

Lecture 6: Backpropagation

Reminder: A2

- Use SGD to train linear classifiers and fully-connected networks
- Today's lecture can help you compute derivatives in A2
- Due Friday January 28, 11:59pm ET

Autograder

- A1: 10 submissions / day
- A2: 5 submissions / day
- A3+: 3 submissions / day

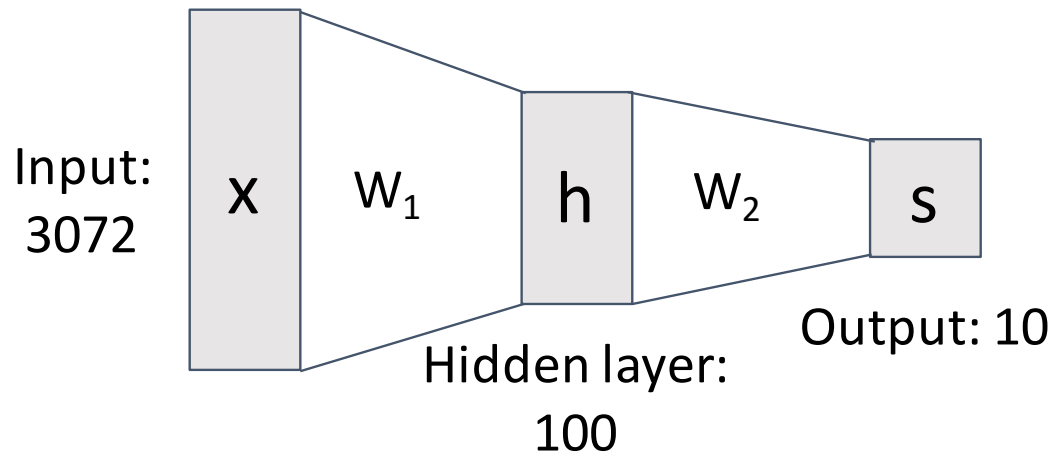
Autograder is a val set, not a training set! You shouldn't be using autograder to debug; you should be testing your code on the side

Extra incentive to start early

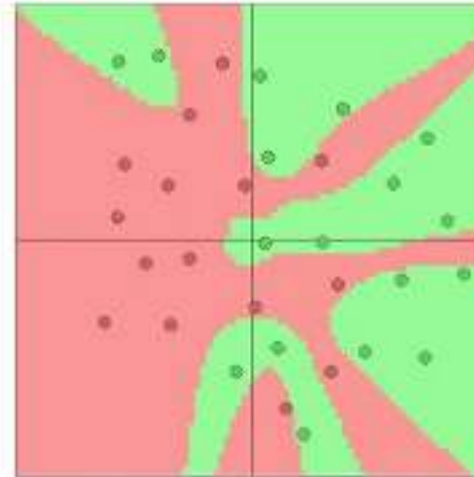
Last time: Neural Networks

From linear classifiers to
fully-connected networks

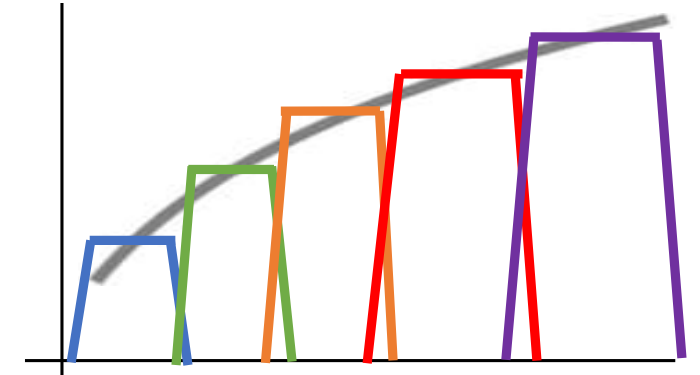
$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



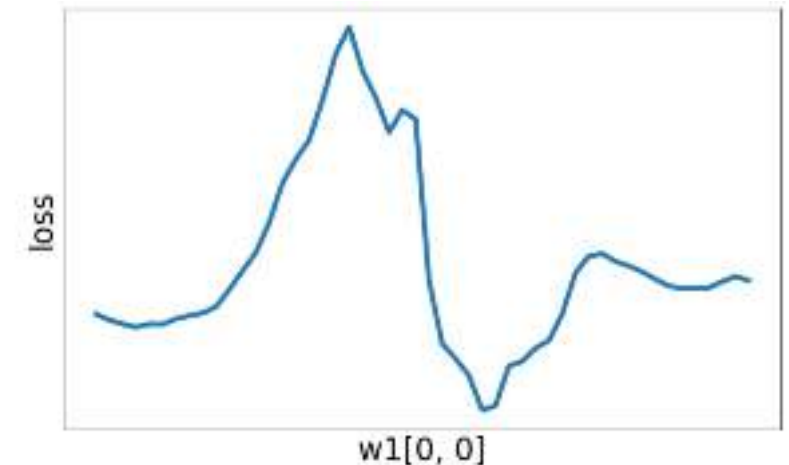
Space Warping



Universal Approximation



Nonconvex



Problem: How to compute gradients?

$$s = W_2 \max(0, W_1 x + b_1) + b_2$$

Nonlinear score function

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Per-element data loss

$$R(W) = \sum_k W_k^2$$

L2 Regularization

$$L(W_1, W_2, b_1, b_2) = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W_1) + \lambda R(W_2)$$

Total loss

If we can compute $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial b_2}$ then we can optimize with SGD

(Bad) Idea: Derive $\nabla_W L$ on paper

$$s = f(x; W) = Wx$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

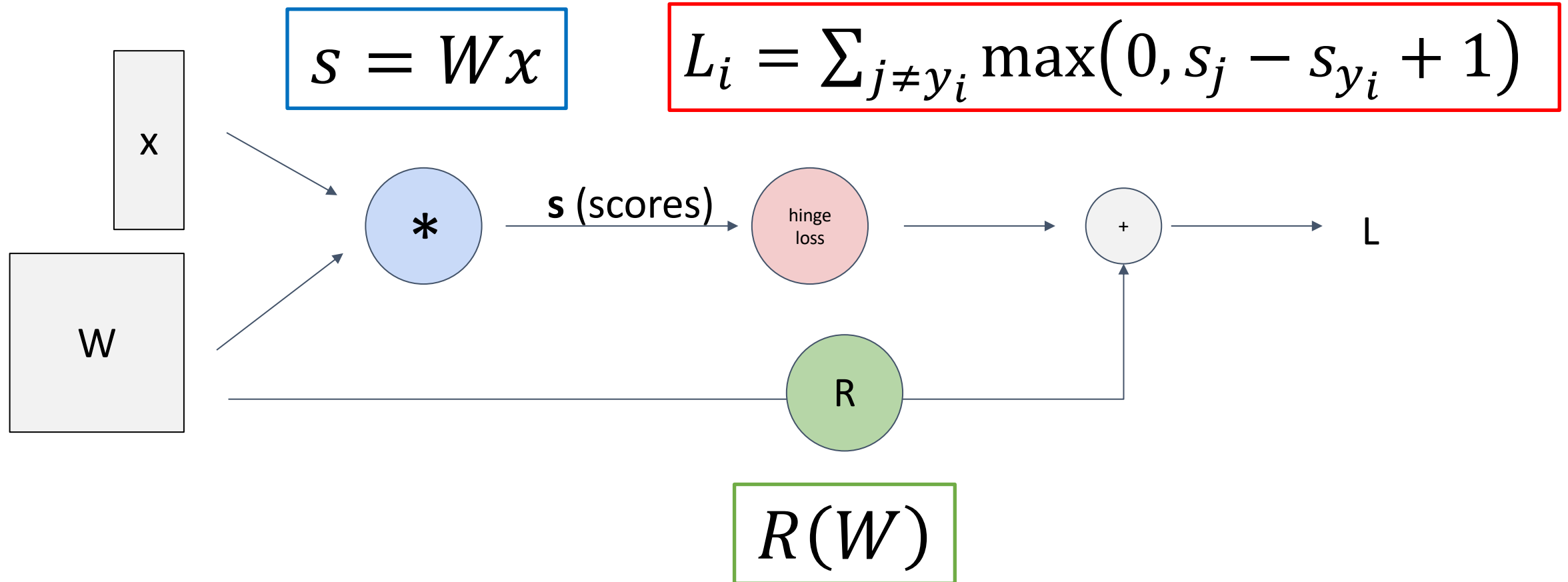
$$\nabla_W L = \nabla_W \left(\frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

Problem: Very tedious: Lots of matrix calculus, need lots of paper

Problem: What if we want to change loss? E.g. use softmax instead of SVM? Need to re-derive from scratch. Not modular!

Problem: Not feasible for very complex models!

Better Idea: Computational Graphs

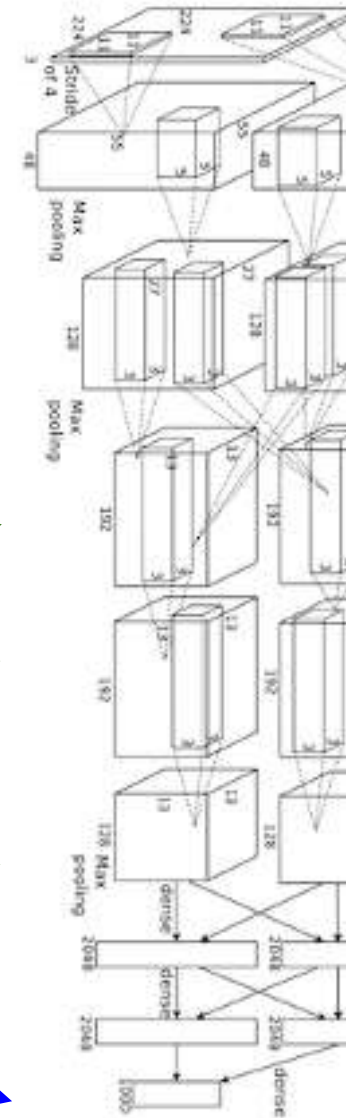


Deep Network (AlexNet)

