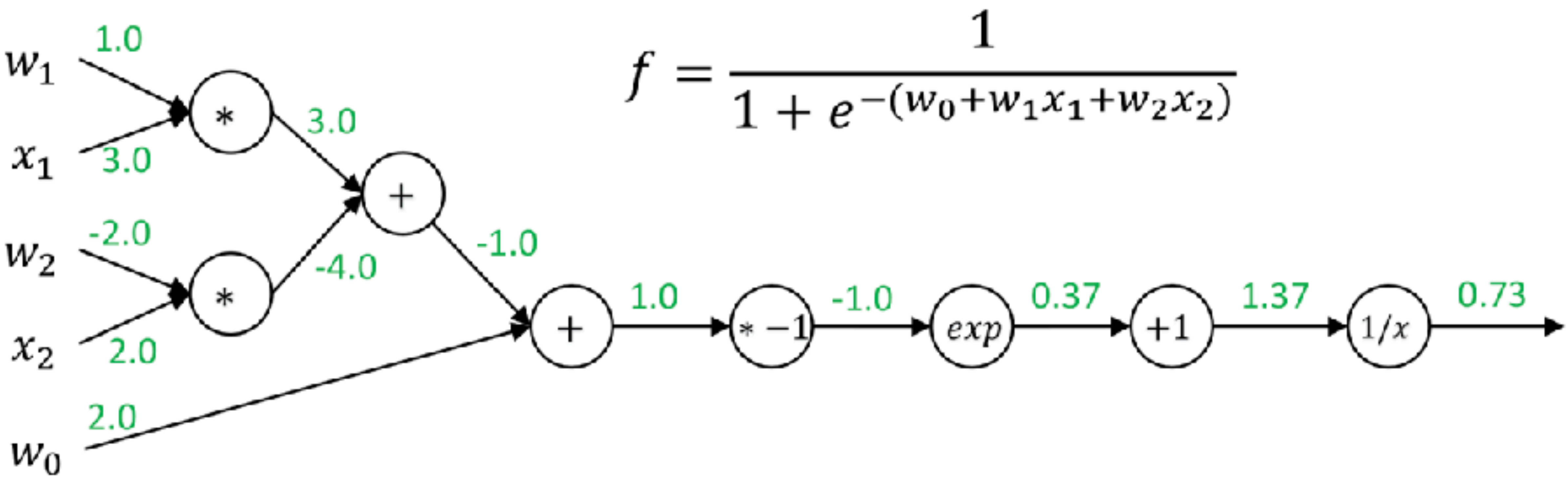
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

Backpropagation simple example

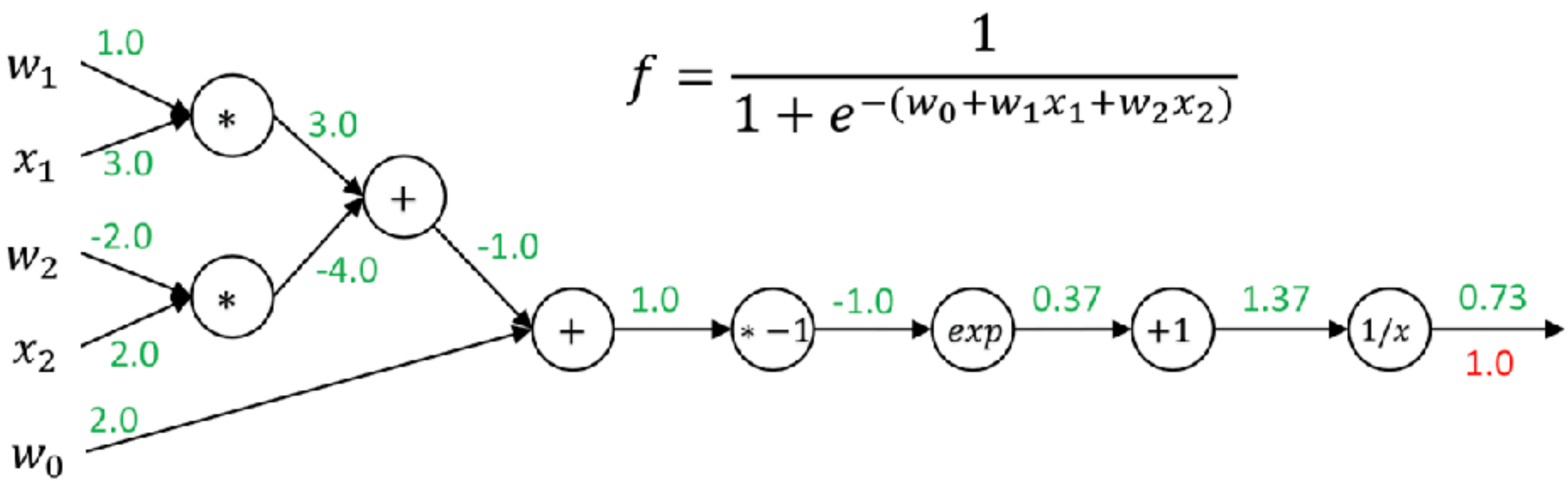
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



