CSCE 585: Machine Learning Systems

Lecture 9: Backpropagation and Automatic Differentiation

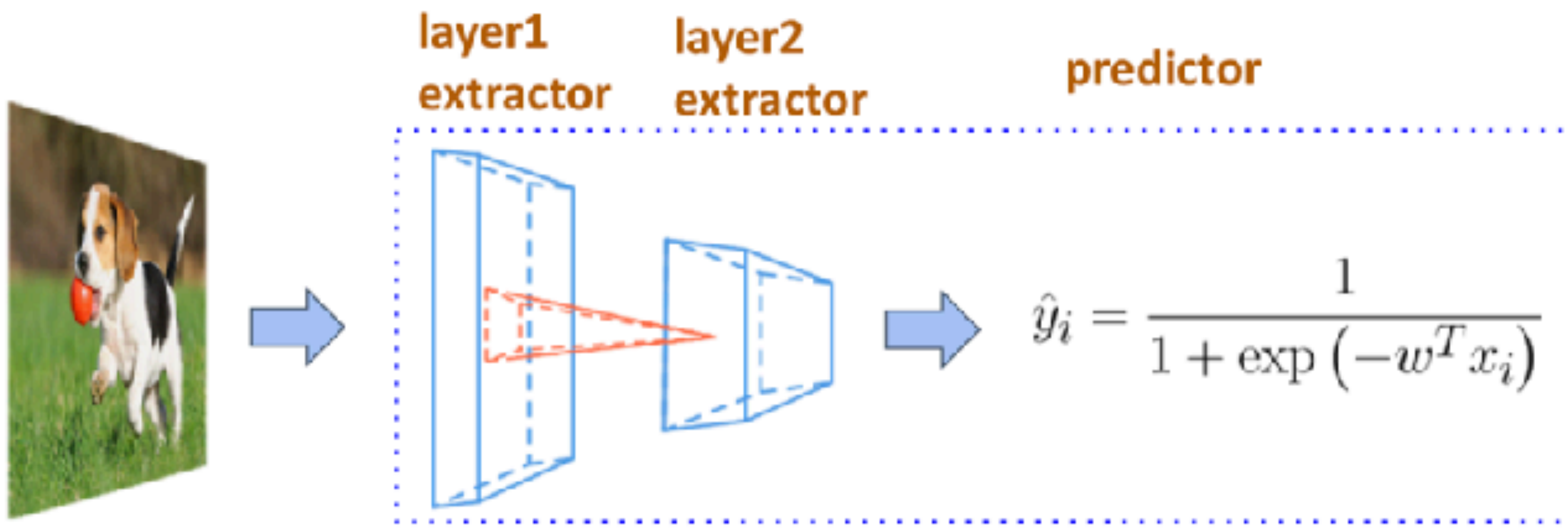
Pooyan Jamshidi

UNIVERSITY OF South Carolina

Motivation: Why Gradients Matter

- Training = optimization.
- Optimization = gradients.
- Gradients tell us how to change parameters to reduce loss.
- Equation:
$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathscr{L}(\theta_t)$$

Model Training Overview



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

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

The Three Ways to Differentiate

Method	Type	Pros	Cons
Numerical	Approximate	Simple	Inaccurate, slow

The Three Ways to Differentiate

Method	Type	Pros	Cons
Numerical	Approximate	Simple	Inaccurate, slow
Symbolic	Exact	Closed form	Expression swell

The Three Ways to Differentiate

Method	Type	Pros	Cons
Numerical	Approximate	Simple	Inaccurate, slow
Symbolic	Exact	Closed form	Expression swell
Automatic	Exact	Efficient, flexible	Requires computational graph

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

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

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

$$[-0.8 \quad 0.3] \cdot \begin{bmatrix} 0.5 \\ -0.2 \end{bmatrix} \qquad\qquad [-0.8 + \varepsilon \quad 0.3] \cdot \begin{bmatrix} 0.5 \\ -0.2 \end{bmatrix}$$

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.

Symbolic Differentiation

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

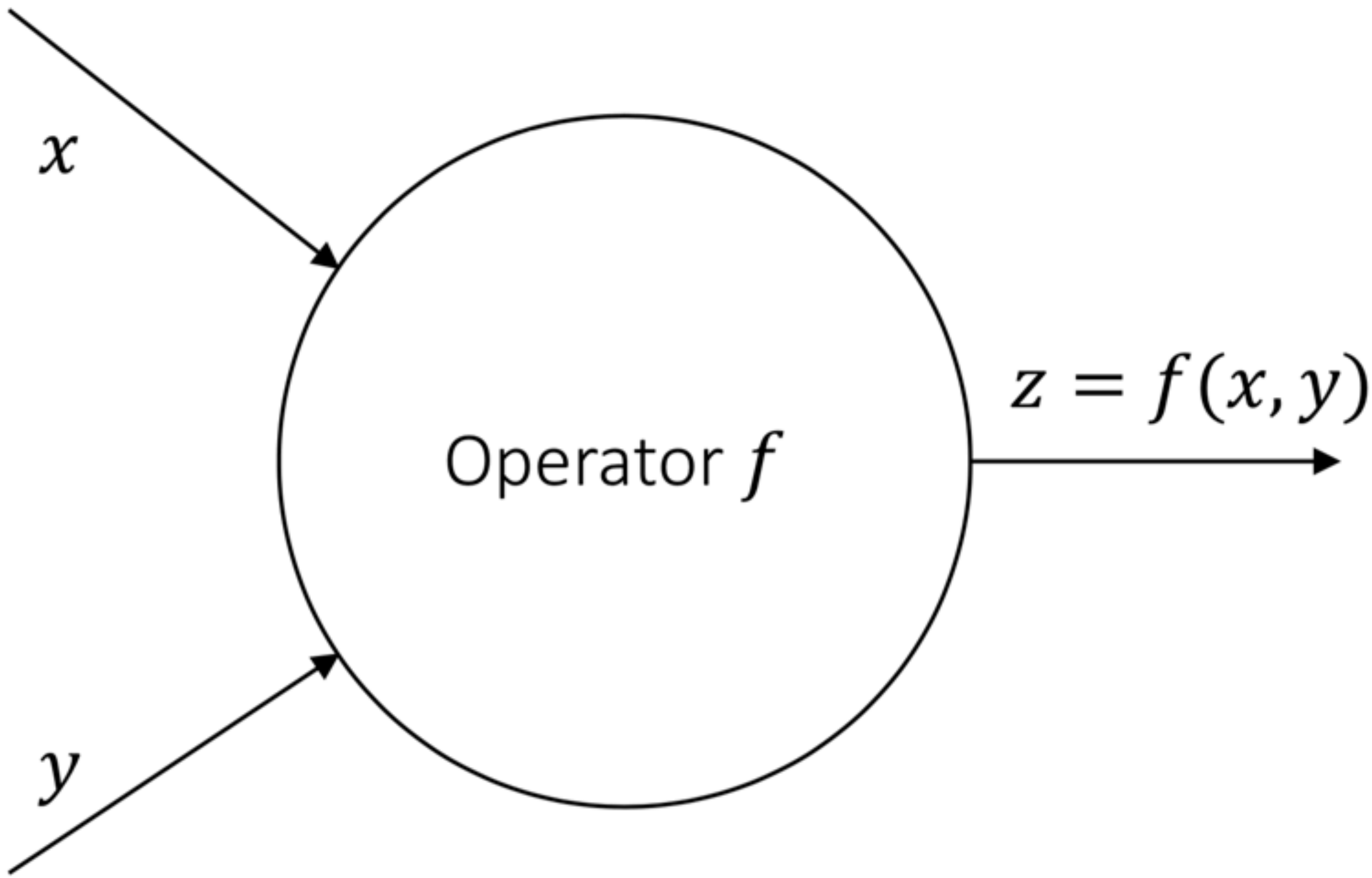
$$\frac{d(f+g)}{dx} = \frac{df}{dx} + \frac{dg}{dx} \qquad \frac{d(fg)}{dx} = \frac{df}{dx}g + f\frac{dg}{dx} \qquad \frac{d\big(h(x)\big)}{dx} = \frac{df\big(g(x)\big)}{dx} \cdot \frac{dg(x)}{x}$$

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

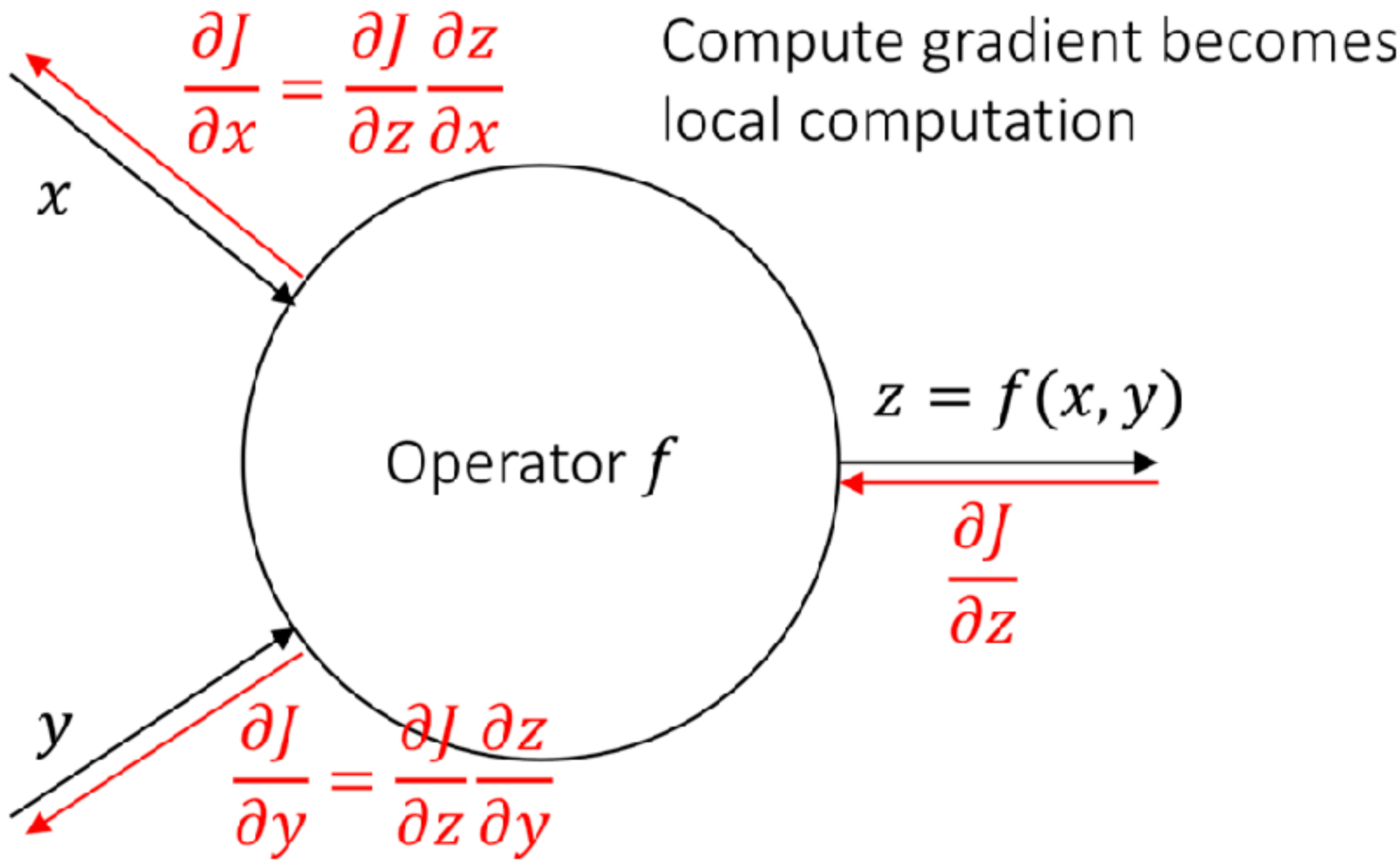
✗ Wasteful to keep around intermediate symbolic expressions if we only need a numeric value of the gradient in the end