input image

weights

loss



Neural Turing Machine

input image

loss

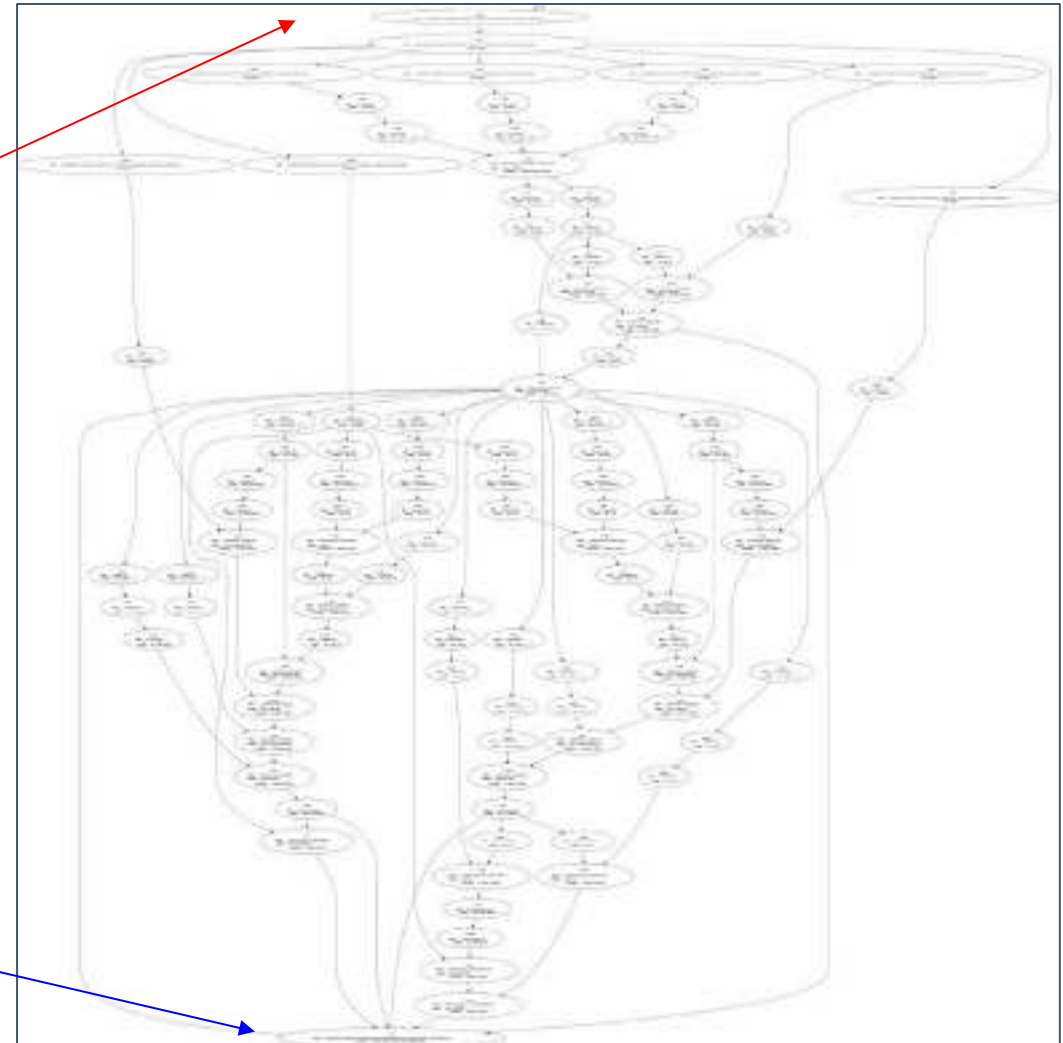
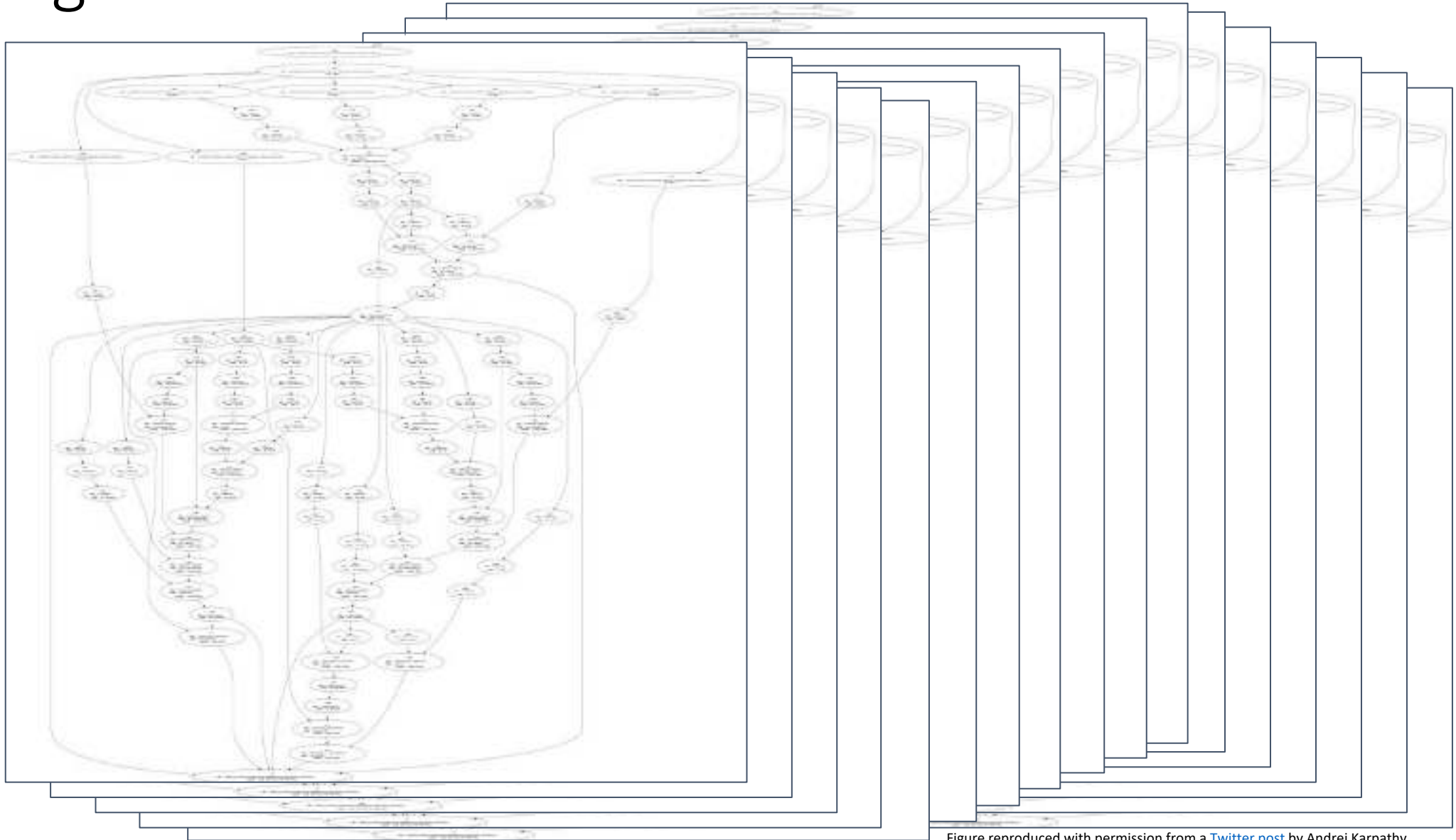