Backpropagation simple example

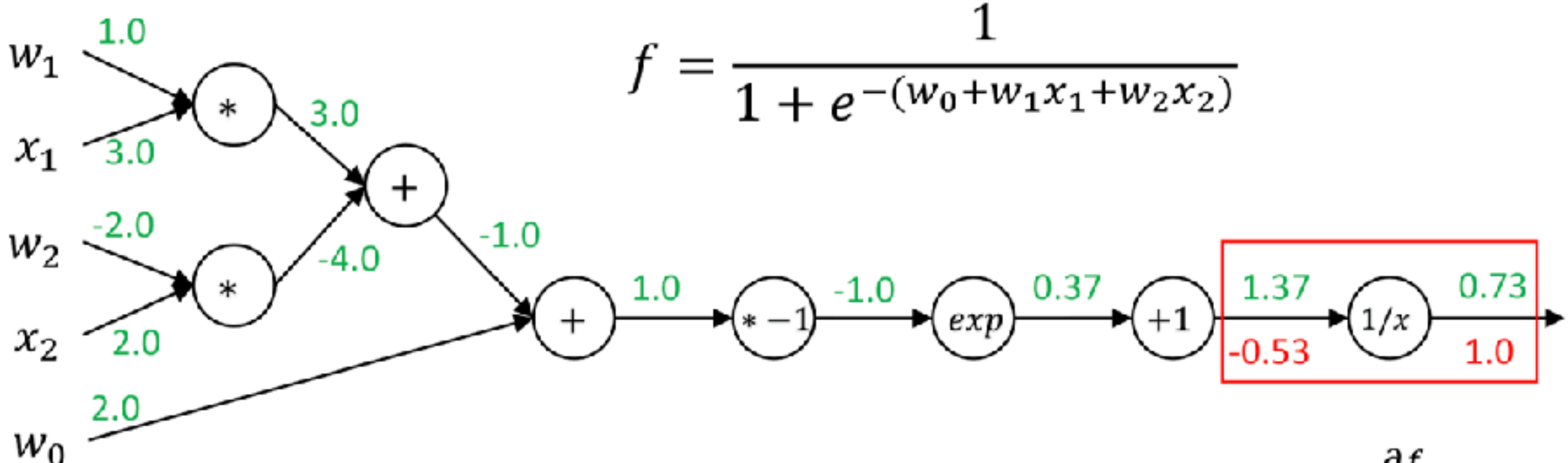


Backpropagation simple example



Backpropagation simple example

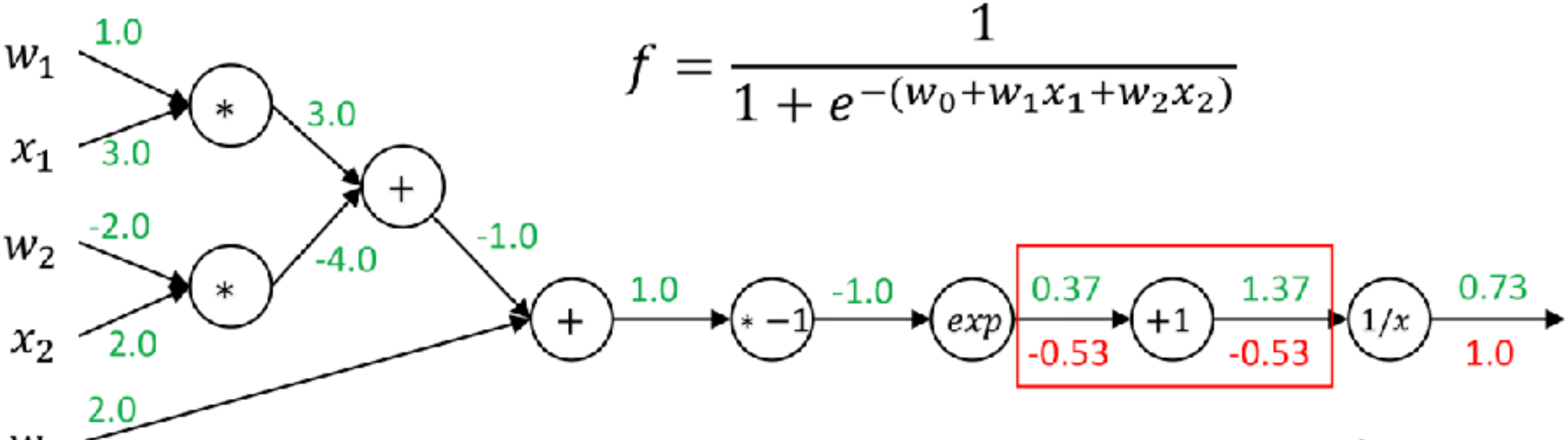
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



$$f(x) = 1/x \quad \rightarrow \quad \frac{\partial f}{\partial x} = -1/x^2$$

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial f}\frac{\partial f}{\partial x} = -1/x^2$$

Backpropagation simple example

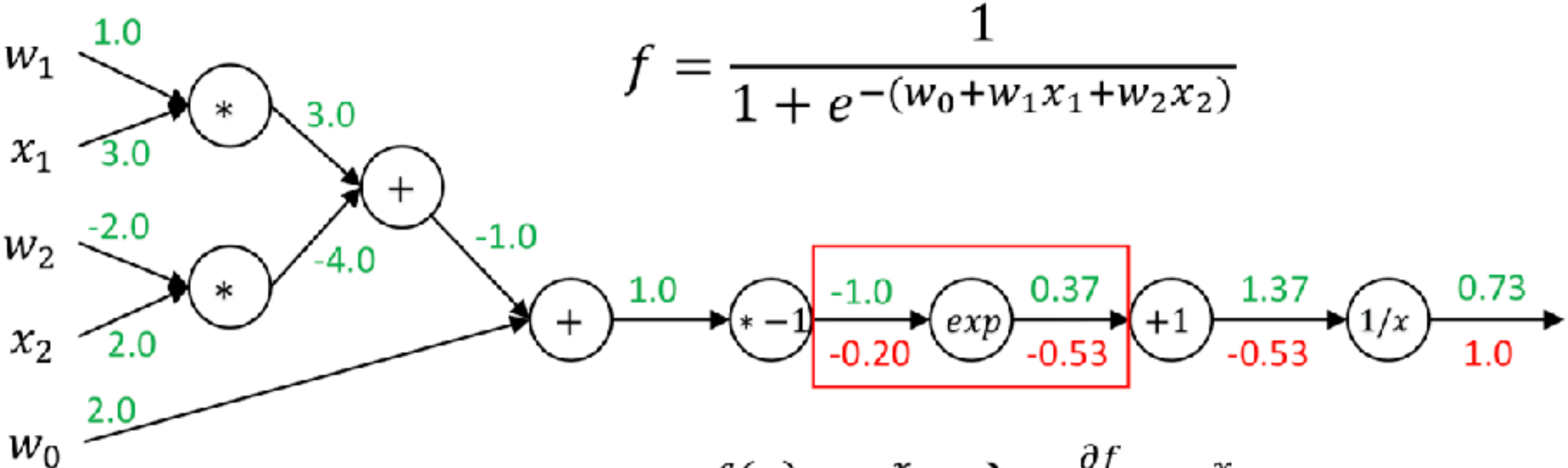


$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

$$f(x) = x + 1 \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1$$

Backpropagation simple example

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

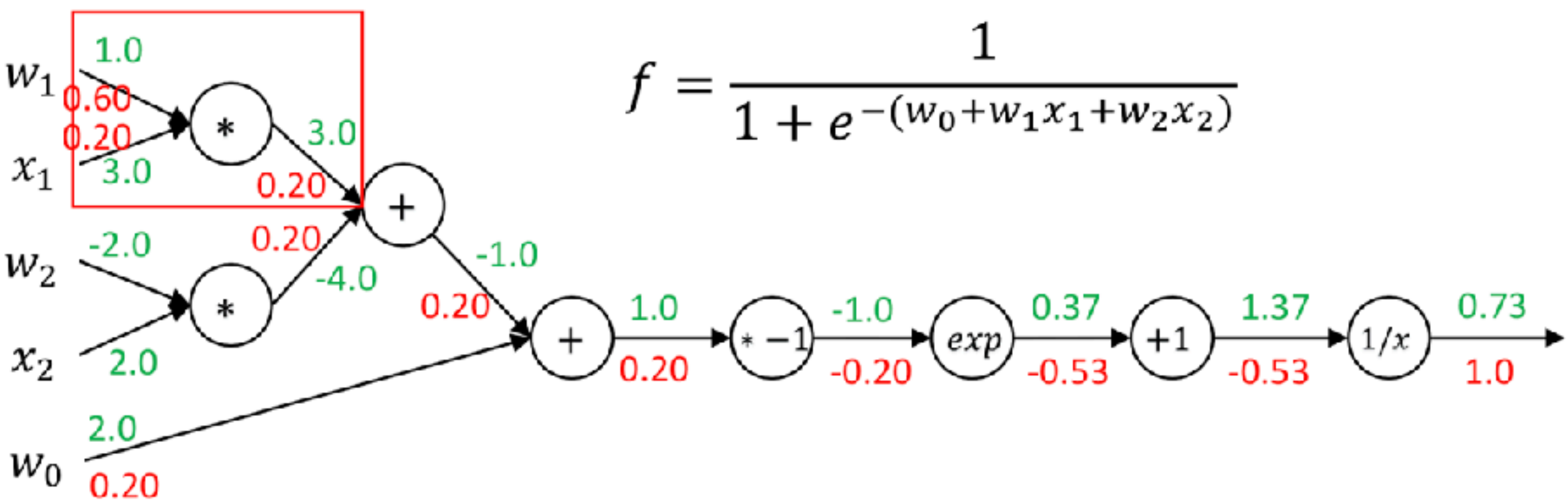


$$f(x) = e^x \quad \rightarrow \quad \frac{\partial f}{\partial x} = e^x$$

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial f}\frac{\partial f}{\partial x} = \frac{\partial J}{\partial f} \cdot e^x$$

Backpropagation simple example

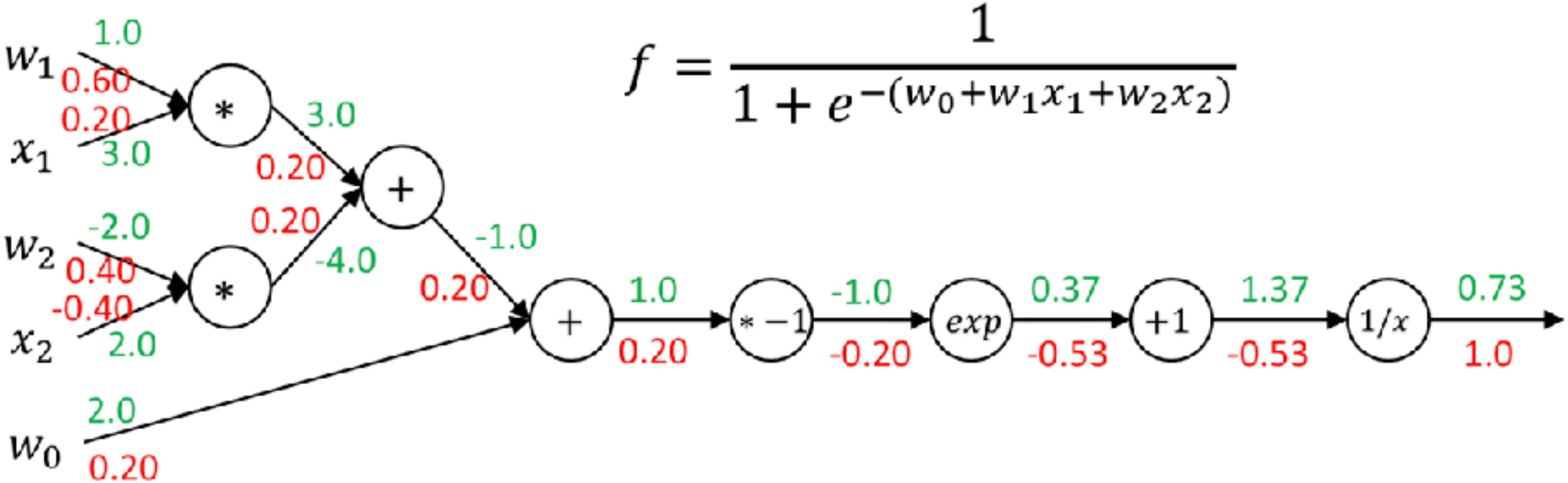
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



$$f(x, w) = xw \quad \rightarrow \quad \frac{\partial f}{\partial x} = w, \ \frac{\partial f}{\partial w} = x$$

Backpropagation simple example

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



Any problem? Can we do better?