✗ Prone to error

Backpropagation



Backpropagation

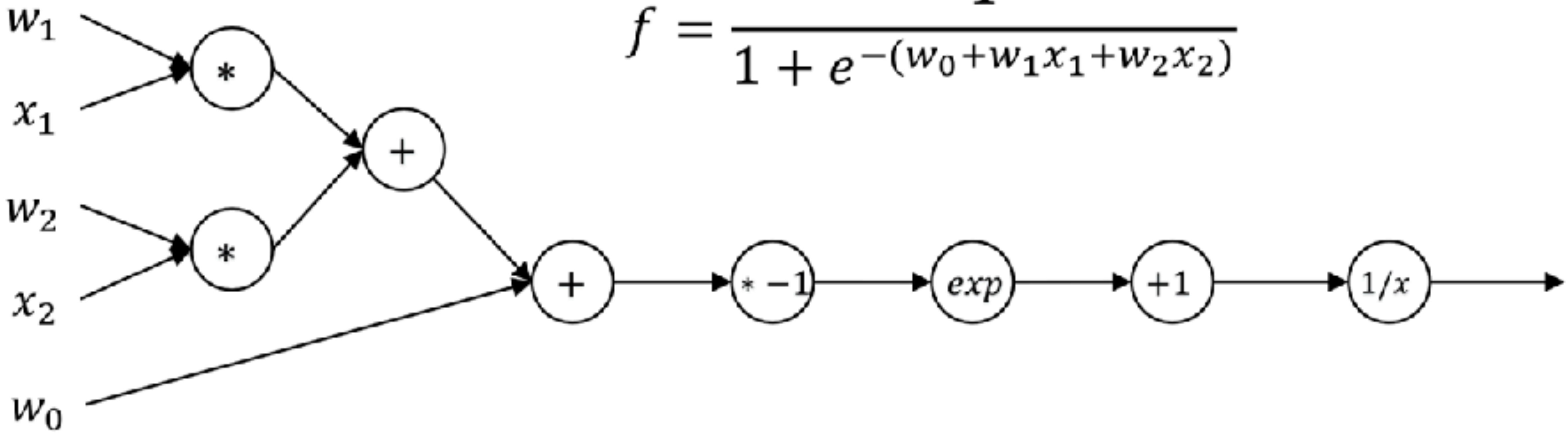


Backpropagation simple example

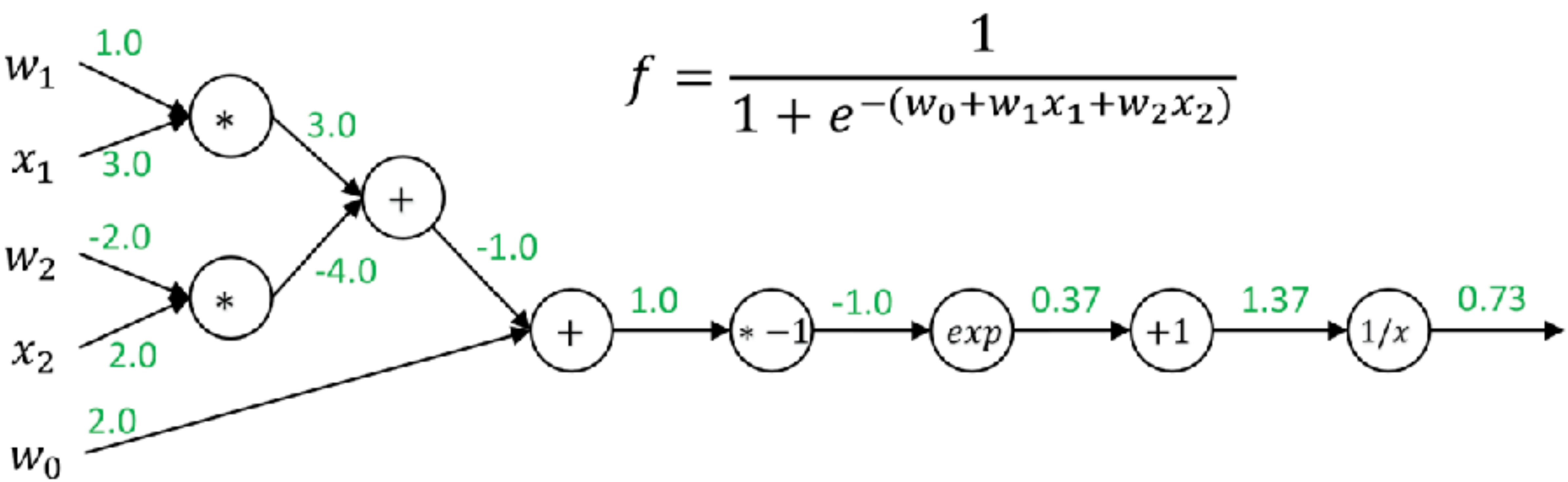
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

Backpropagation simple example

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

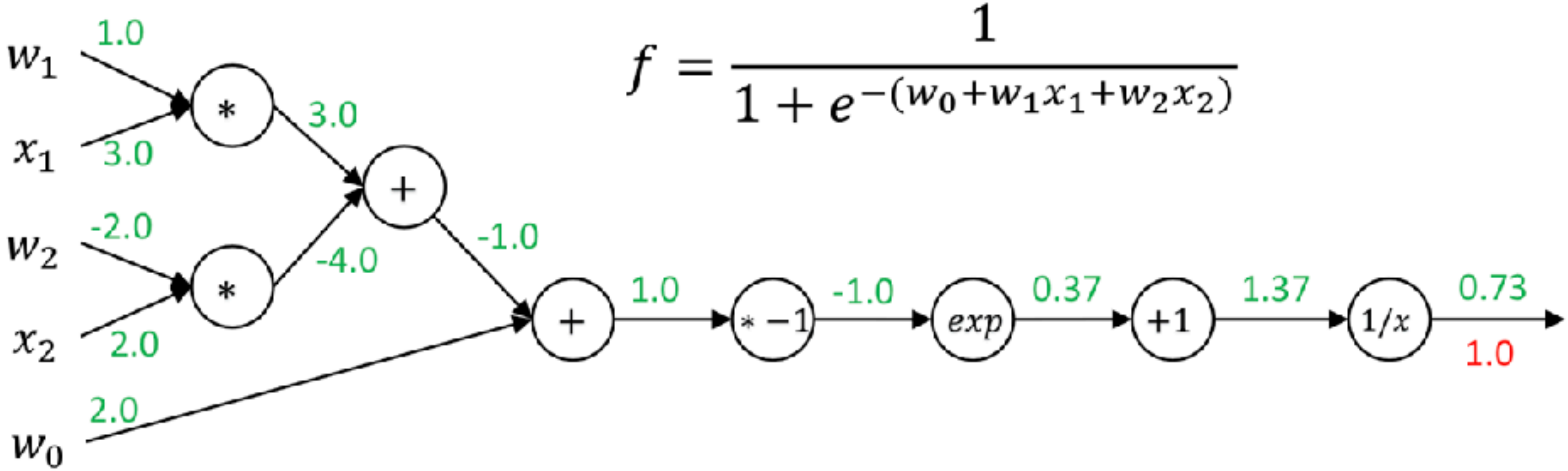


Backpropagation simple example

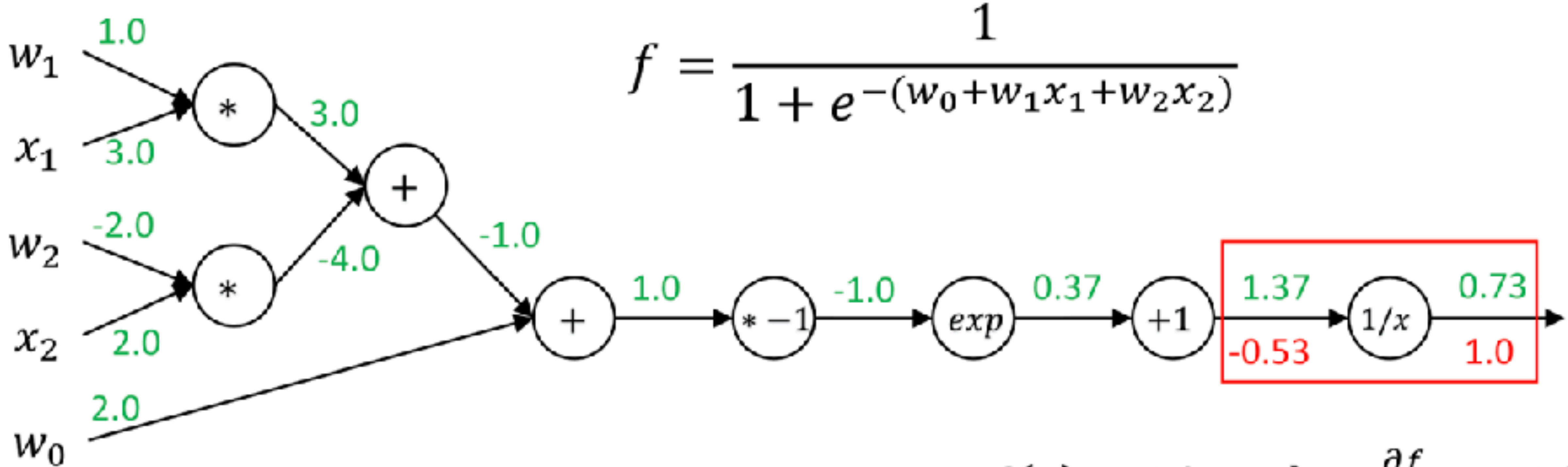


Backpropagation simple example

$$f = \frac{1}{1+e^{-(w_0+w_1x_1+w_2x_2)}}$$



Backpropagation simple example



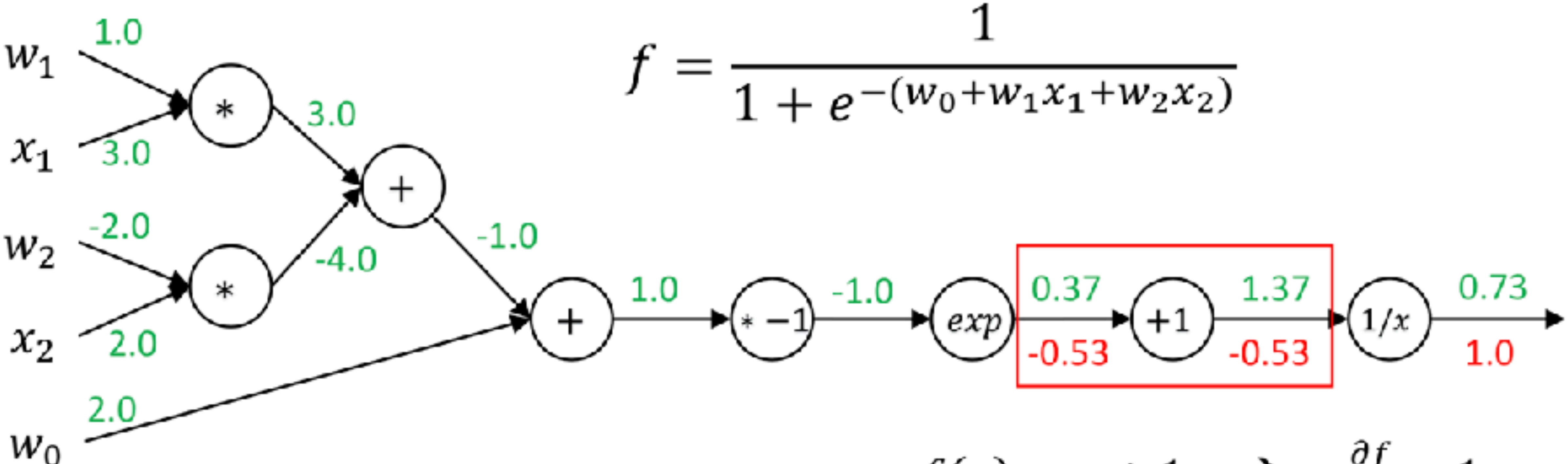
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