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

Neural Turing Machine

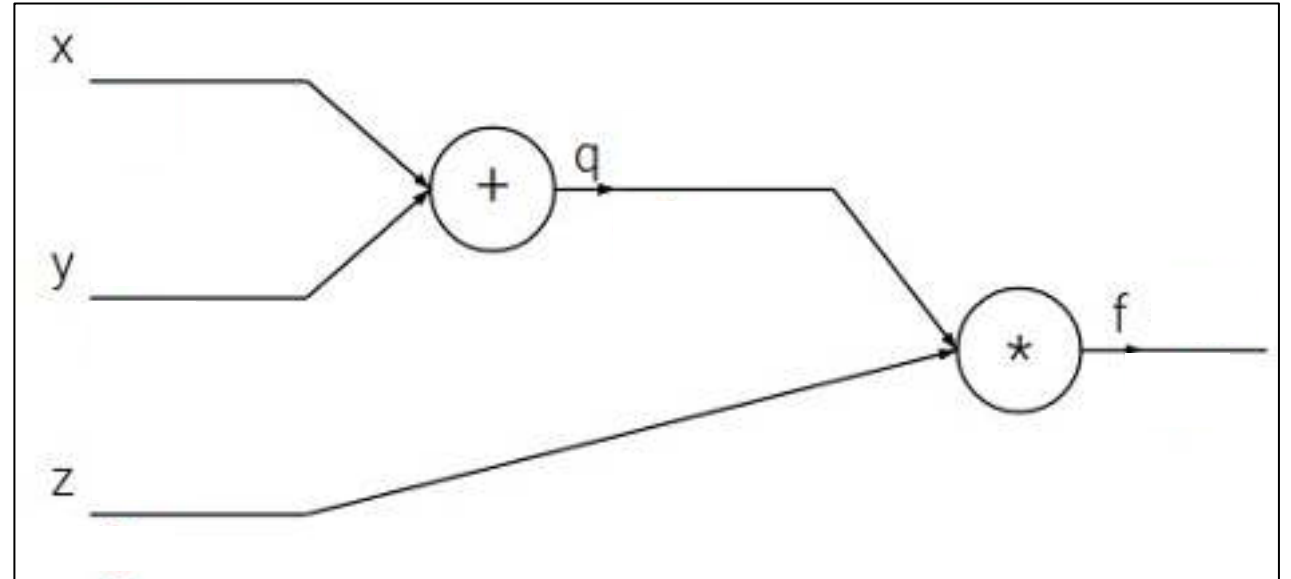


Graves et al, arXiv 2014

Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

Backpropagation: Simple Example

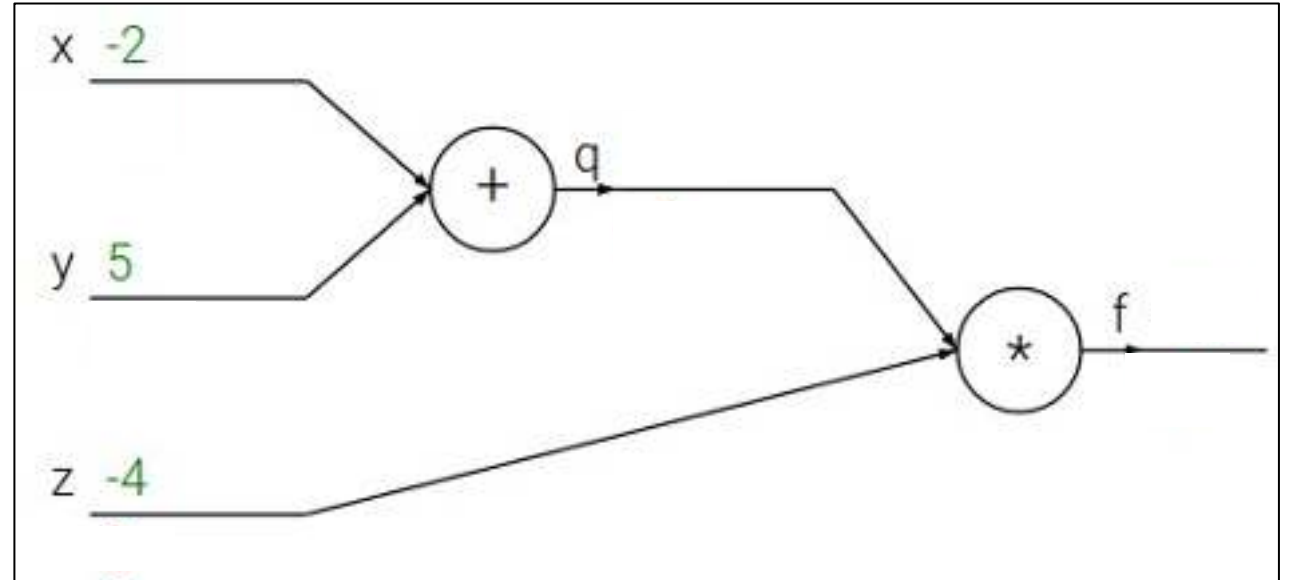
$$f(x, y, z) = (x + y) \cdot z$$



Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



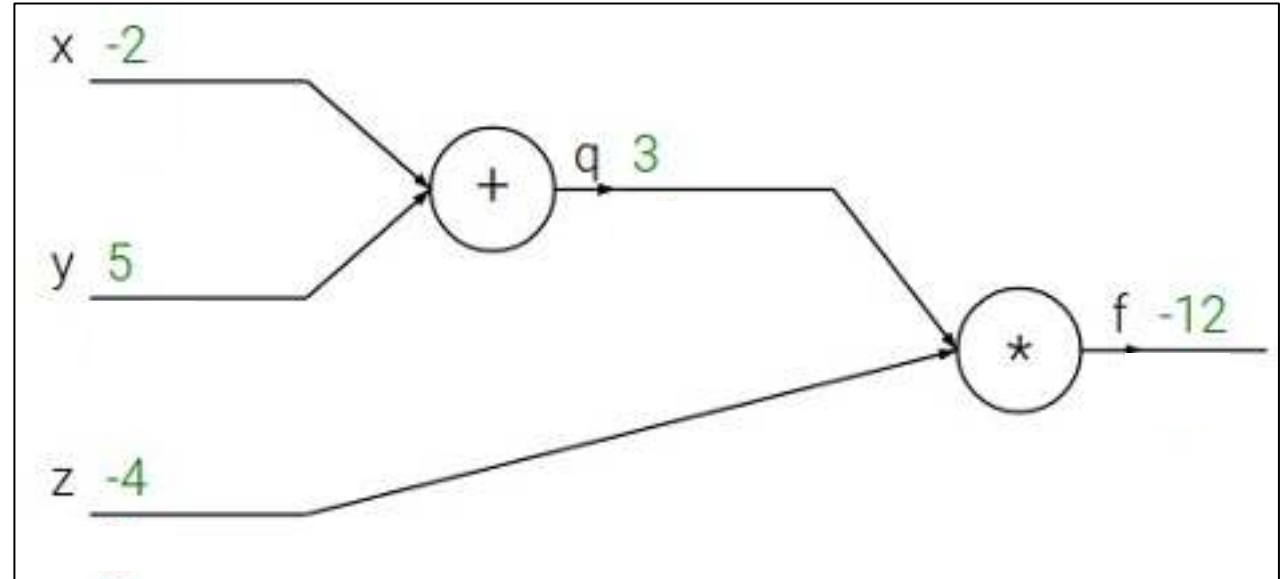
Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$

1. Forward pass: Compute outputs

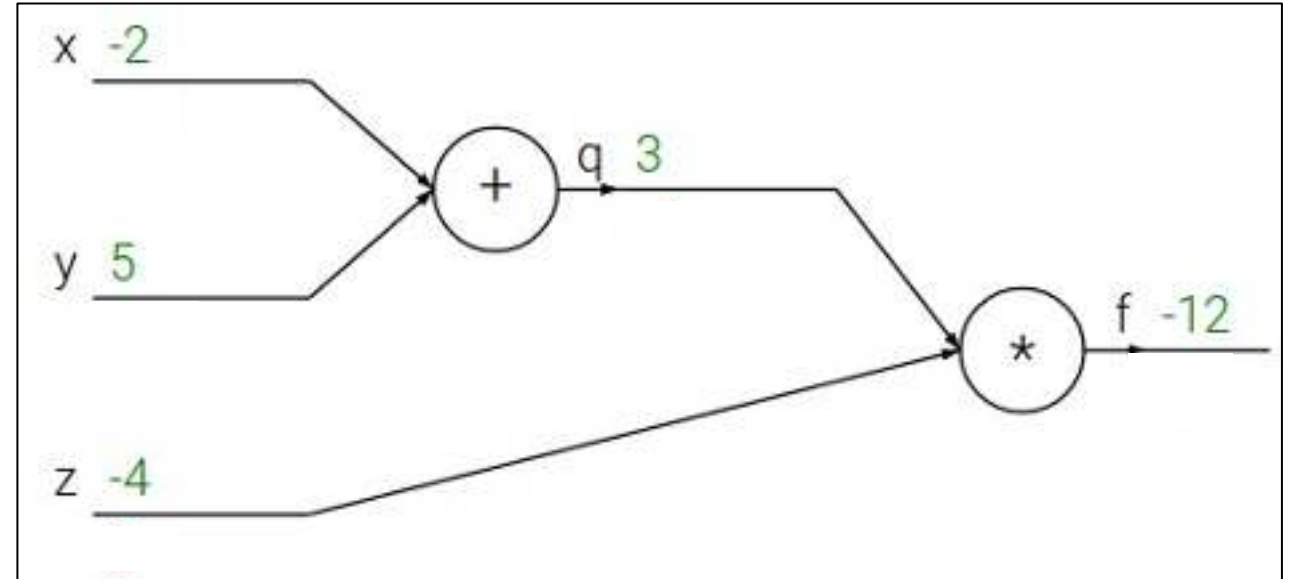
$$q = x + y \quad f = q \cdot z$$



Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



1. Forward pass: Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

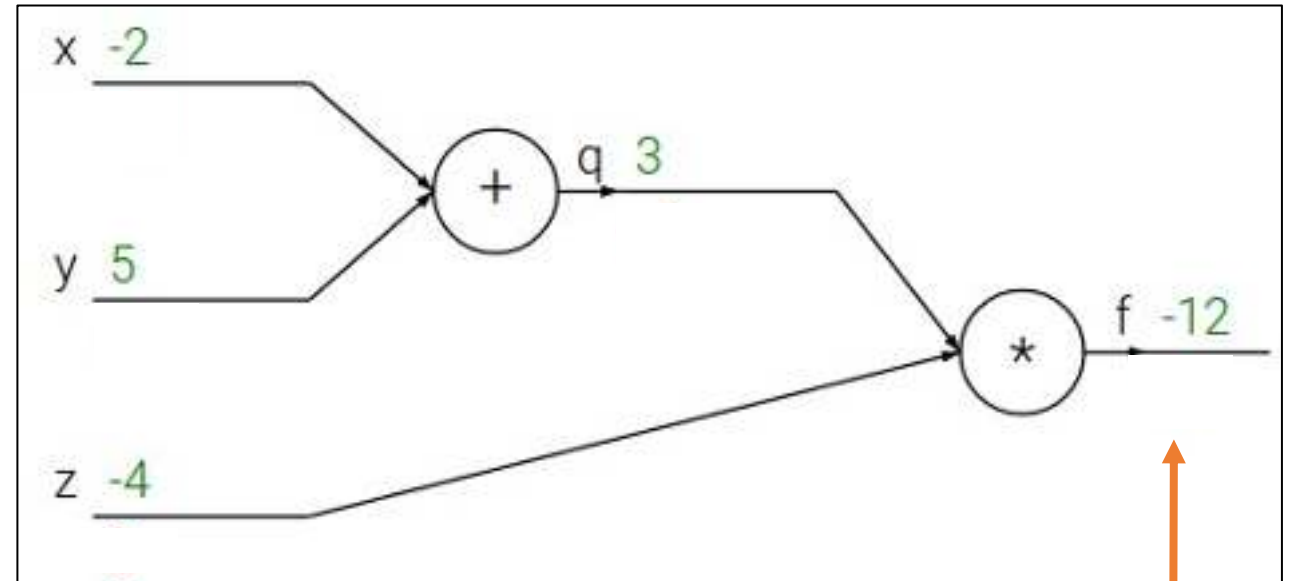
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial f}$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