Problems of backpropagation

- You always need to keep intermediate data in the memory during the forward pass in case it will be used in the backpropagation.
- Lack of flexibility, e.g., compute the gradient of gradient.

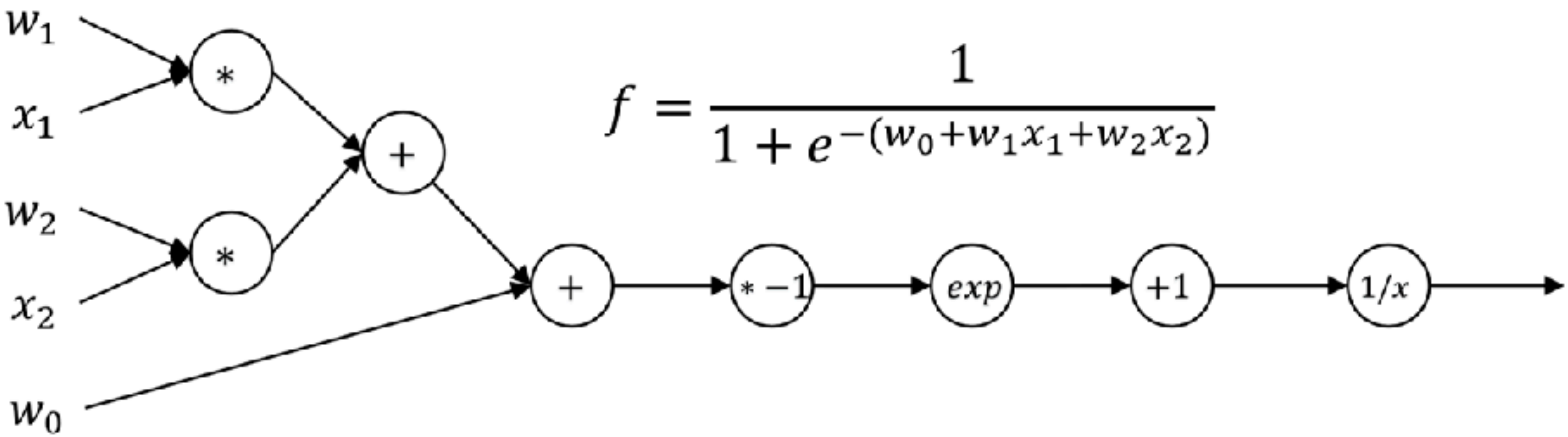
Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

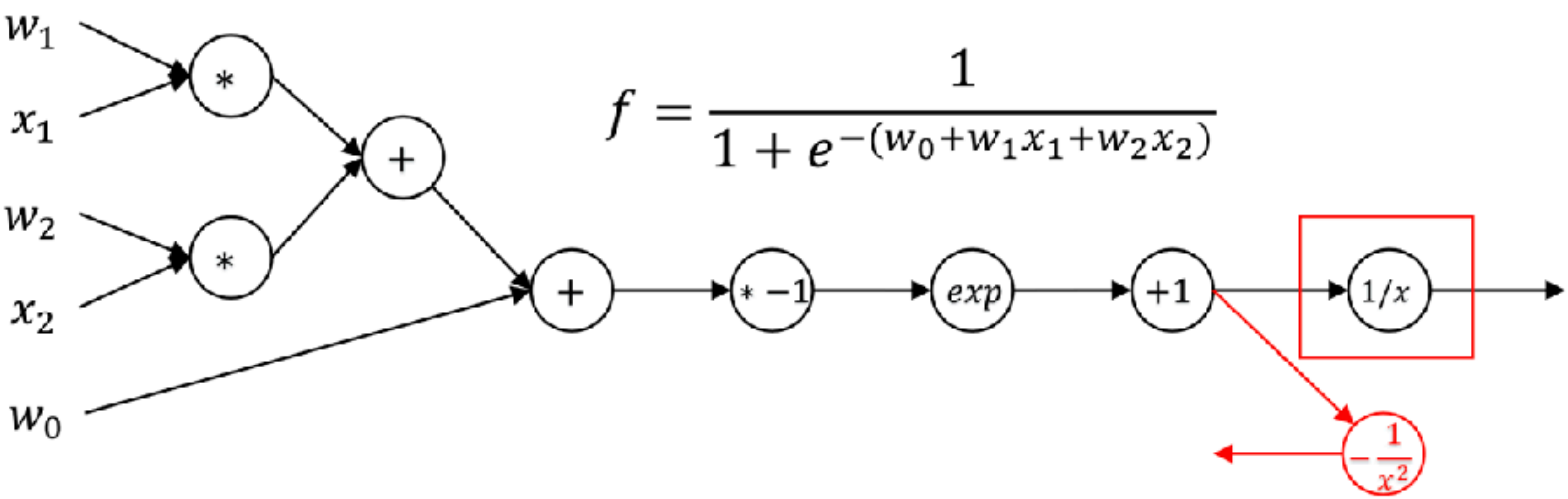
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