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

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

Backpropagation simple example

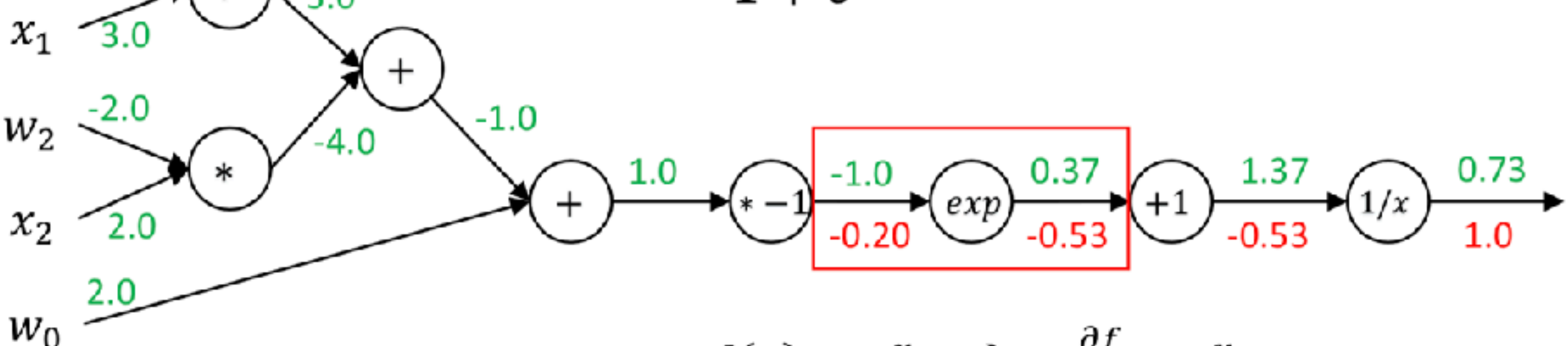
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



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

Backpropagation simple example

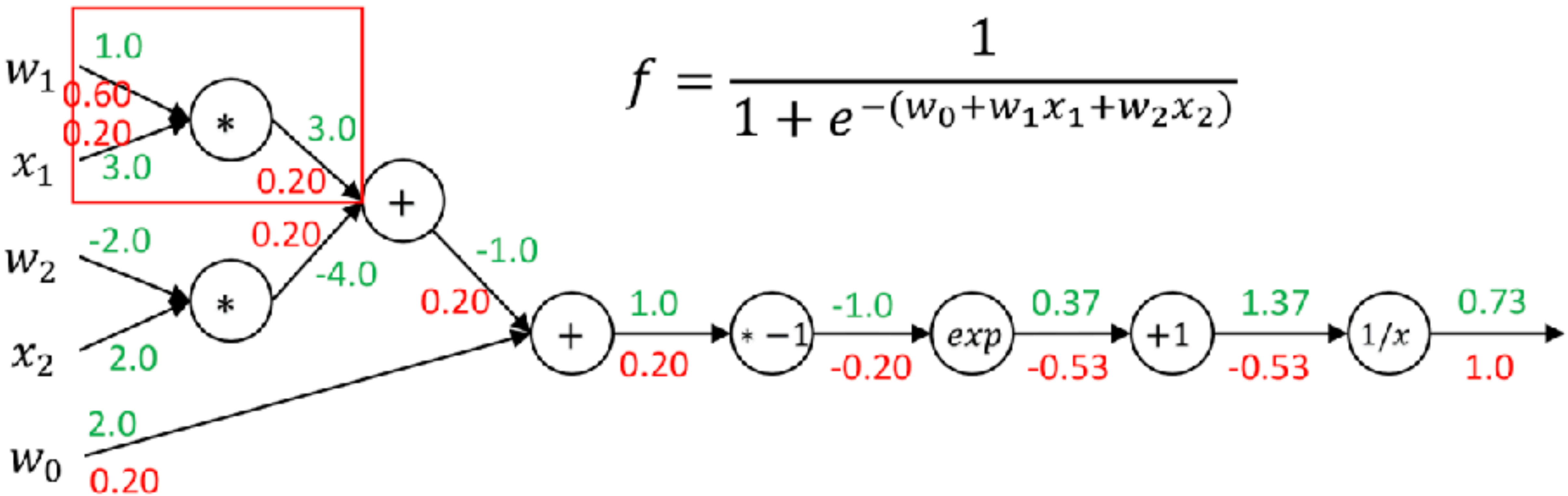
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