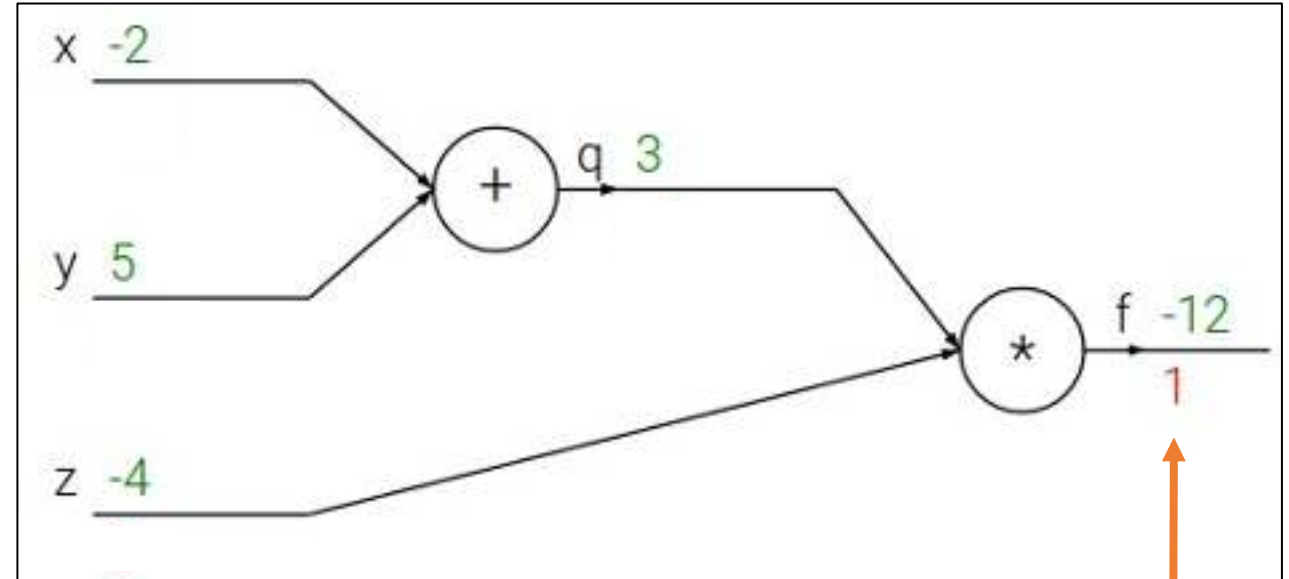
e.g. $x = -2, y = 5, z = -4$

1. Forward pass: Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial f}$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

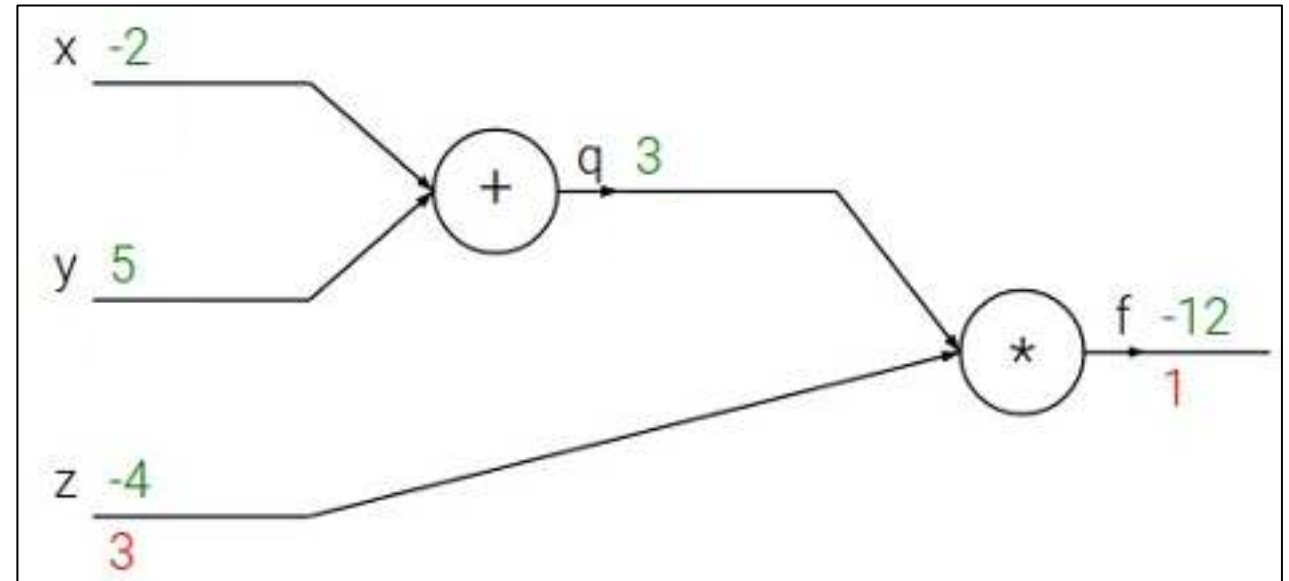
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial z}$$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

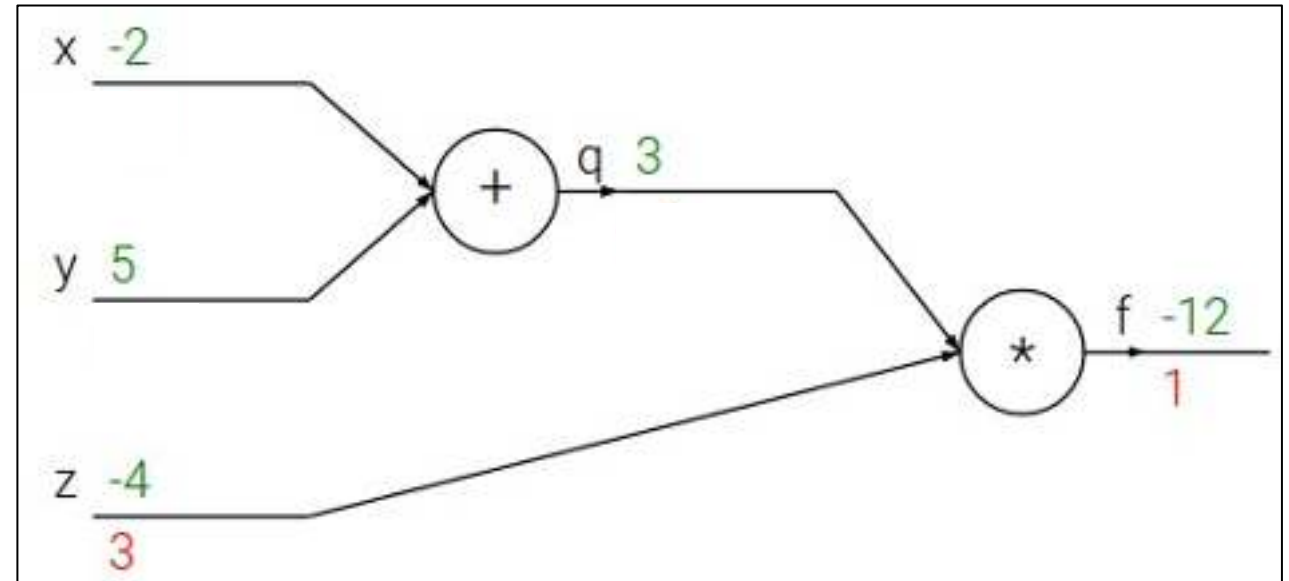
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad \boxed{f = q \cdot z}$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\boxed{\frac{\partial f}{\partial z} = q}$$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

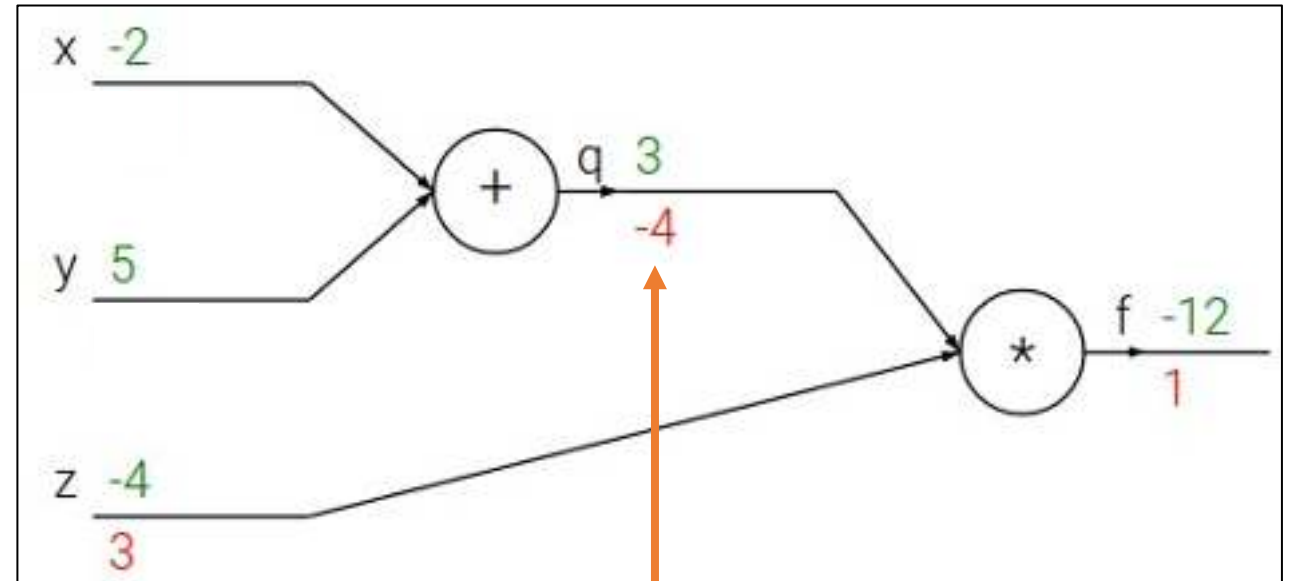
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

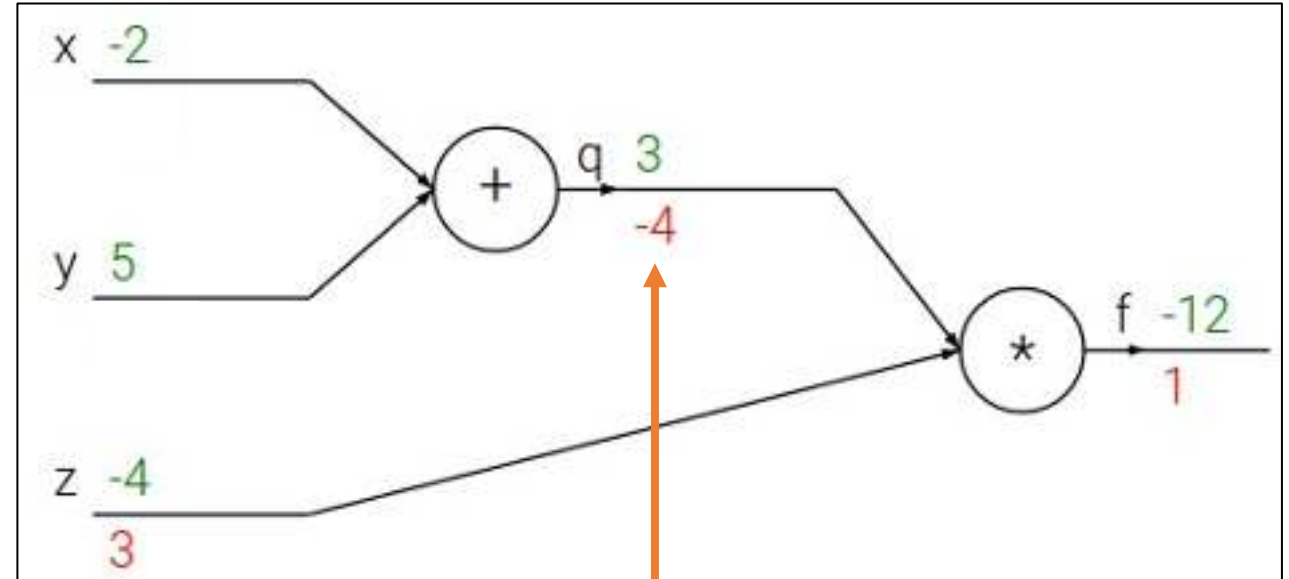
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad \boxed{f = q \cdot z}$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\boxed{\frac{\partial f}{\partial q} = z}$$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