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

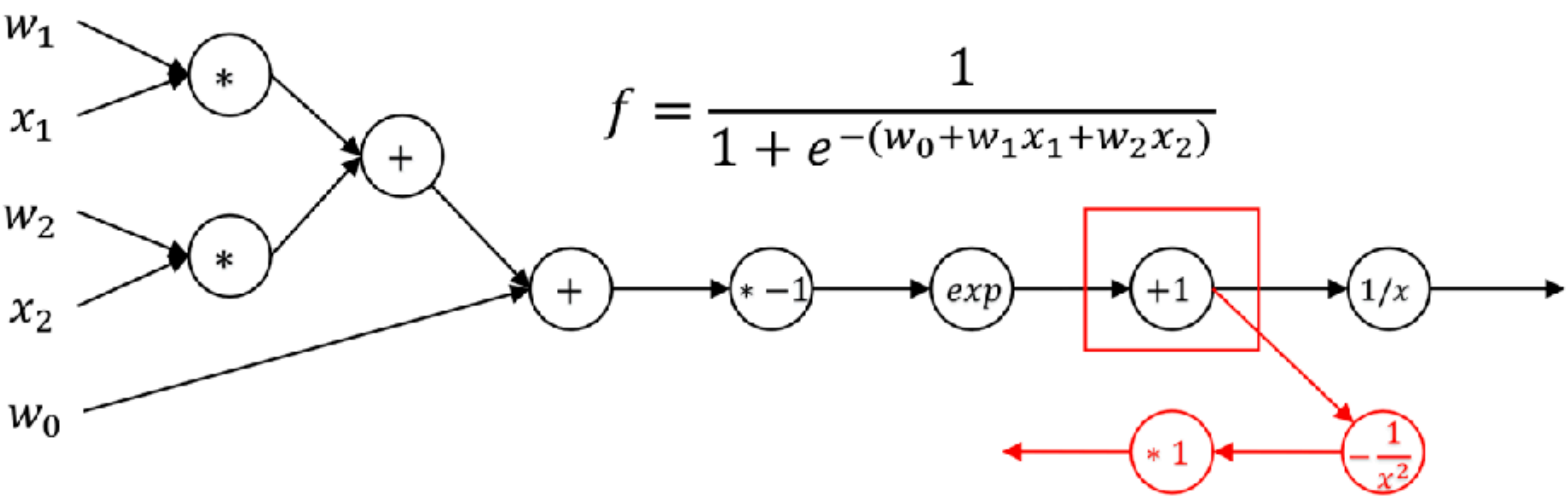


$$f(x) = 1/x \quad \rightarrow \quad \frac{\partial f}{\partial x} = -1/x^2$$

Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

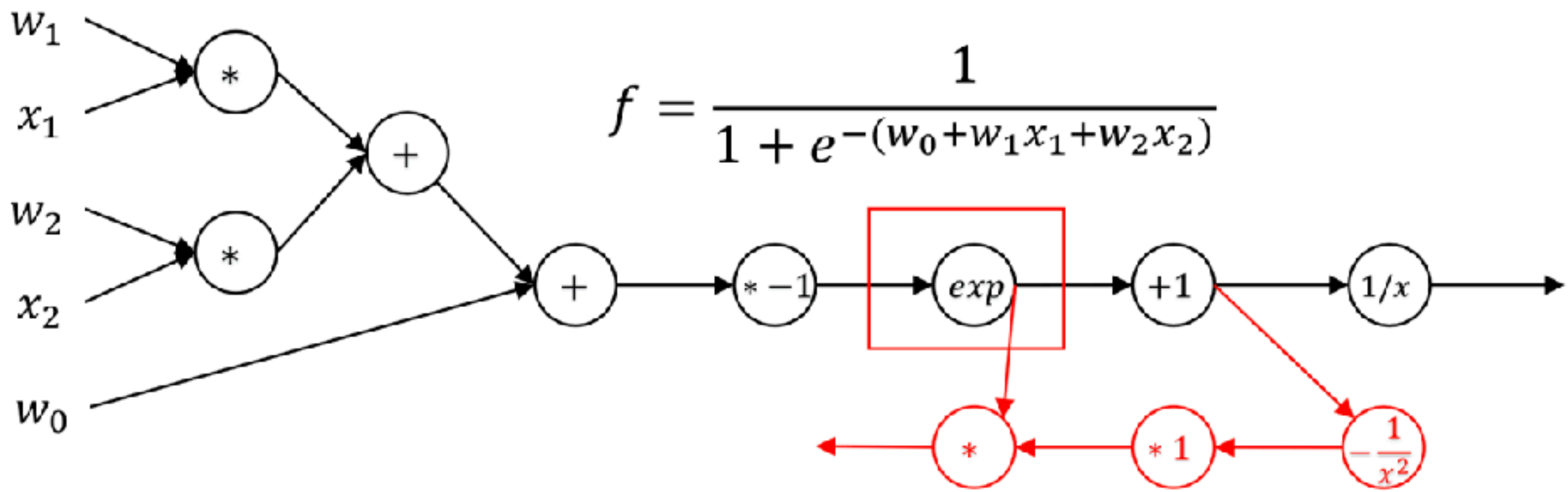
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



$$f(x) = x + 1 \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1$$

Automatic Differentiation (autodiff)

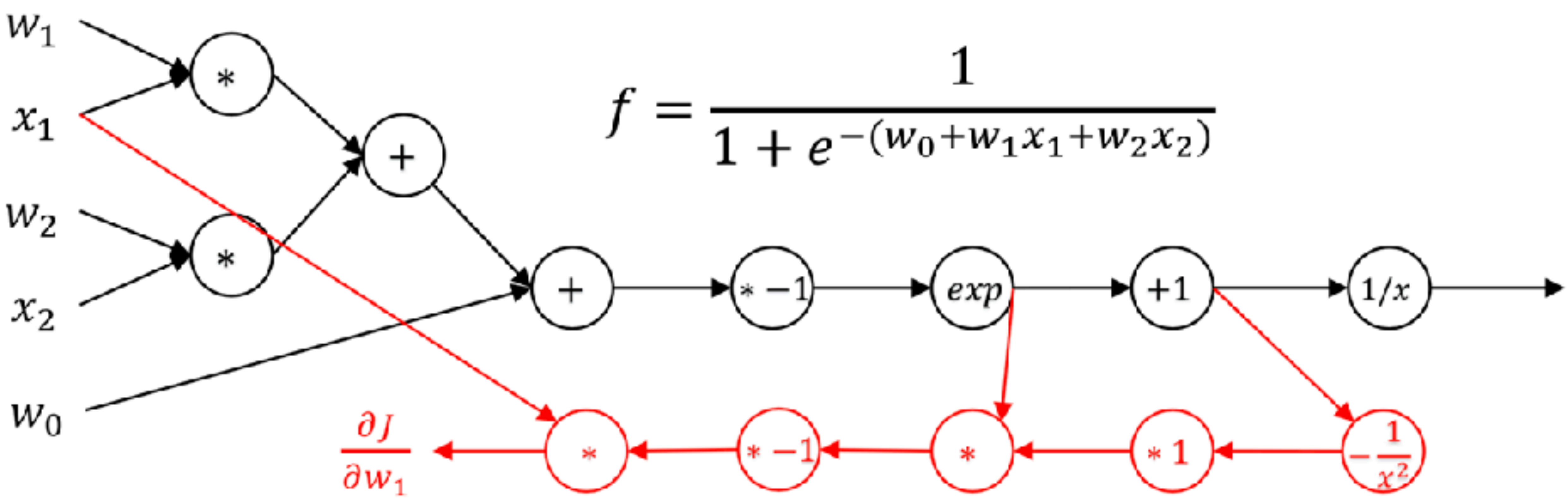
- Create computation graph for gradient computation



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

Automatic Differentiation (autodiff)

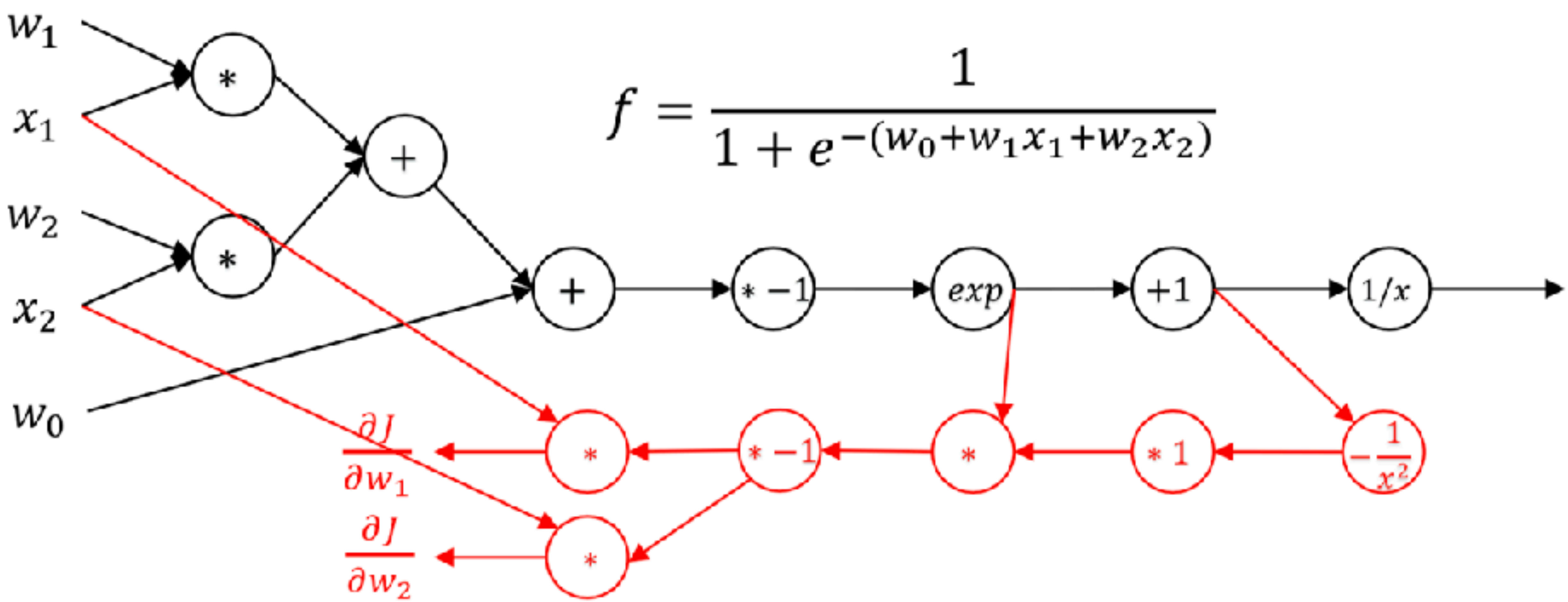
- Create computation graph for gradient computation