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

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

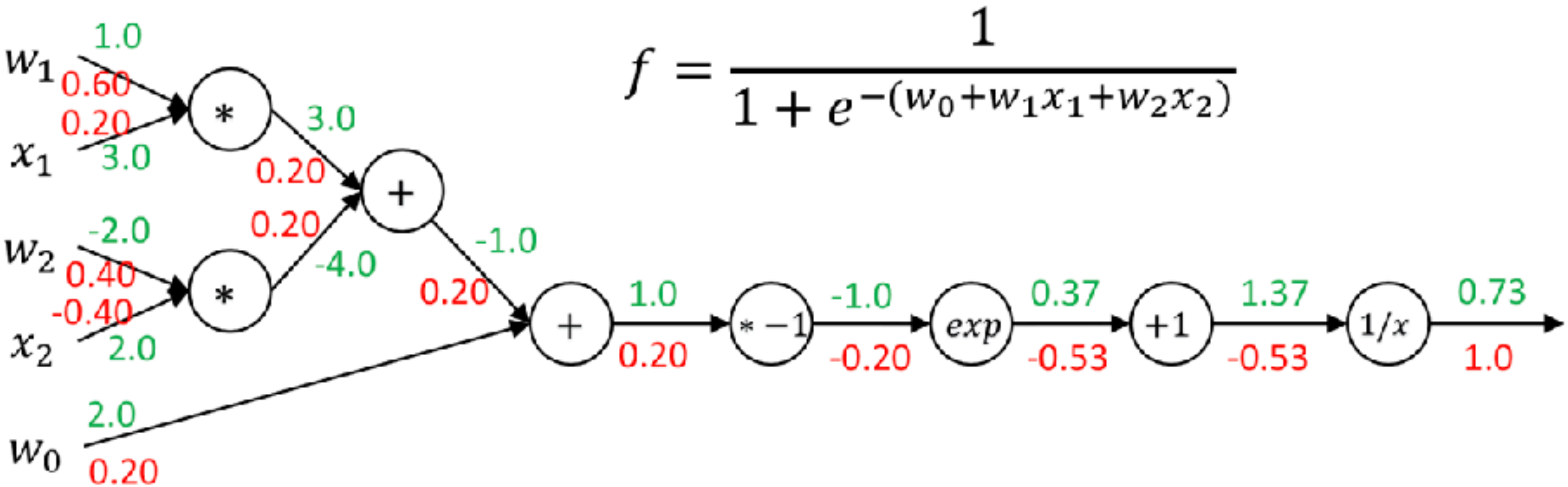
Backpropagation simple example



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

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

Backpropagation simple example



$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

Forward Pass

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator   #(1)
num = x + sigy # numerator                                #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                               #(4)
xpysqr = xpy**2                                           #(5)
den = sigx + xpysqr # denominator                         #(6)
invden = 1.0 / den                                        #(7)
f = num * invden # done!                                  #(8)
```

Backward Pass

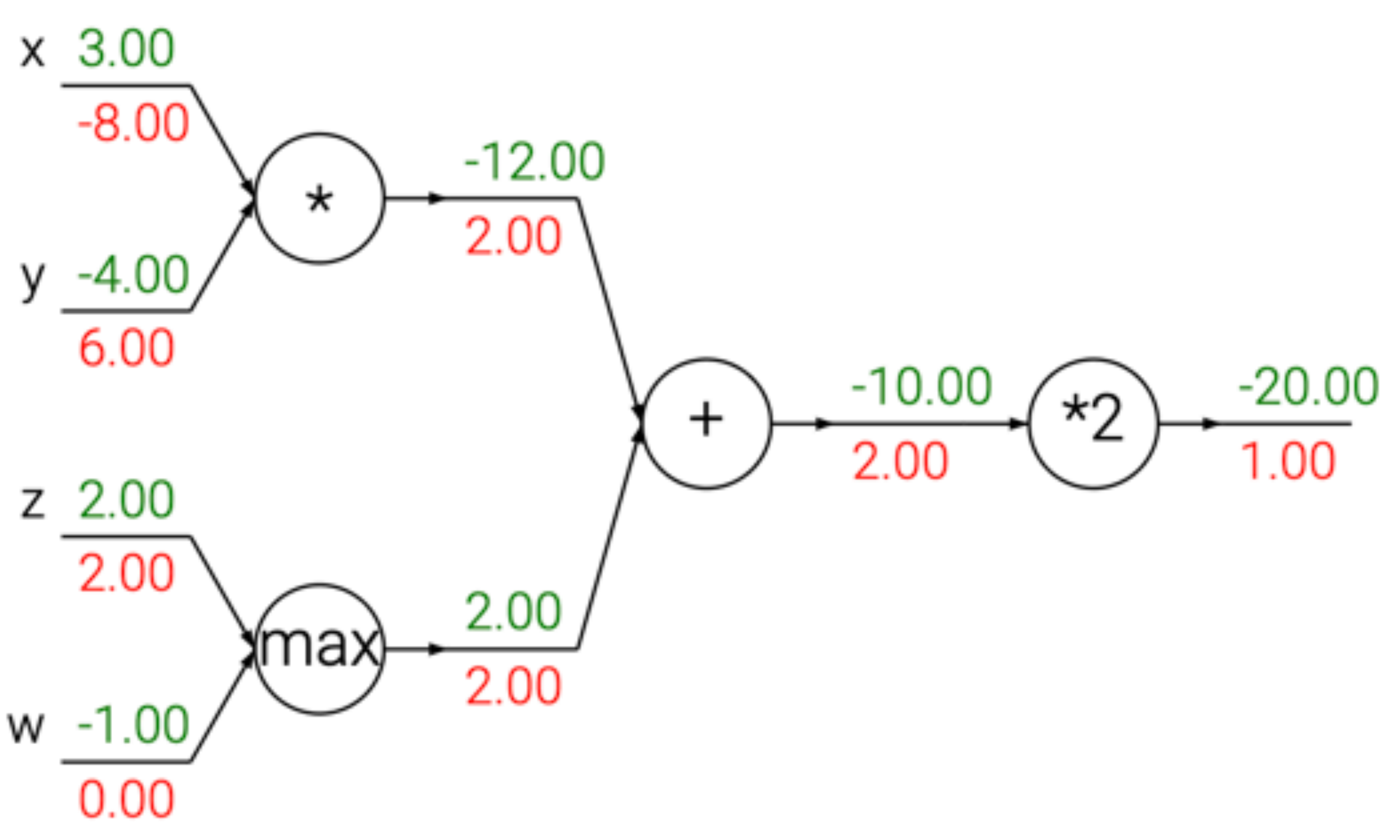
```
# backprop f = num * invden
dnum = invden # gradient on numerator                             #(8)
dinvden = num                                                     #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                                #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                                #(6)
dxpysqr = (1) * dden                                              #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                                        #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                                   #(4)
dy = (1) * dxpy                                                   #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below  #(3)
# backprop num = x + sigy
dx += (1) * dnum                                                  #(2)
dsigy = (1) * dnum                                                #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                                 #(1)
# done! phew
```

Cache forward pass variables. To compute the backward pass it is very helpful to have some of the variables that were used in the forward pass. In practice you want to structure your code so that you cache these variables, and so that they are available during backpropagation. If this is too difficult, it is possible (but wasteful) to recompute them.

Gradients add up at forks. The forward expression involves the variables **x,y** multiple times, so when we perform backpropagation we must be careful to use `+=` instead of `=` to accumulate the gradient on these variables (otherwise we would overwrite it). This follows the *multivariable chain rule* in Calculus, which states that if a variable branches out to different parts of the circuit, then the gradients that flow back to it will add.