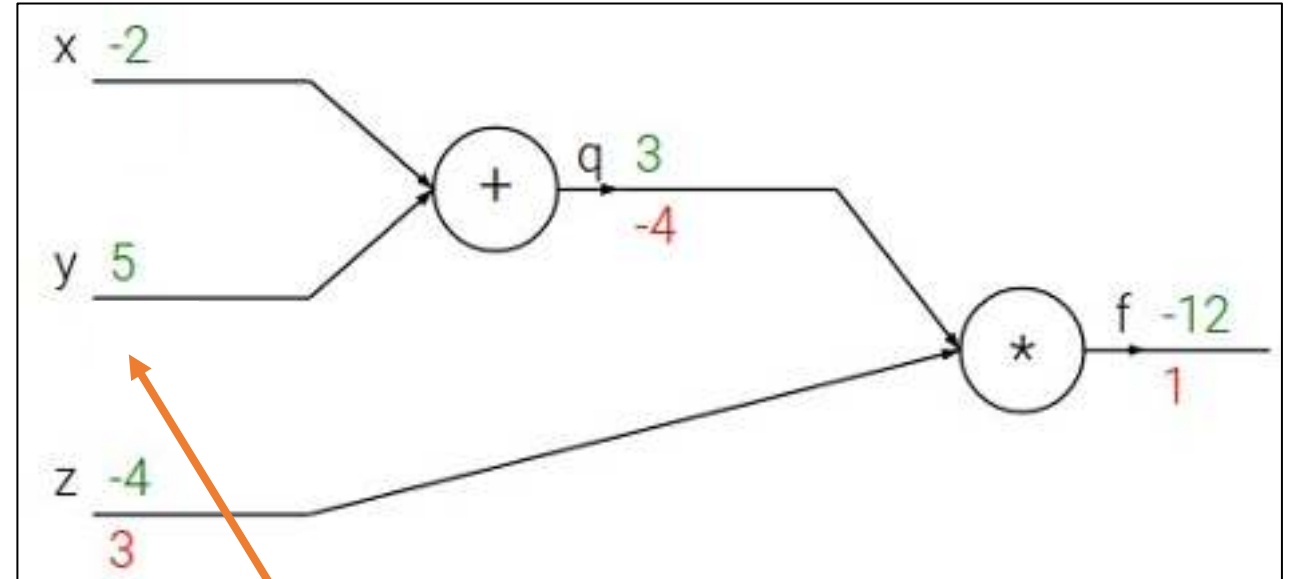
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial y}$$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

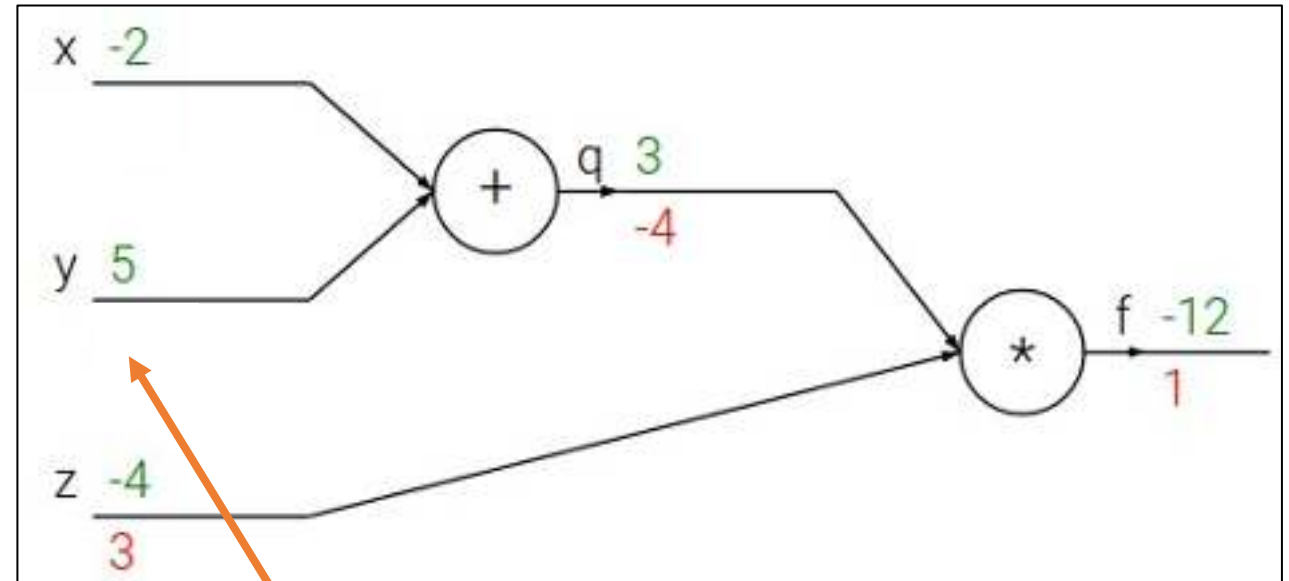
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain Rule

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y} \frac{\partial f}{\partial q}$$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

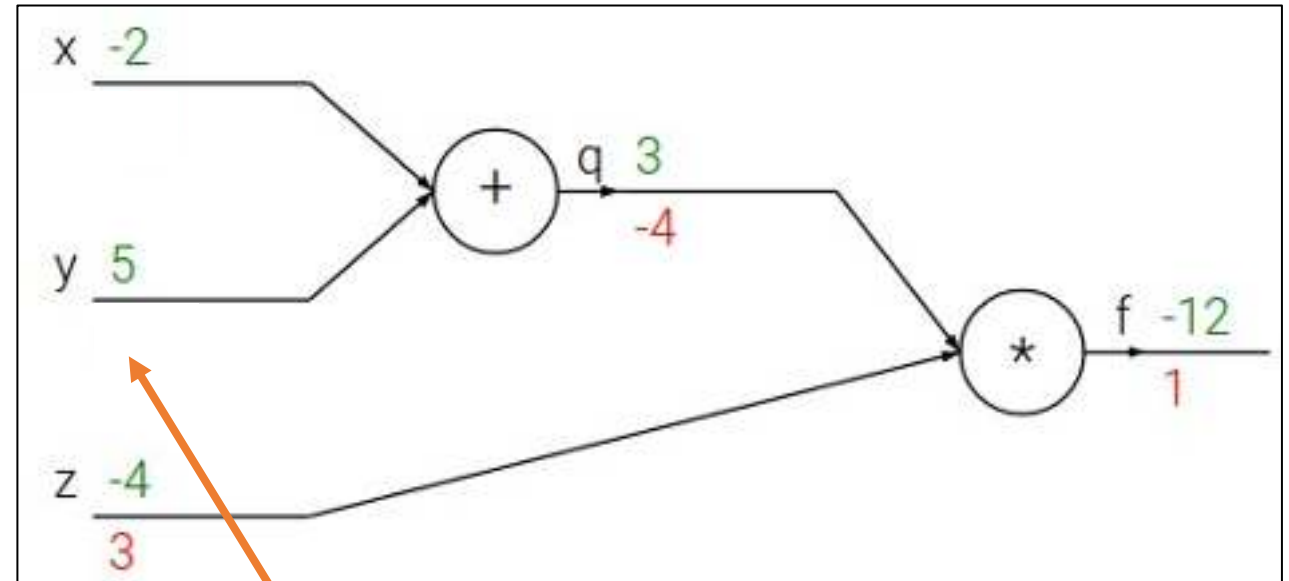
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain Rule

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y} \frac{\partial f}{\partial q}$$

Downstream
Gradient

Local
Gradient

Upstream
Gradient

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

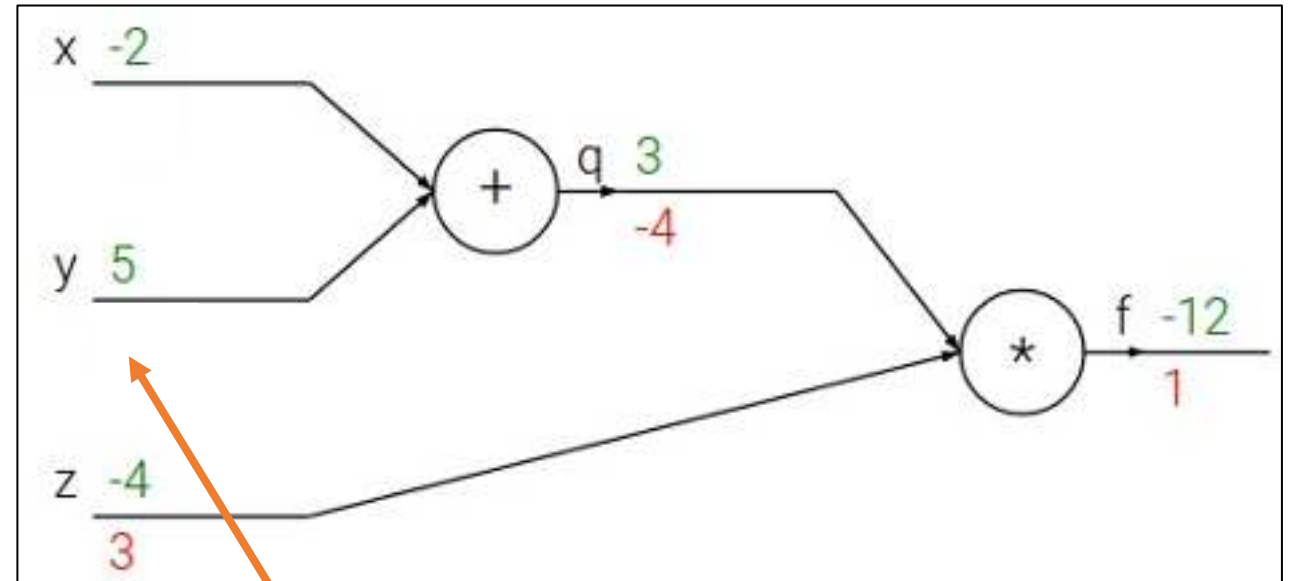
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain Rule