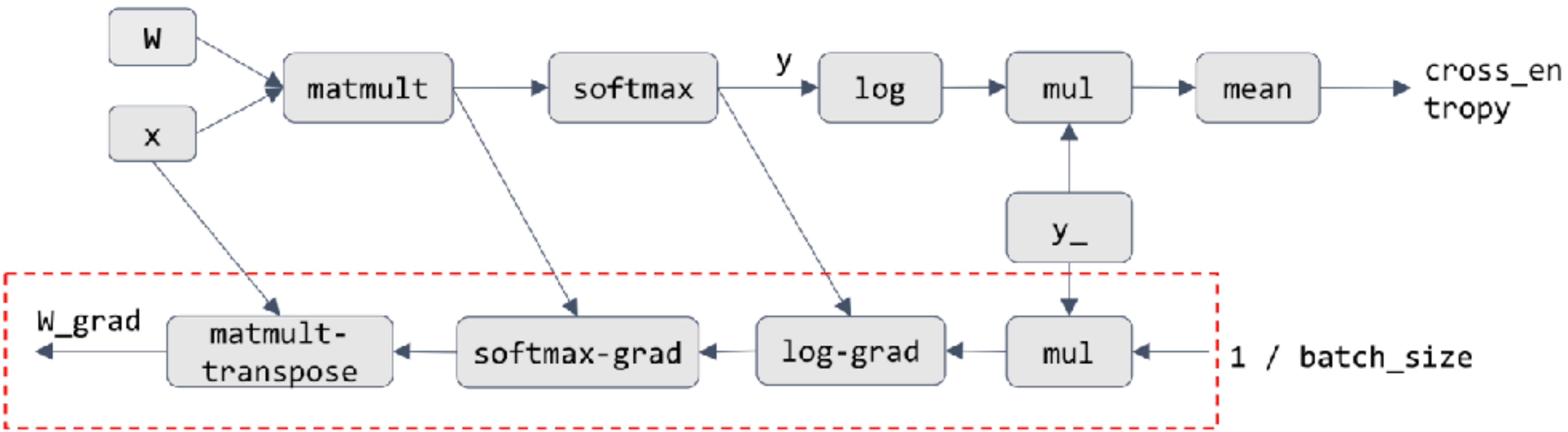
$$f(x,w) = xw \rightarrow \frac{\partial f}{\partial w} = x$$

Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

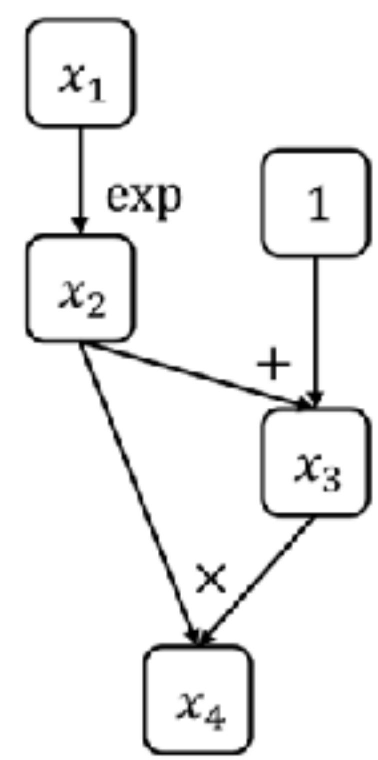


AutoDiff Algorithm



AutoDiff Algorithm

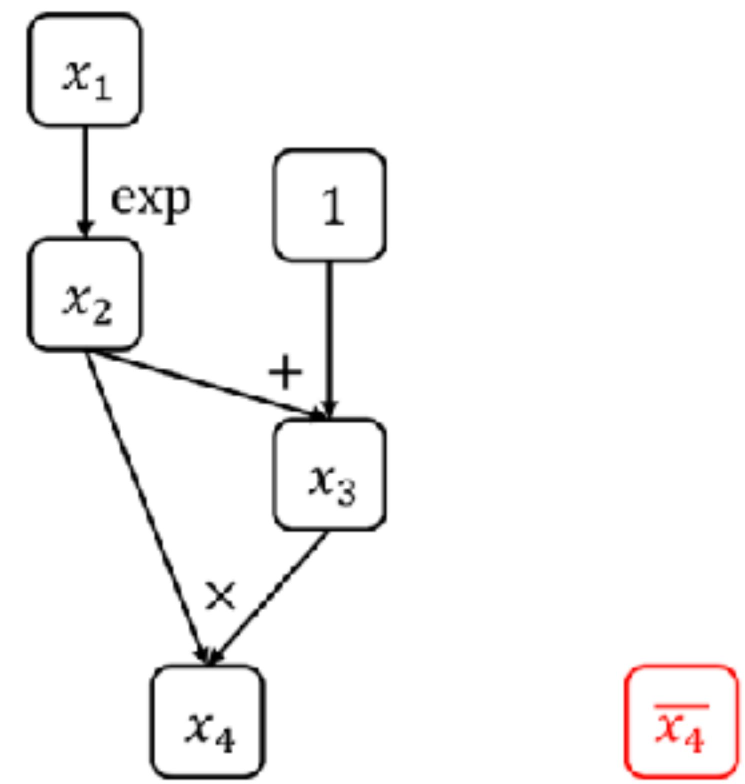
```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for
 input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```



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: $\overline{x_4}$

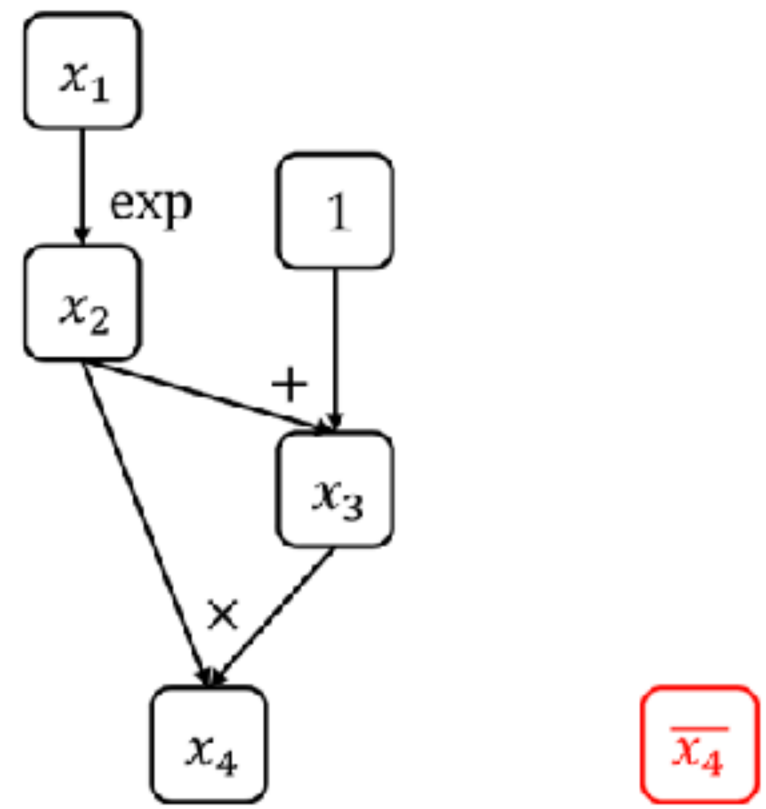


AutoDiff Algorithm