Patterns in backward flow

It is interesting to note that in many cases the backward-flowing gradient can be interpreted on an intuitive level. For example, the three most commonly used gates in neural networks (*add,mul,max*), all have very simple interpretations in terms of how they act during backpropagation. Consider this example circuit:



An example circuit demonstrating the intuition behind the operations that backpropagation performs during the backward pass in order to compute the gradients on the inputs. Sum operation distributes gradients equally to all its inputs. Max operation routes the gradient to the higher input. Multiply gate takes the input activations, swaps them and multiplies by its gradient.

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.

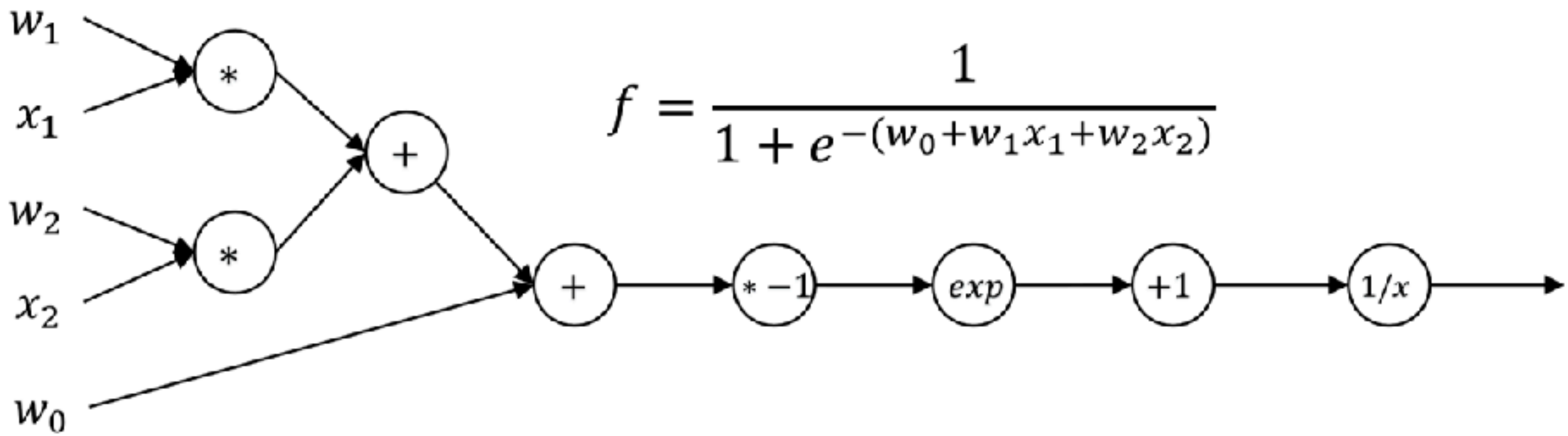
- Lack of flexibility, e.g., compute the gradient of gradient.

Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

Automatic Differentiation (autodiff)

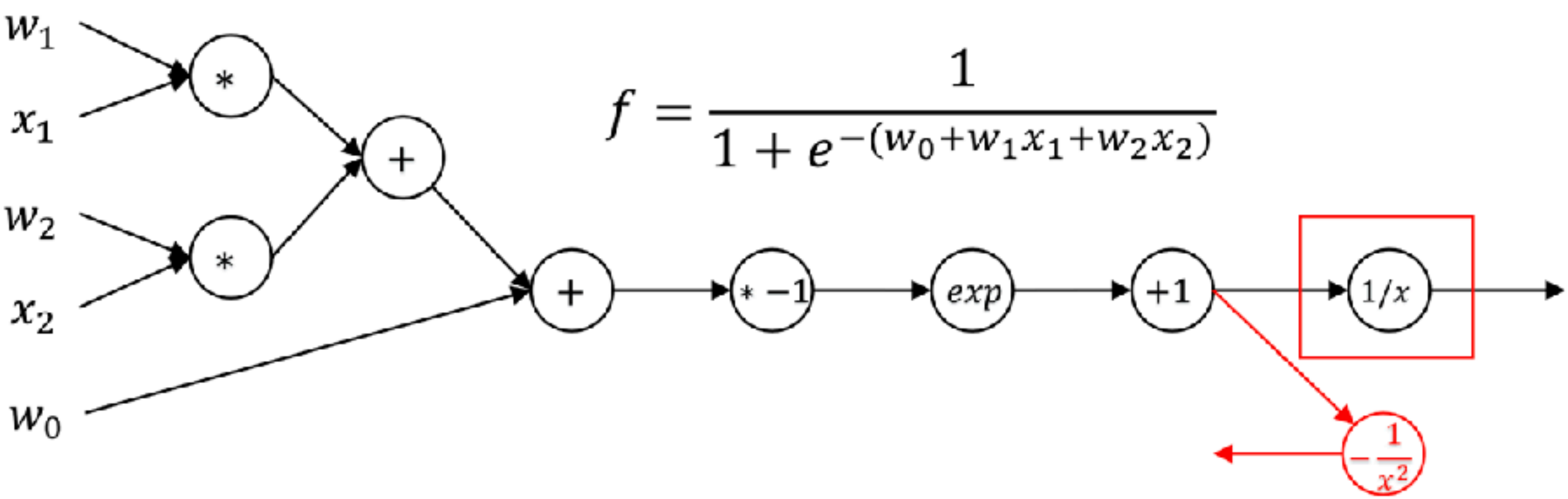
- Create computation graph for gradient computation



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

Automatic Differentiation (autodiff)

- Create computation graph for gradient computation



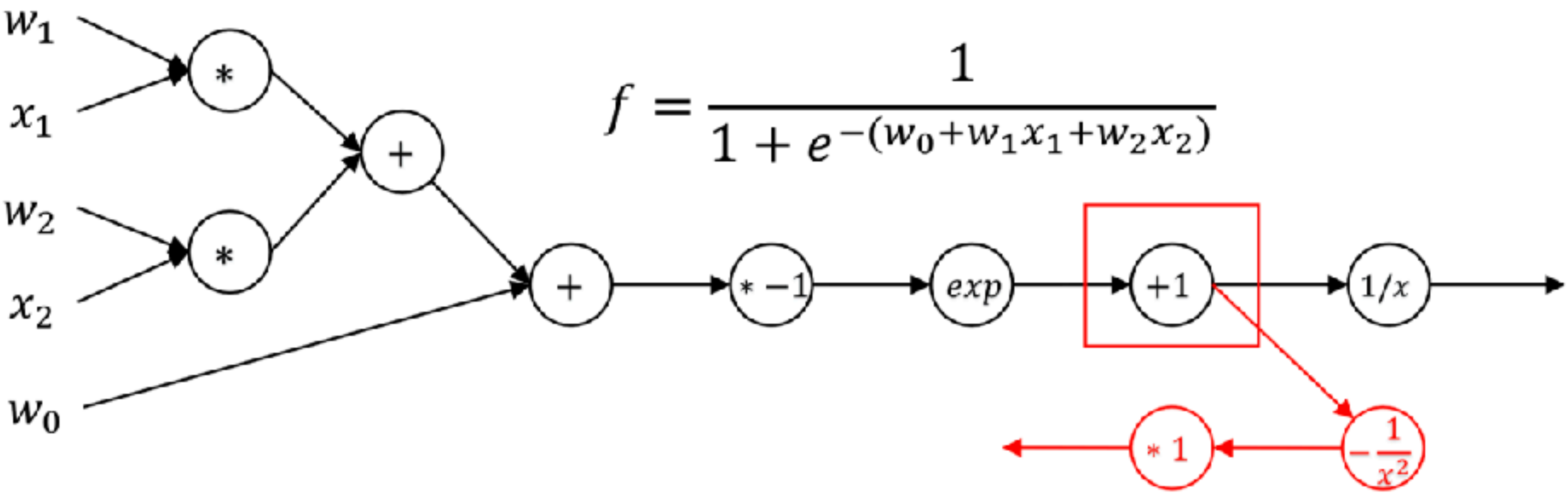
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

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

Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

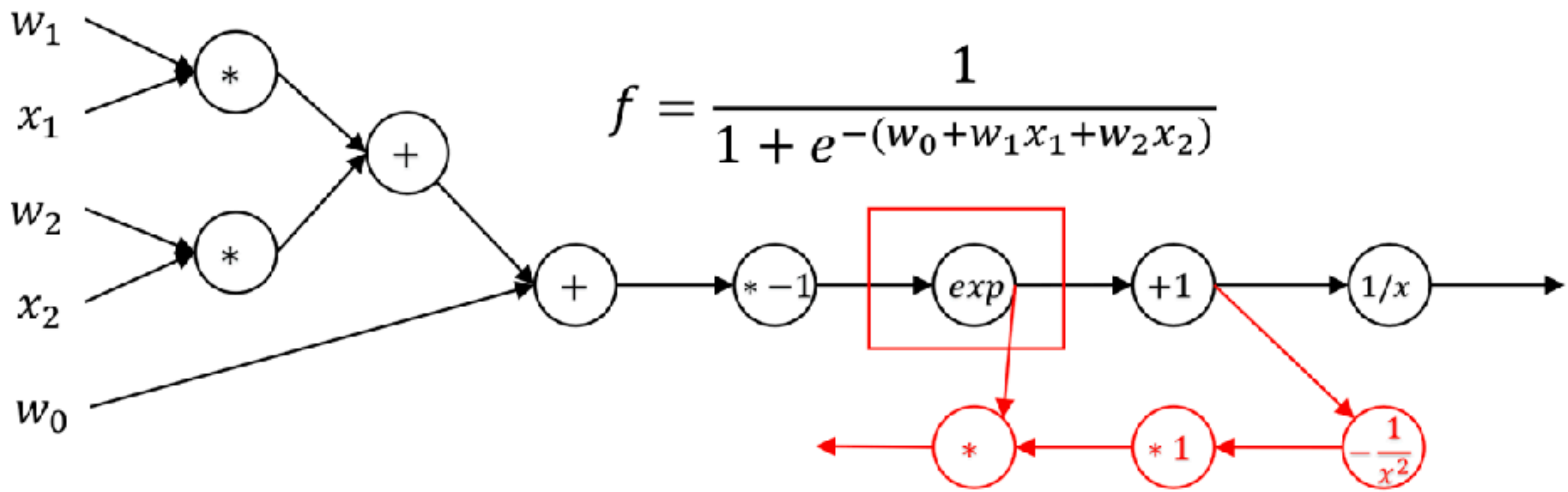
$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



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

Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

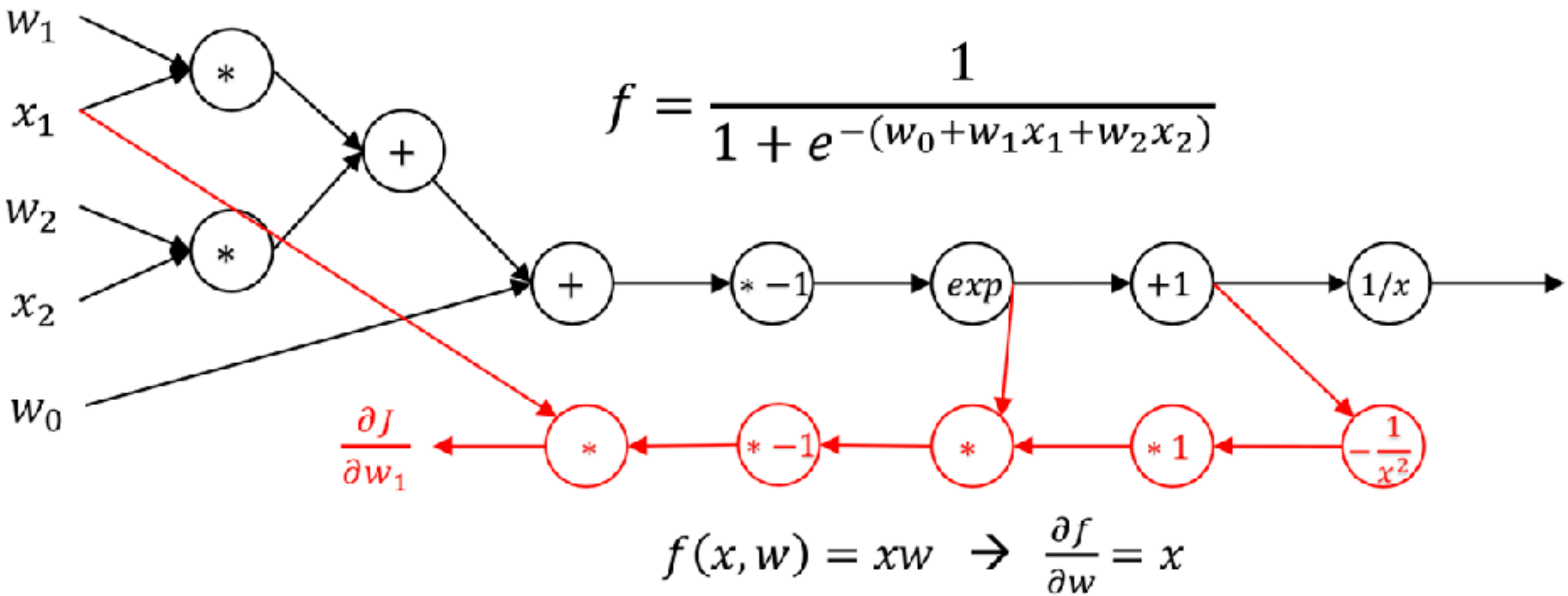


$$f = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

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

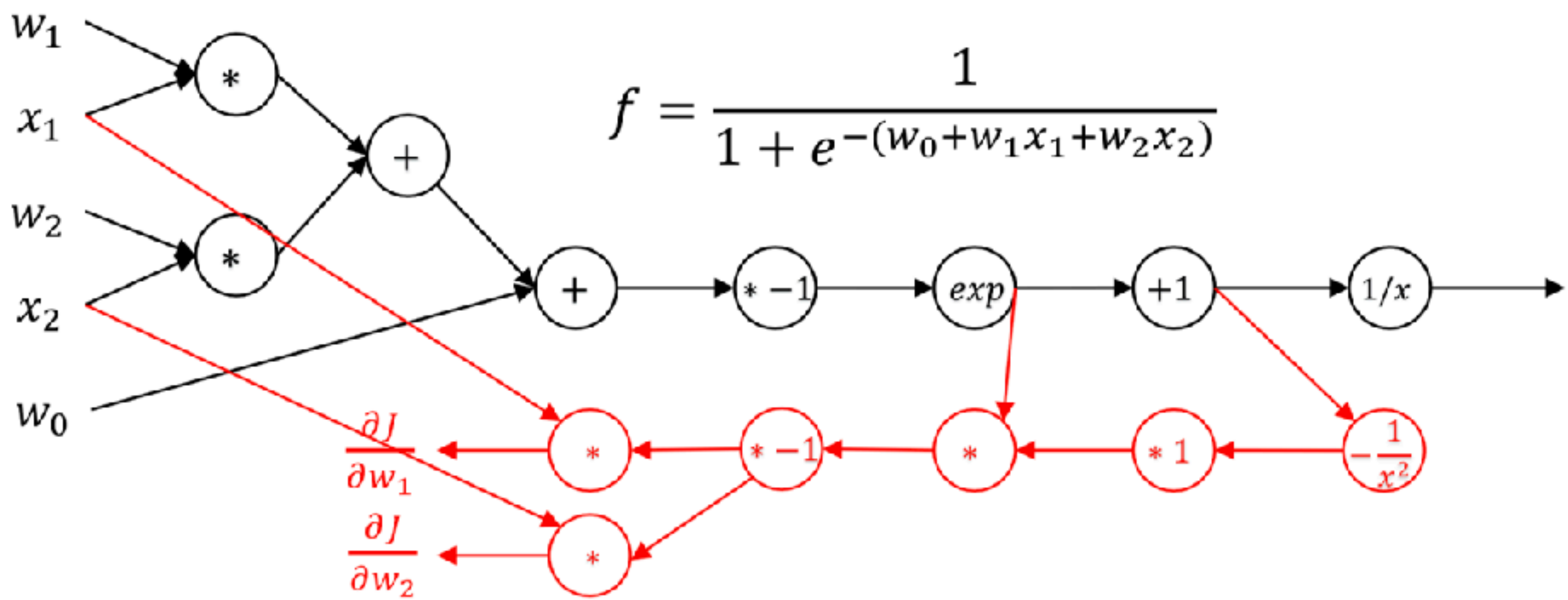
Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

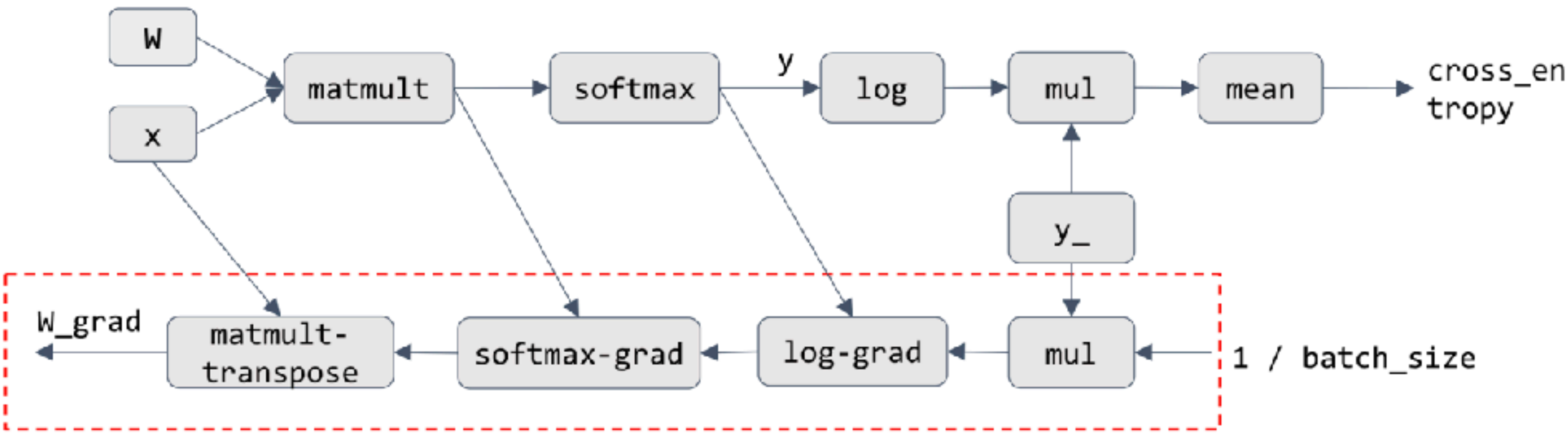


Automatic Differentiation (autodiff)

- Create computation graph for gradient computation

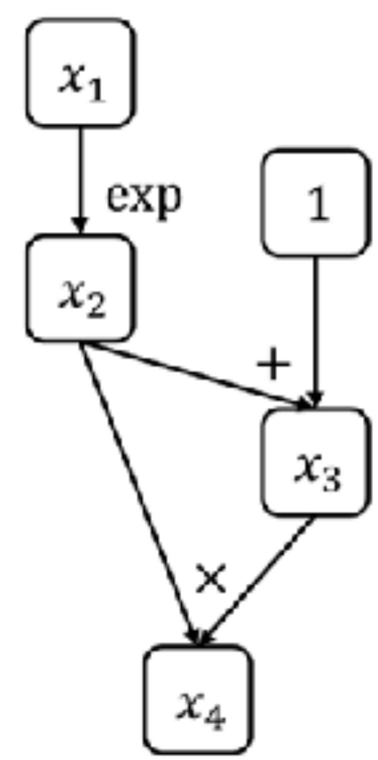


AutoDiff Algorithm



AutoDiff Algorithm

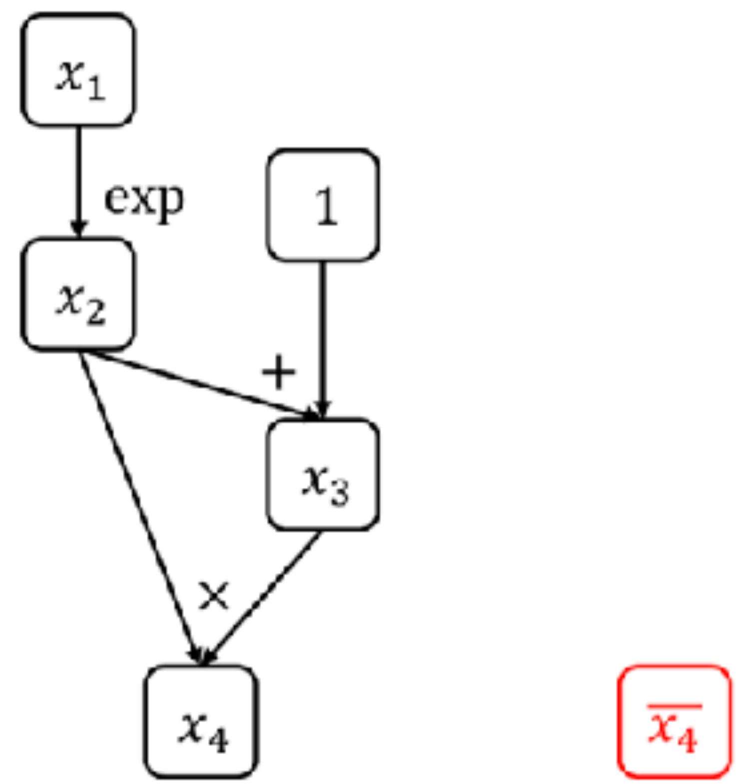
```
def gradient(out):
  node_to_grad[out] = 1
  nodes = get_node_list(out)
  for node in reverse_topo_order(nodes):
    grad ← sum partial adjoints from output edges
    input_grads ← node.op.gradient(input, grad) for
 input in node.inputs
    add input_grads to node_to_grad
  return node_to_grad
```



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: $\overline{x_4}$

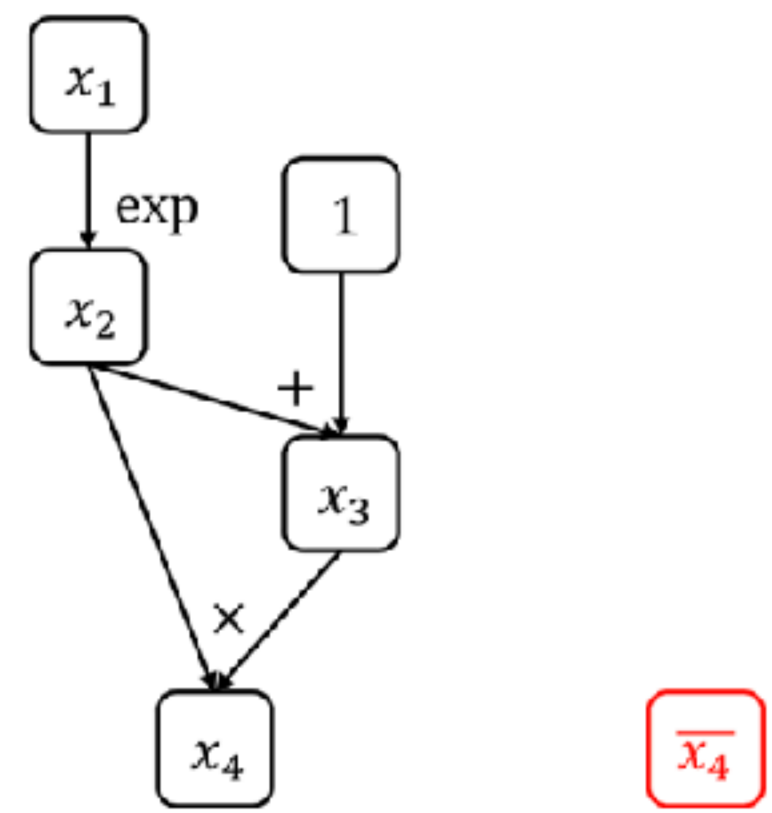


AutoDiff Algorithm