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y} \frac{\partial f}{\partial q}$$

$$\frac{\partial q}{\partial y} = 1$$

Downstream
Gradient

Local
Gradient

Upstream
Gradient

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

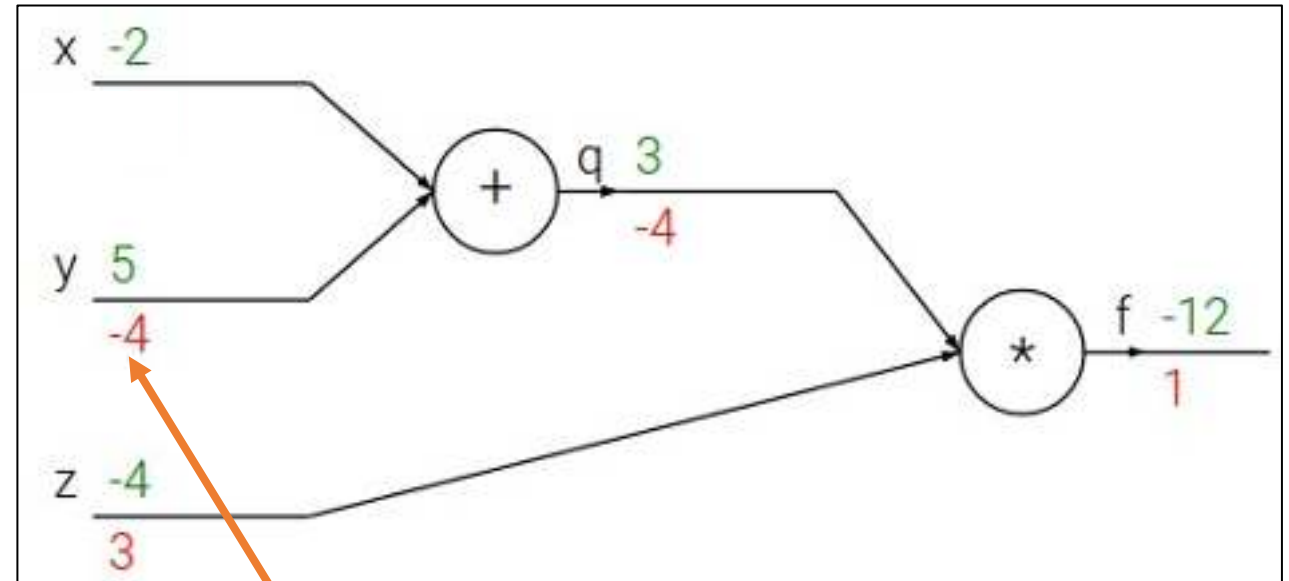
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain Rule

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y} \frac{\partial f}{\partial q}$$

$$\frac{\partial q}{\partial y} = 1$$

Downstream
Gradient

Local
Gradient

Upstream
Gradient

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

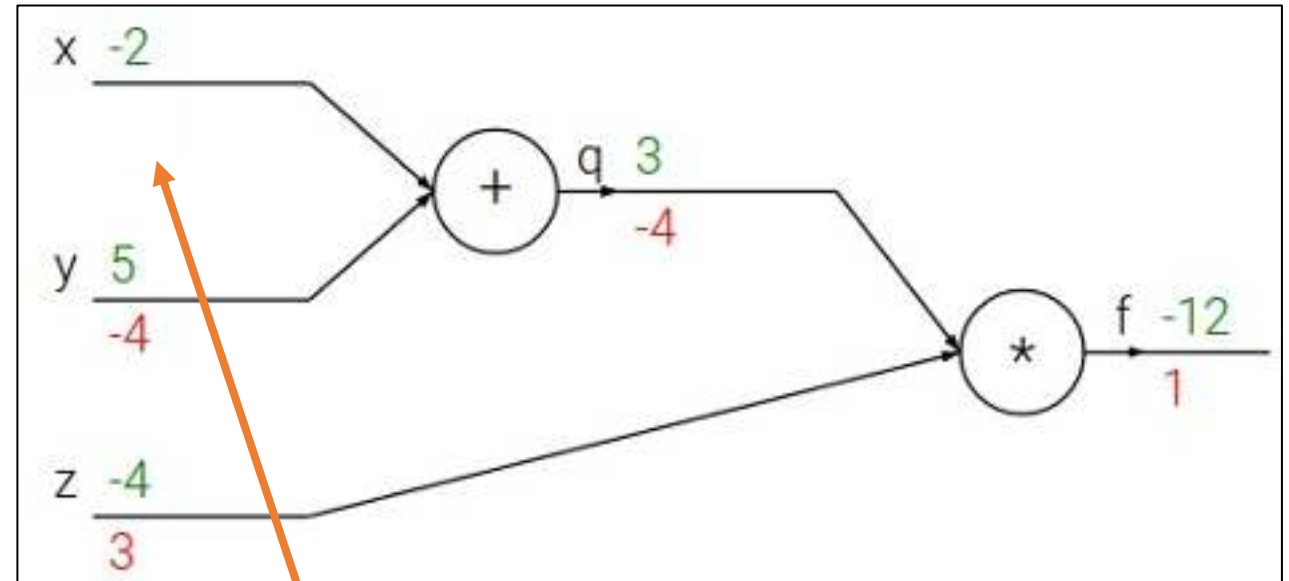
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain Rule

$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x} \frac{\partial f}{\partial q}$$

$$\frac{\partial q}{\partial x} = 1$$

Downstream
Gradient

Local
Gradient

Upstream
Gradient

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

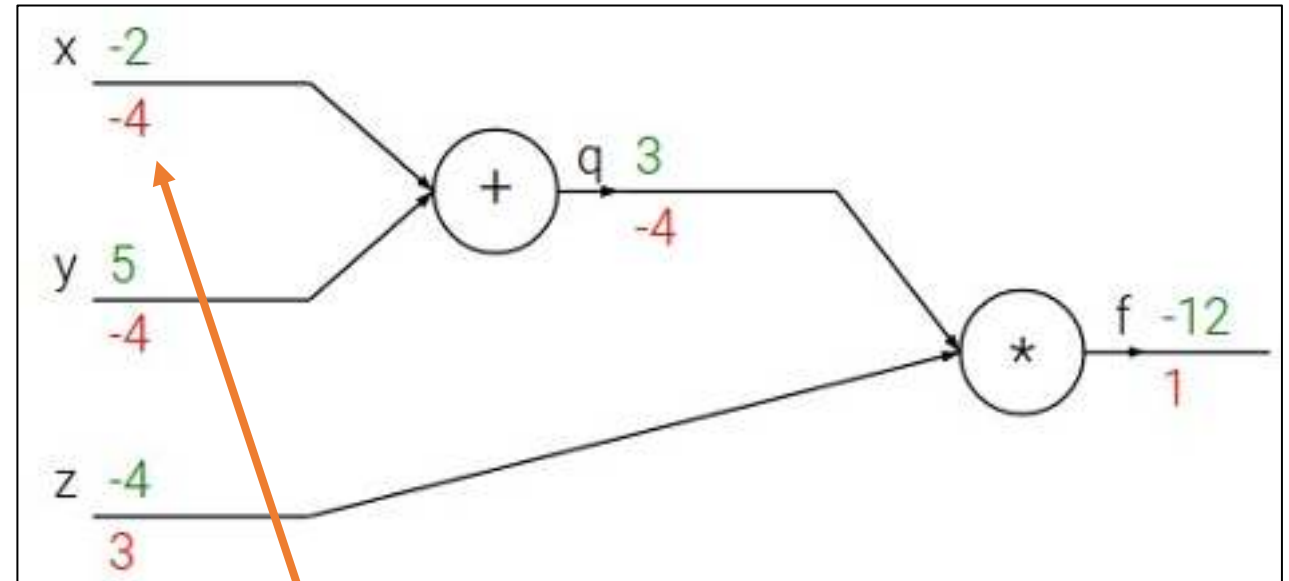
e.g. $x = -2, y = 5, z = -4$

1. **Forward pass:** Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass:** Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain Rule