```
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

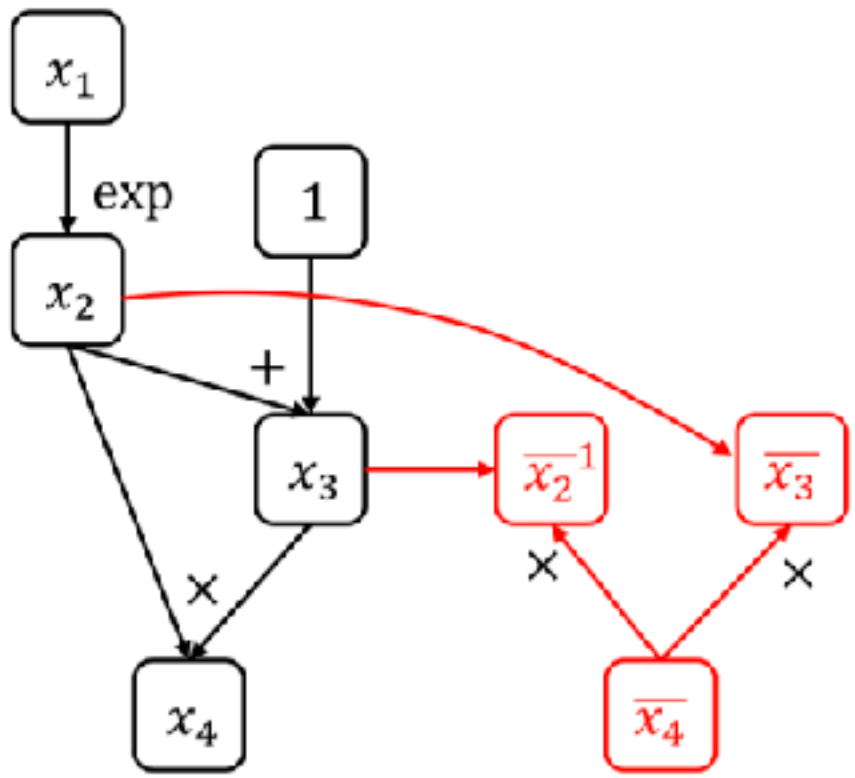


AutoDiff Algorithm

```
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

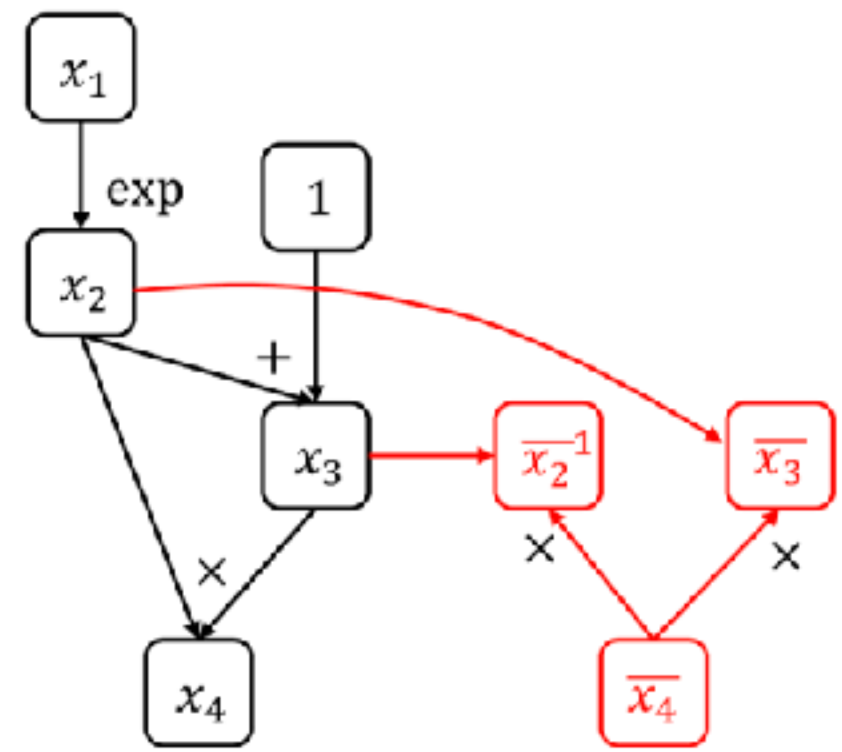


AutoDiff Algorithm

```
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

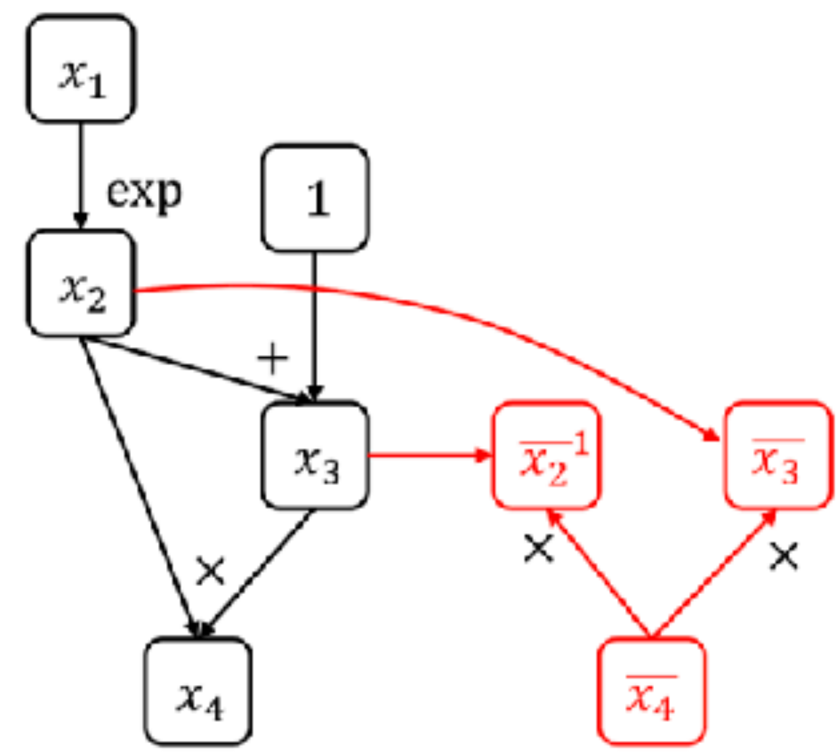


AutoDiff Algorithm

```
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

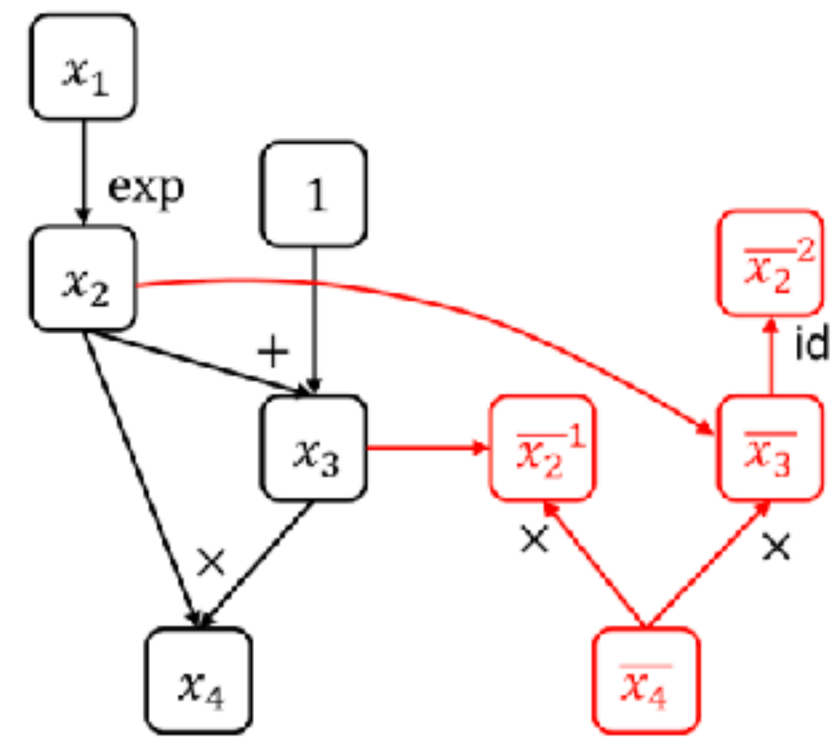


AutoDiff Algorithm