```
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

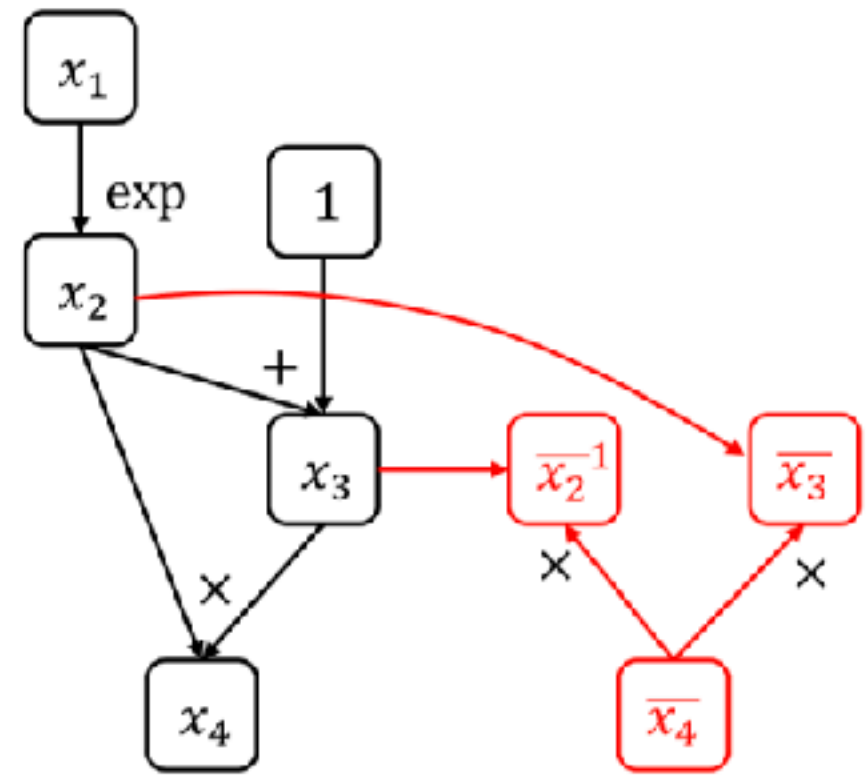


AutoDiff Algorithm

```
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

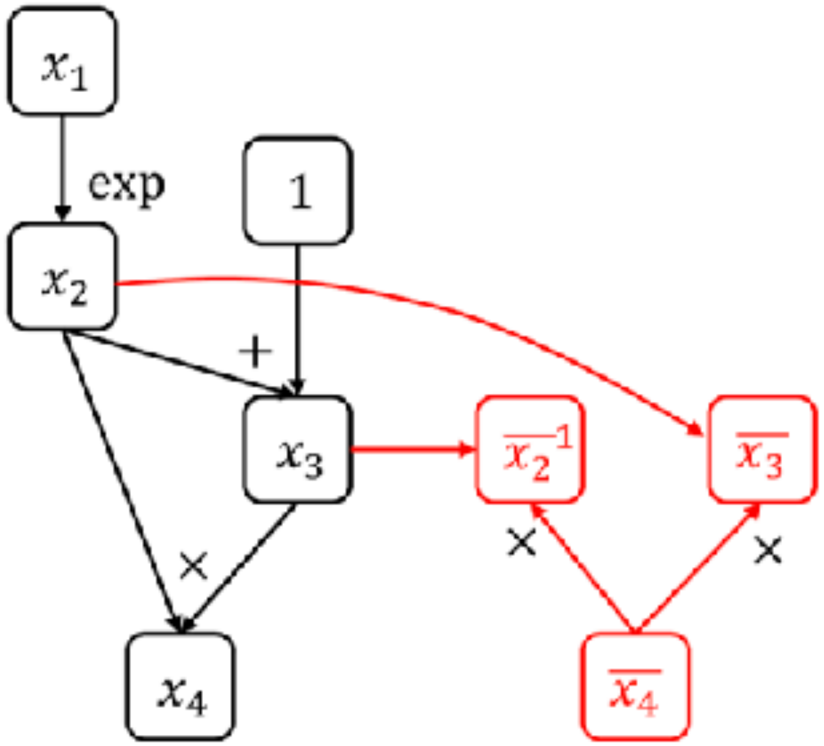


AutoDiff Algorithm

```
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

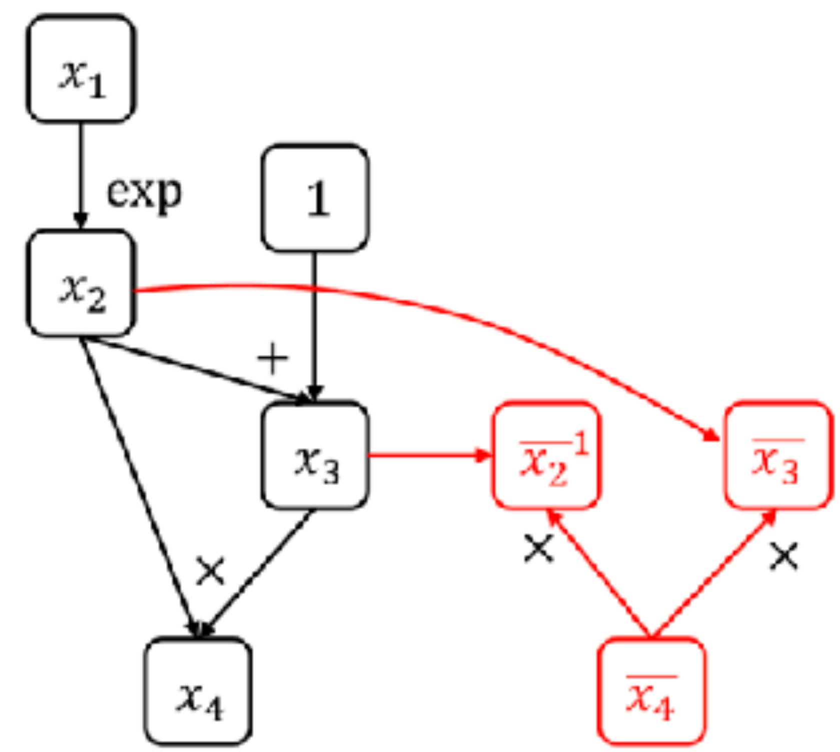


AutoDiff Algorithm

```
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
```