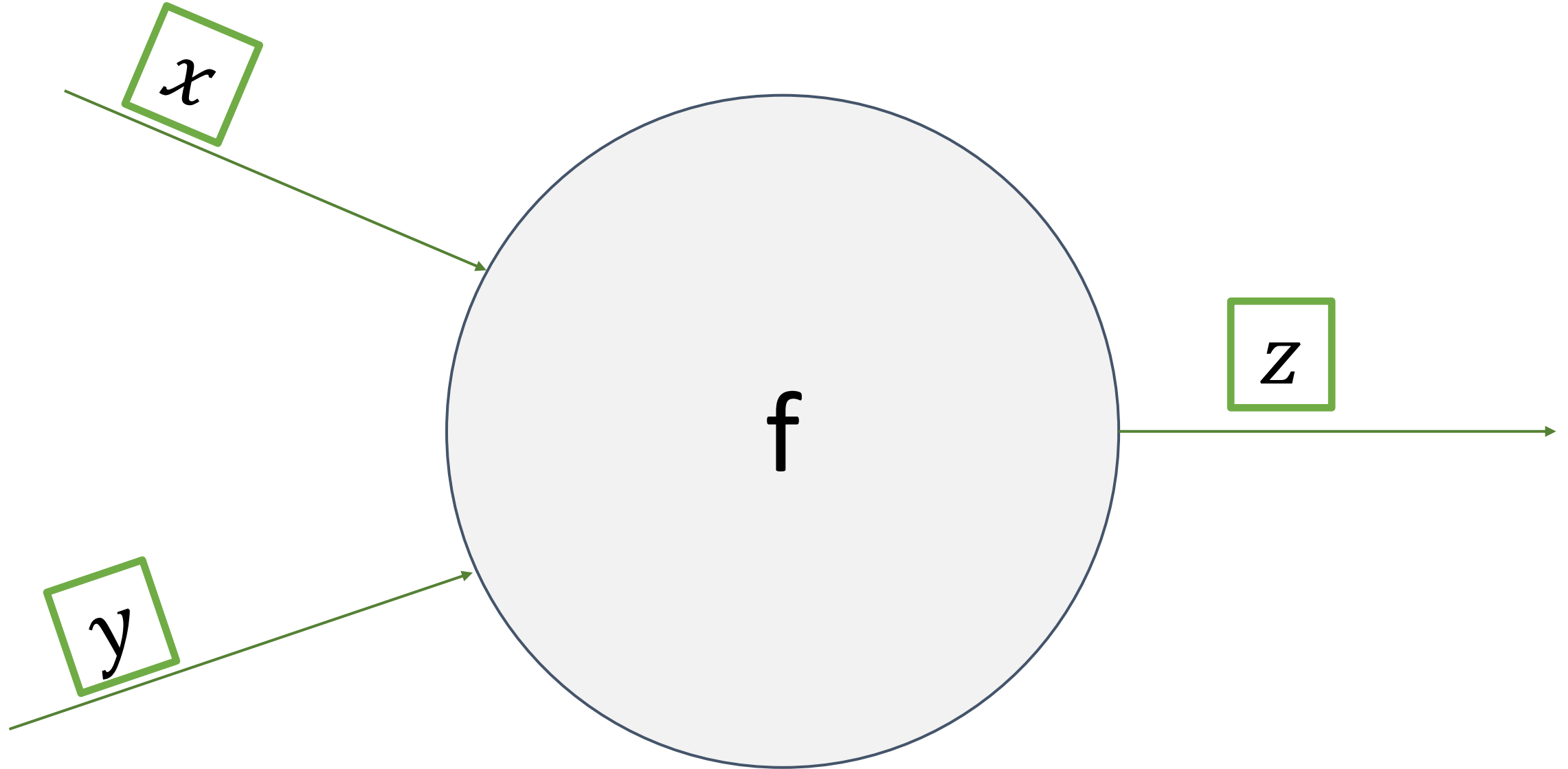
$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x} \frac{\partial f}{\partial q}$$