```
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
```

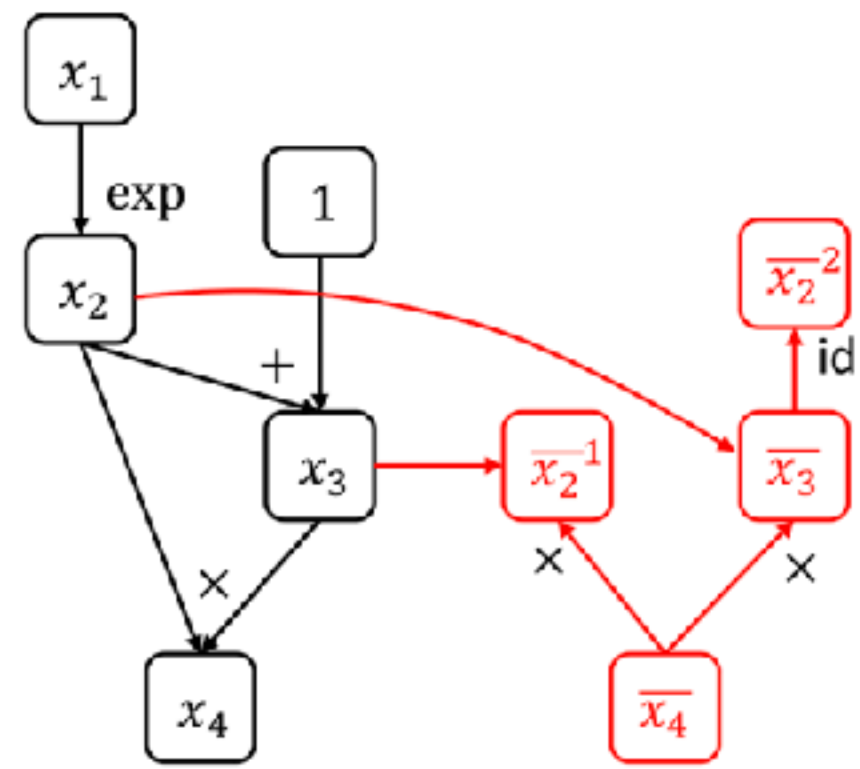
node_to_grad:

x_4: $\overline{x_4}$

x_3: $\overline{x_3}$

x_2: $\overline{x_2}^1$, $\overline{x_2}^2$

x_4	x_3	x_2	x_1

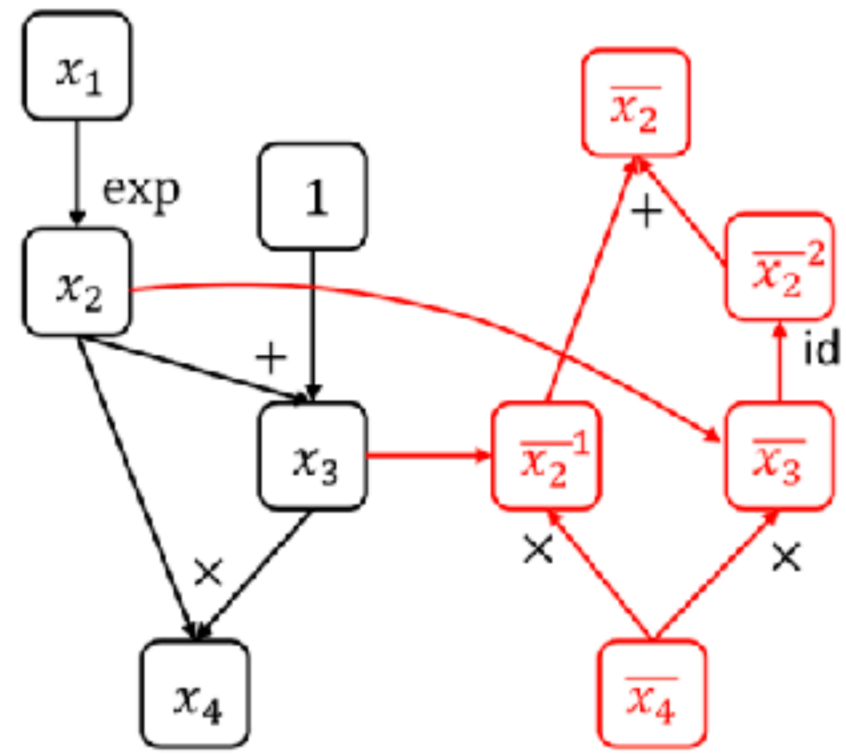


AutoDiff Algorithm

```
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

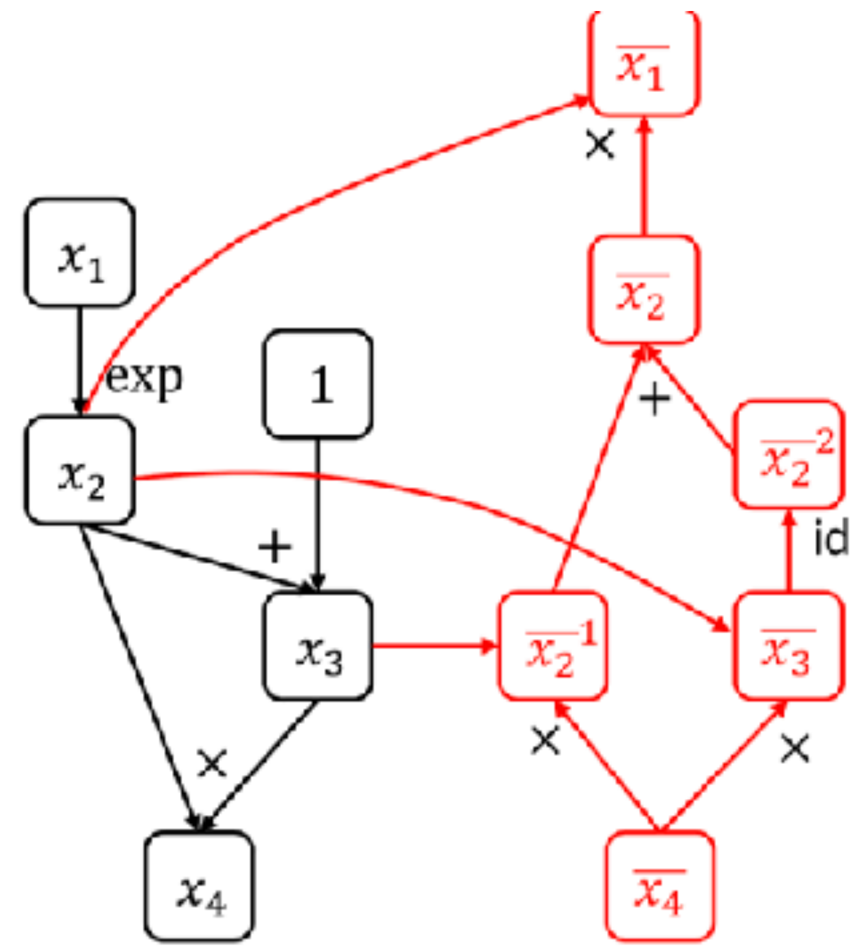


AutoDiff Algorithm

```
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

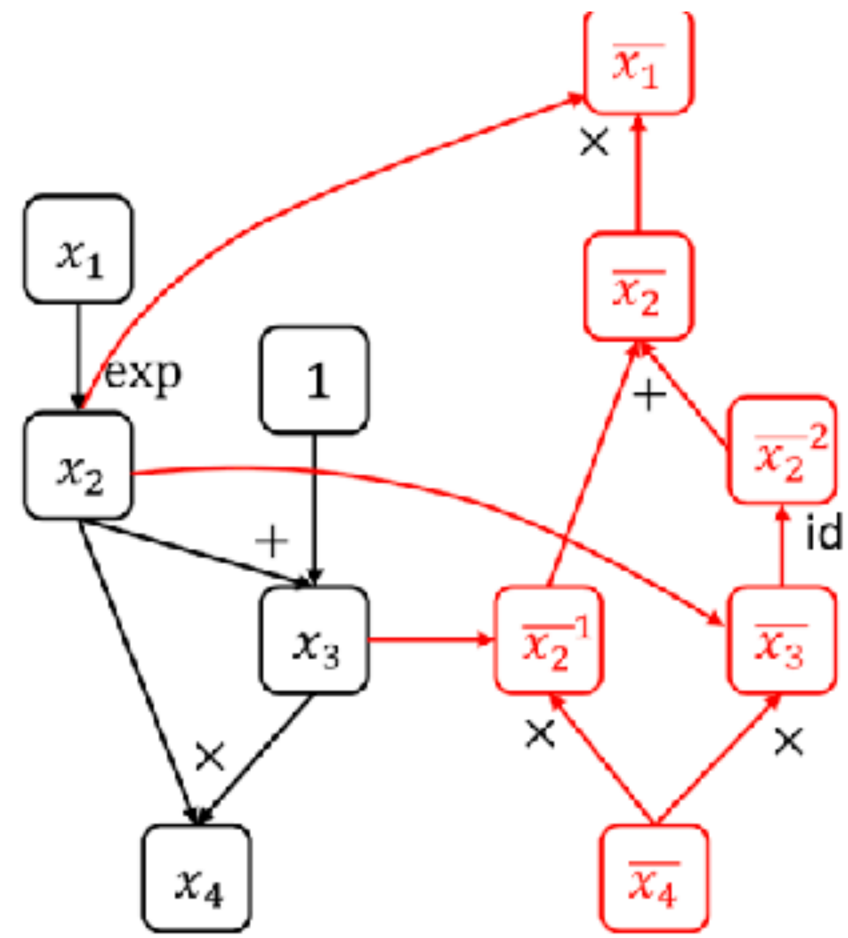


AutoDiff Algorithm