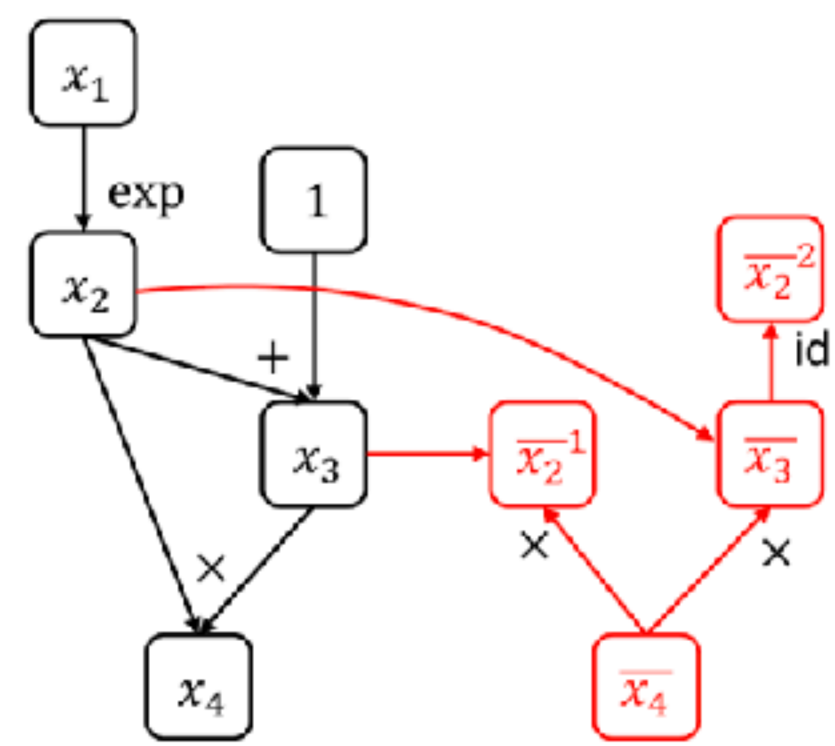
node_to_grad:

x_4: $\overline{x_4}$

x_3: $\overline{x_3}$

x_2: $\overline{x_2}^1$

x_4	x_3	x_2	x_1

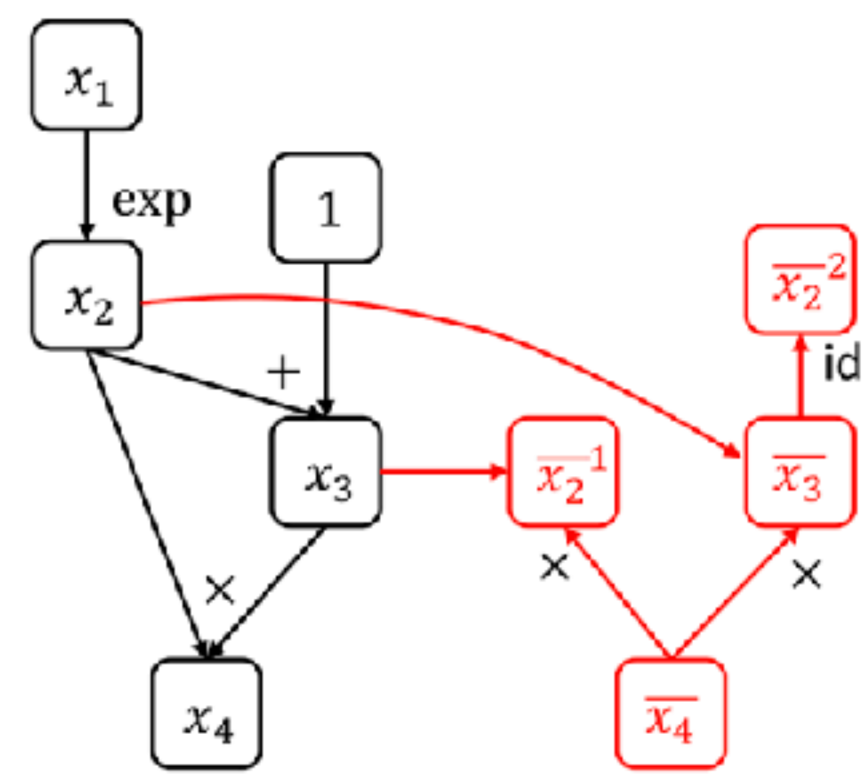


AutoDiff Algorithm

```
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

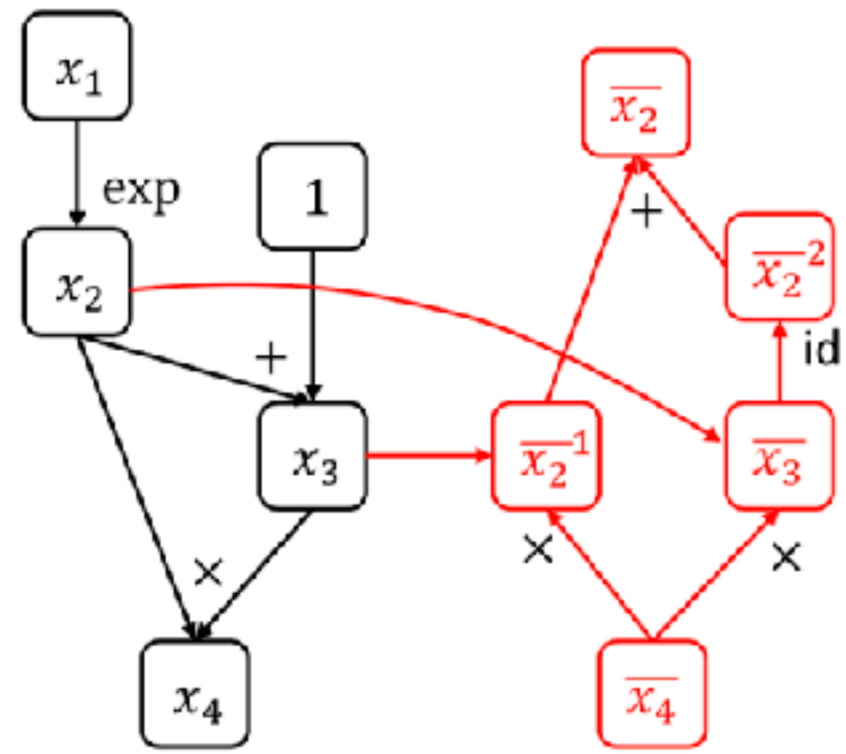


AutoDiff Algorithm

```
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
```

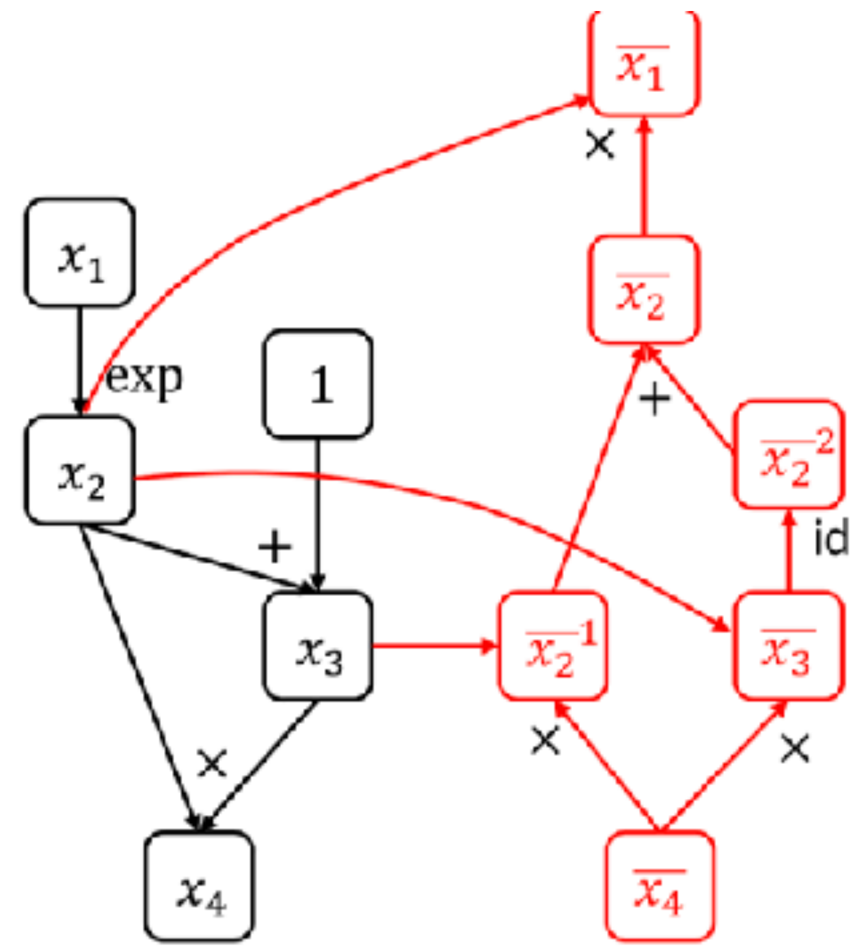
node_to_grad:

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



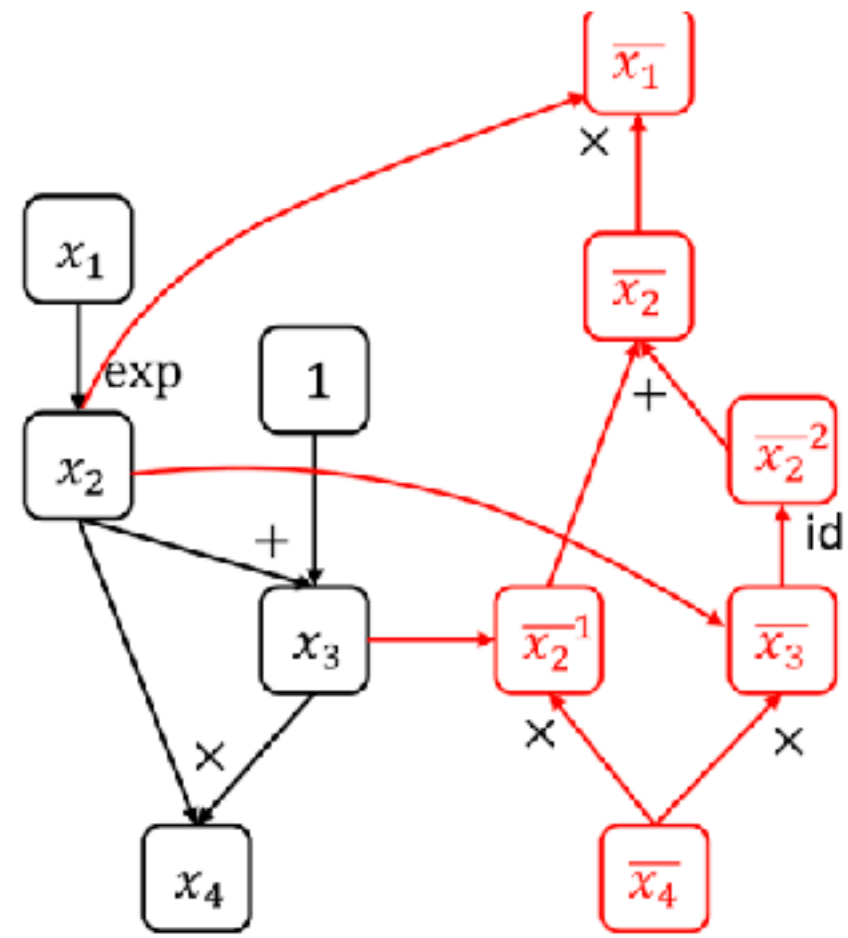
AutoDiff Algorithm

```
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:

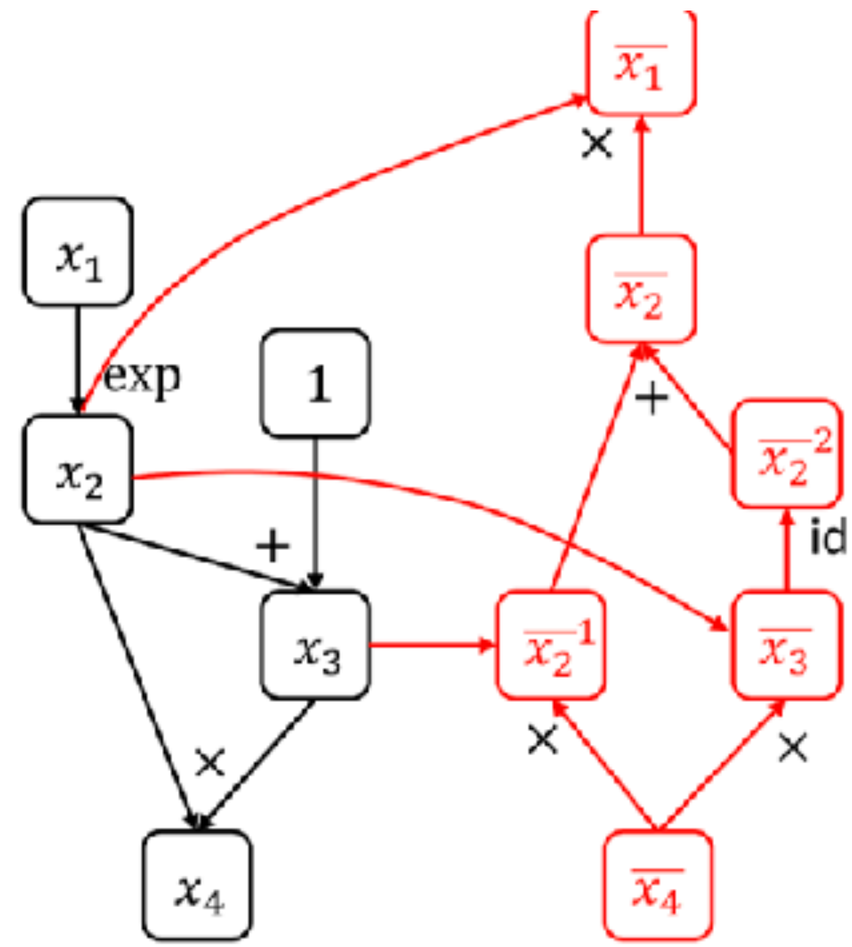
x_4: $\overline{x_4}$

x_3: $\overline{x_3}$

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

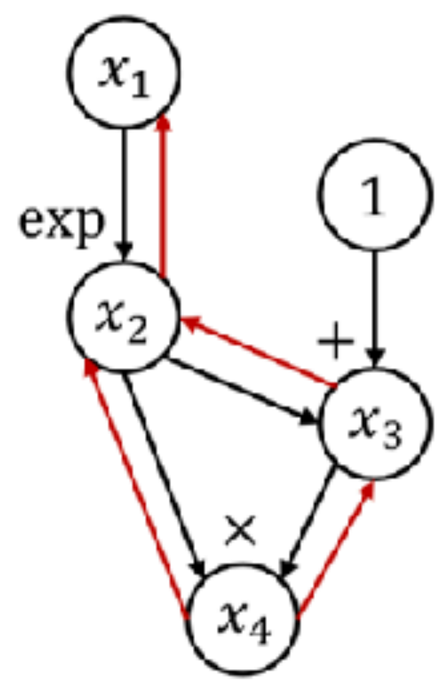
x_1: $\overline{x_1}$

x_4	x_3	x_2	x_1

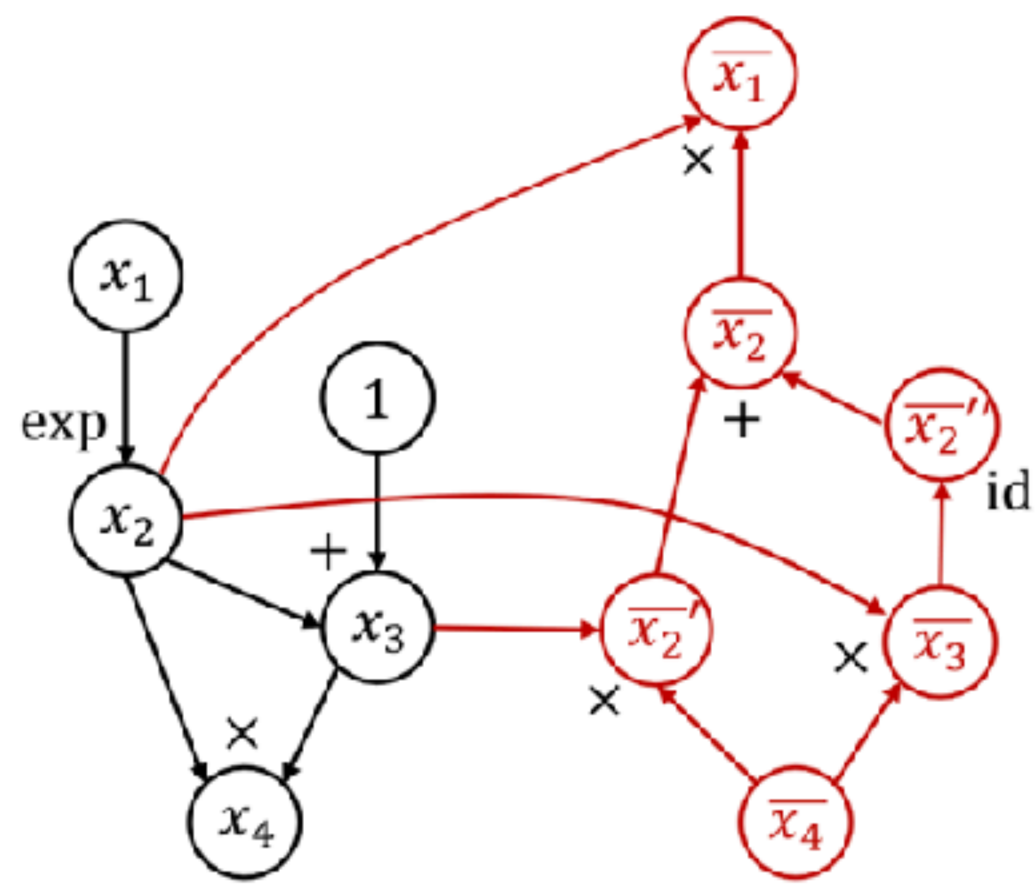


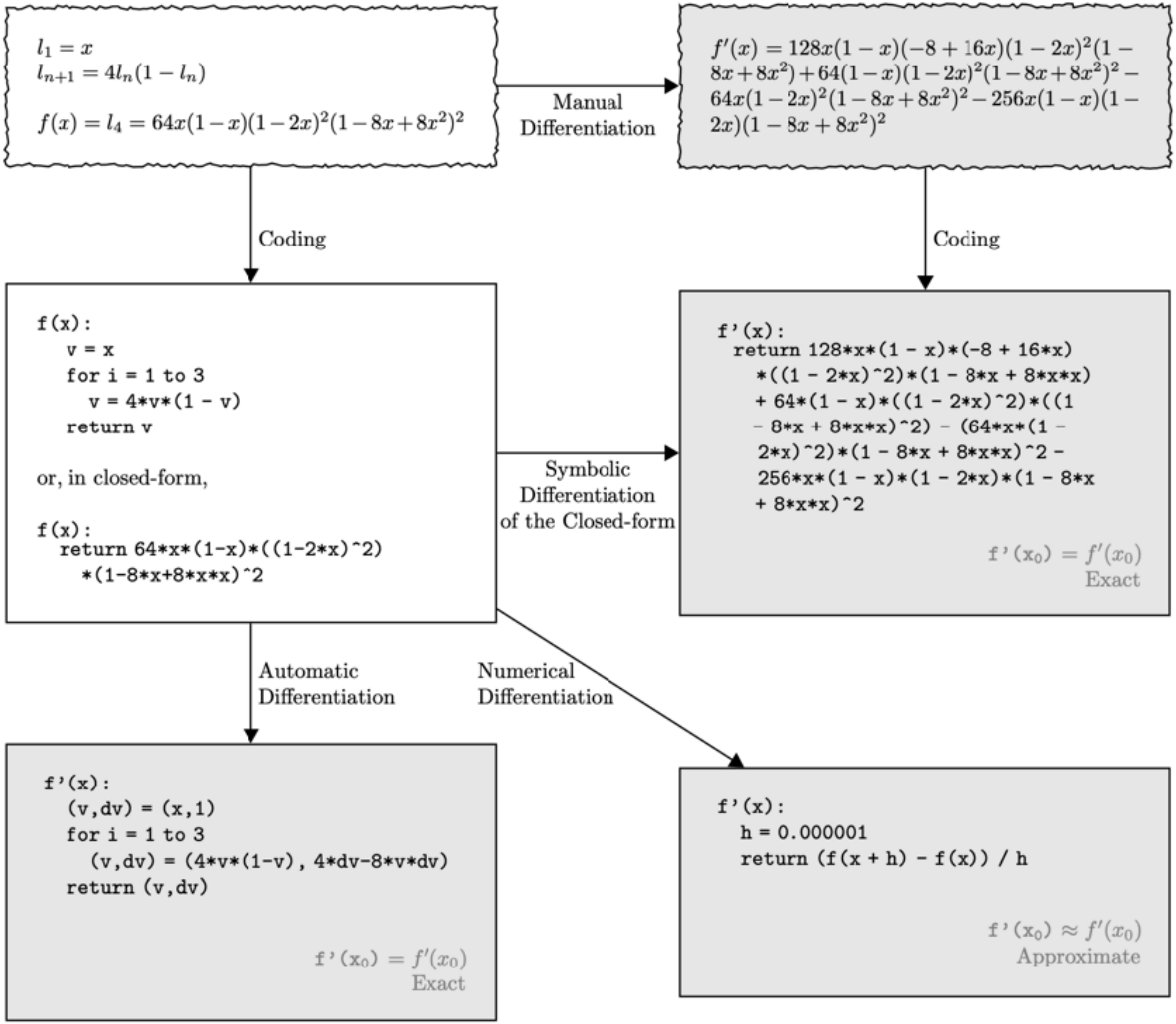
Backpropagation vs AutoDiff

Backpropagation



AutoDiff





$l_1 = x$
$l_{n+1} = 4l_n(1 - l_n)$
$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$
Manual Differentiation
$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$
Coding
Coding
f(x):
v = x
for i = 1 to 3
v = 4*v*(1 - v)
return v
or, in closed-form,
f(x):
return 64*x*(1-x)*((1-2*x)^2)
*(1-8*x+8*x*x)^2
Symbolic Differentiation of the Closed-form
f'(x):
return 128*x*(1 - x)*(-8 + 16*x)
*((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
+ 64*(1 - x)*((1 - 2*x)^2)*((1
- 8*x + 8*x*x)^2) - (64*x*(1 -
2*x)^2)*(1 - 8*x + 8*x*x)^2 -
256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
+ 8*x*x)^2
f'(x0) = $f'(x_0)$
Exact
Automatic Differentiation
Numerical Differentiation
f'(x):
(v,dv) = (x,1)
for i = 1 to 3
(v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
return (v,dv)
f'(x0) = $f'(x_0)$
Exact
f'(x):
h = 0.000001
return (f(x + h) - f(x)) / h
f'(x0) ≈ $f'(x_0)$
Approximate

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/