```
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_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:

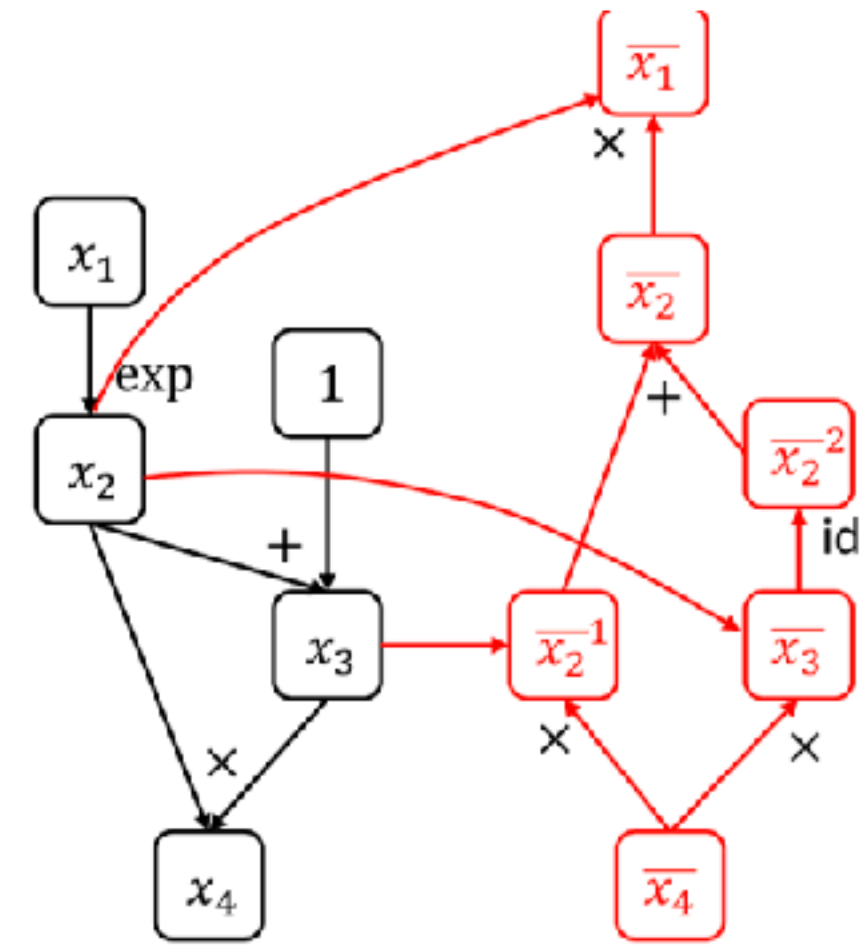
x_4: $\overline{x_4}$

x_3: $\overline{x_3}$

x_2: $\overline{x_2}^1$, $\overline{x_2}^2$

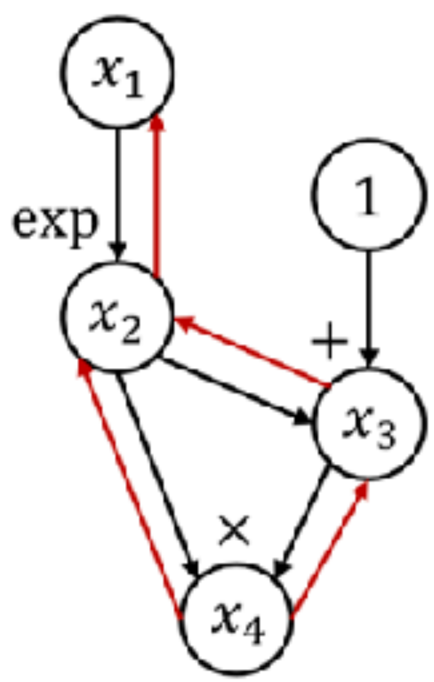
x_1: $\overline{x_1}$

x_4	x_3	x_2	x_1

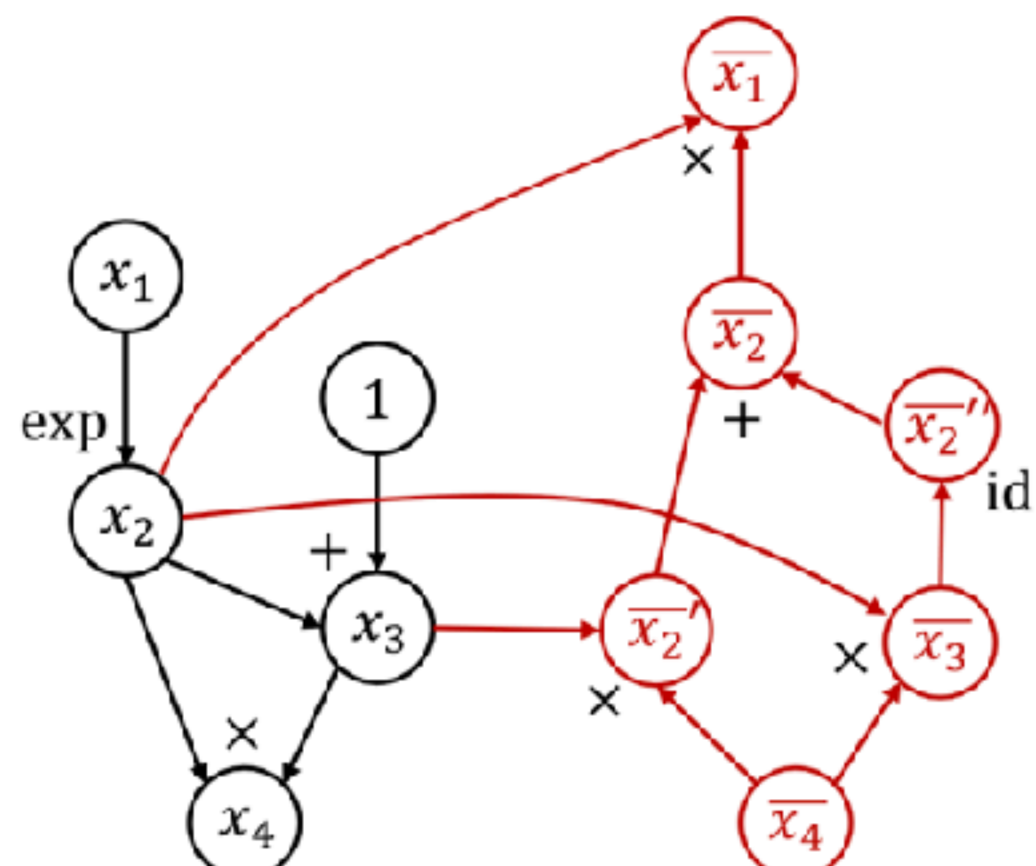


Backpropagation vs AutoDiff

Backpropagation



AutoDiff



Recap

- Numerical differentiation
  - Tool to check the correctness of implementation
- Backpropagation
  - Easy to understand and implement
  - Bad for memory use and schedule optimization
- Automatic differentiation
  - Generate gradient computation to entire computation graph
  - Better for system optimization

References

- Automatic differentiation in machine learning: a survey https://arxiv.org/abs/1502.05767

- CS231n backpropagation: http://cs231n.github.io/optimization-2/