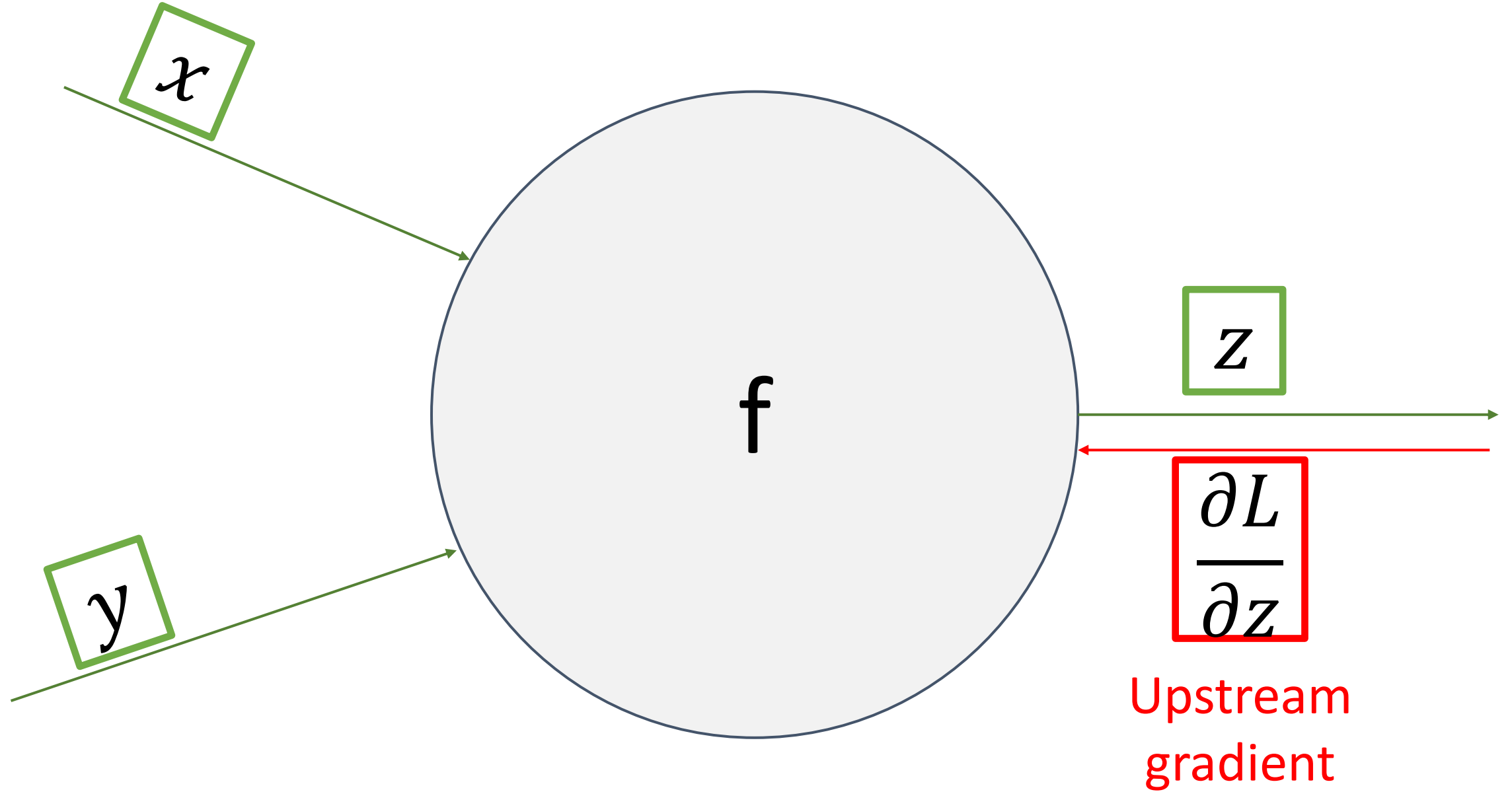
$$\frac{\partial q}{\partial x} = 1$$

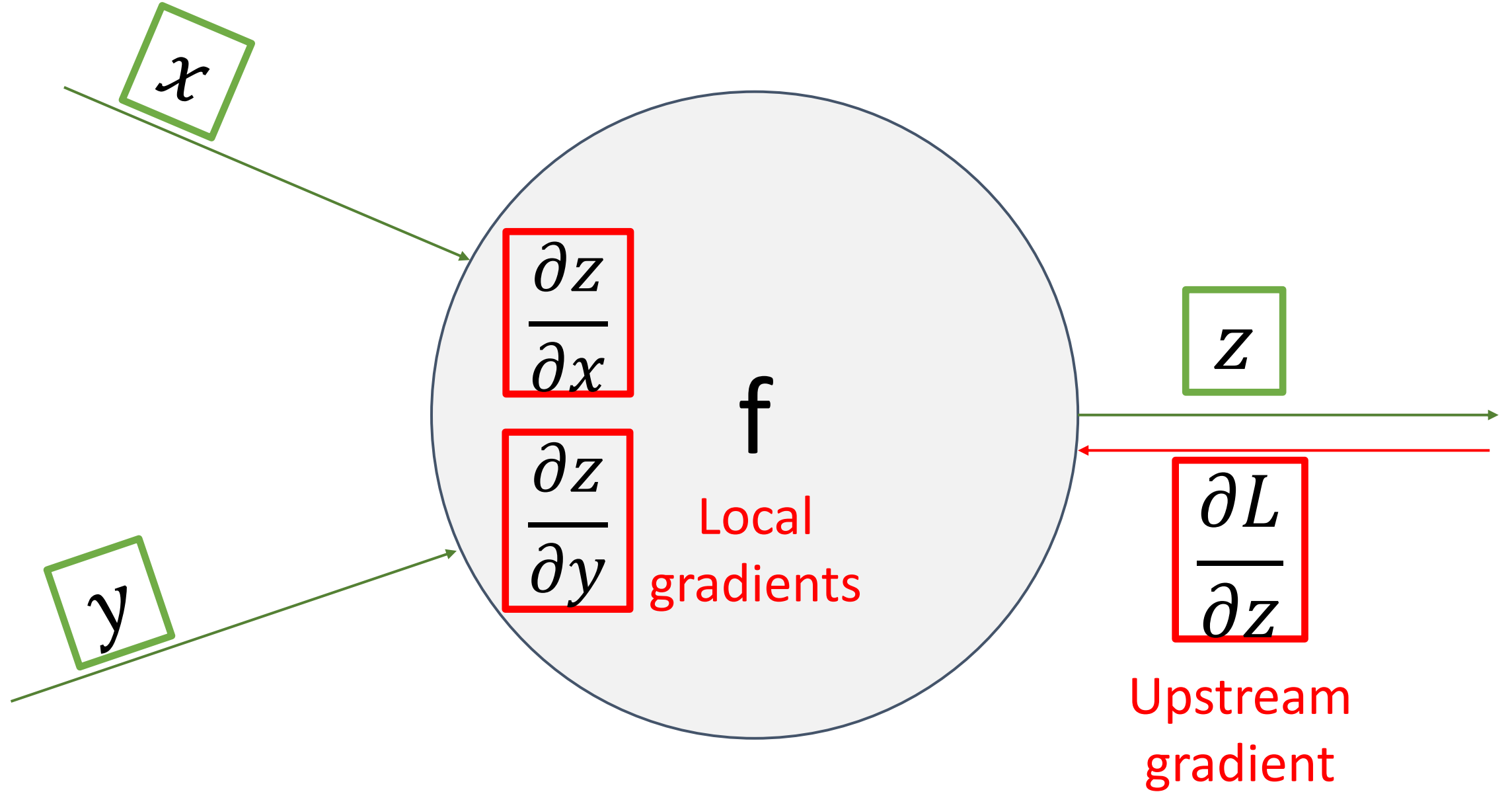
Downstream
Gradient

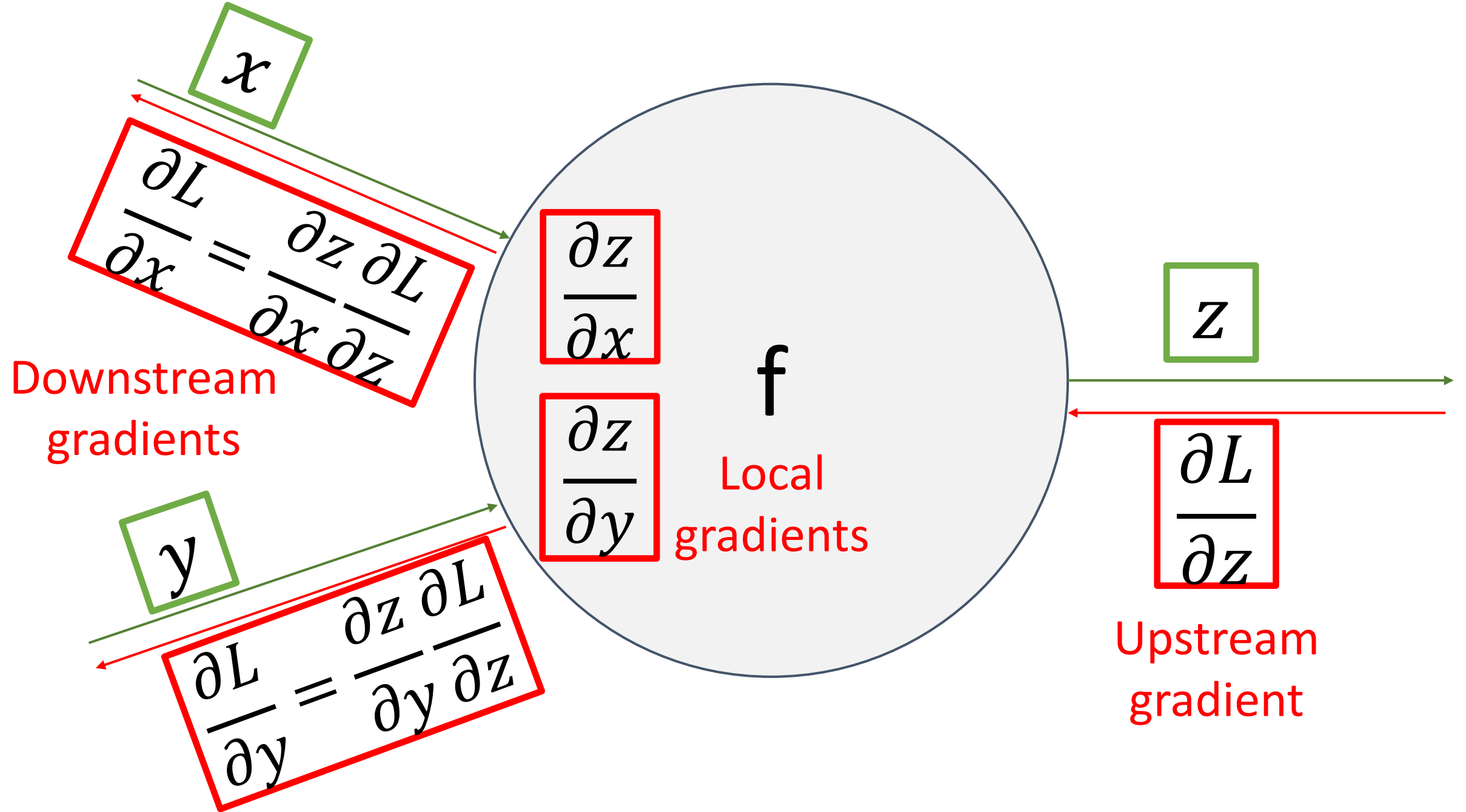
Local
Gradient

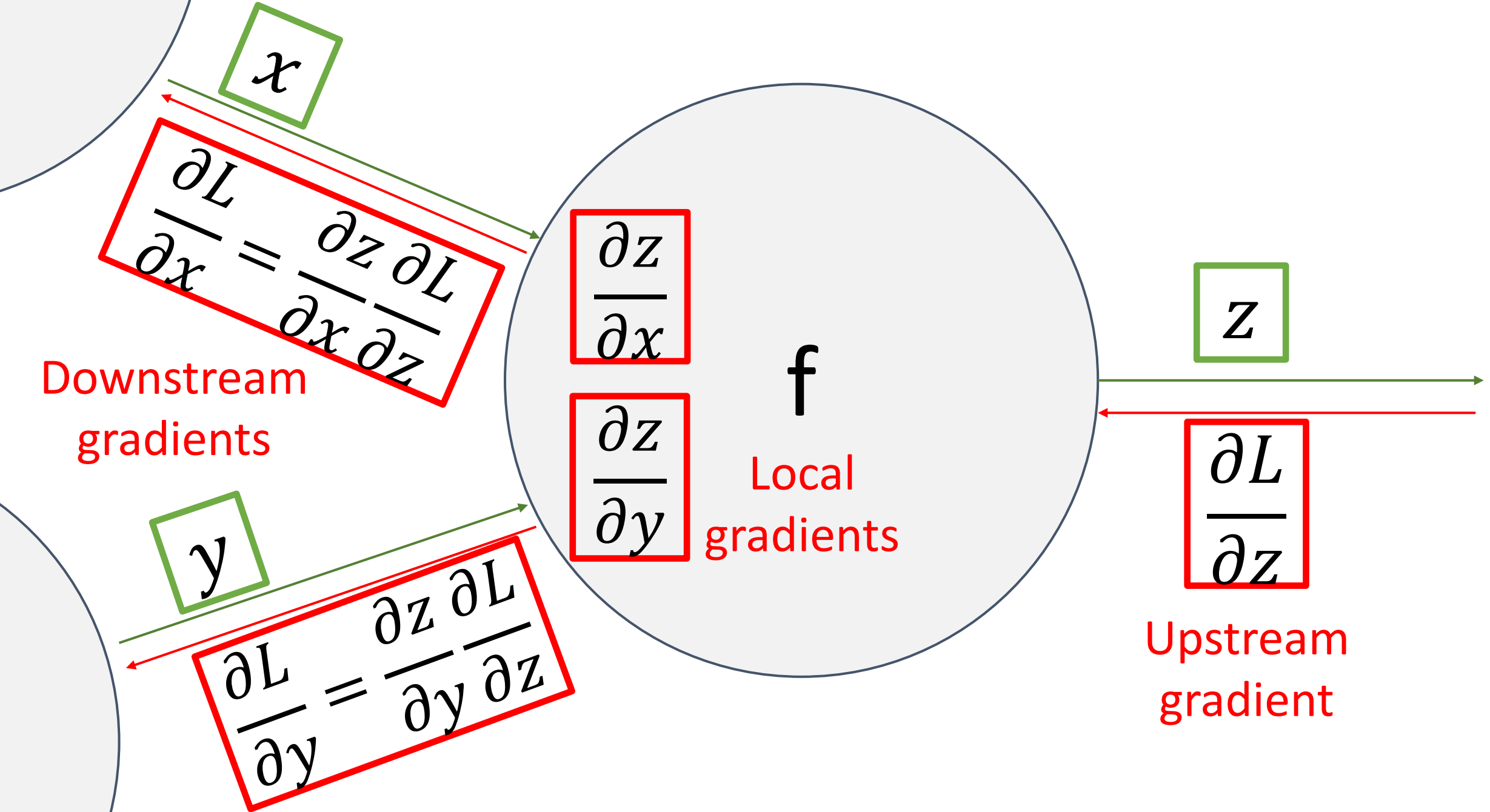
Upstream
Gradient



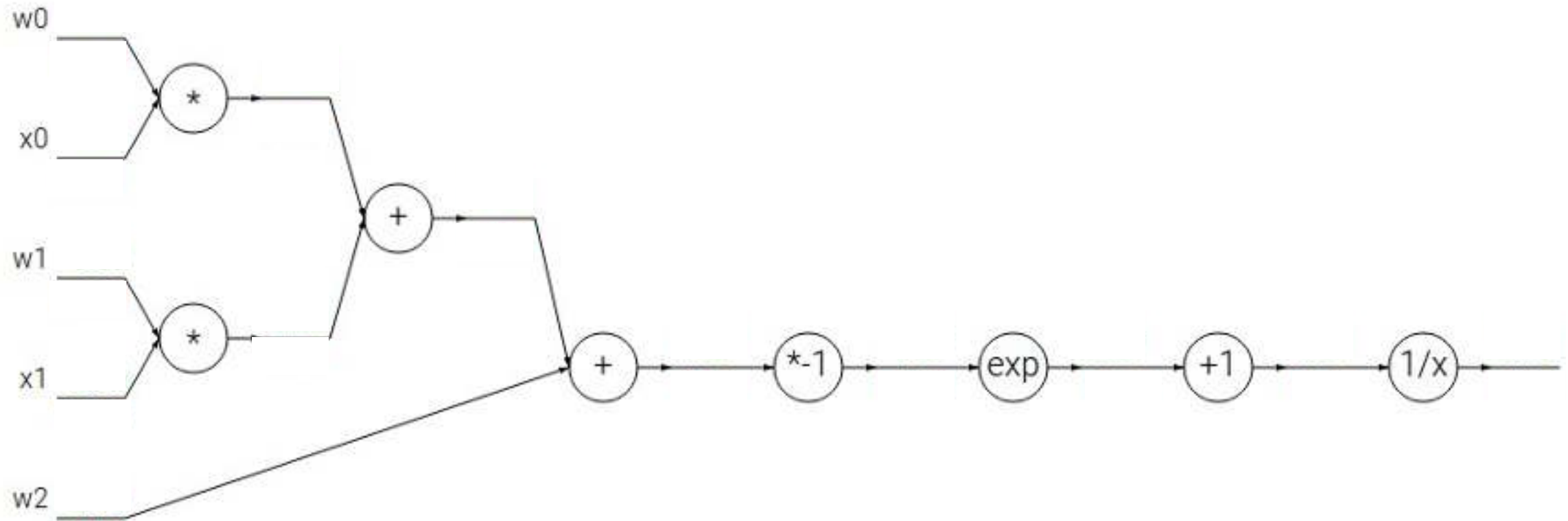






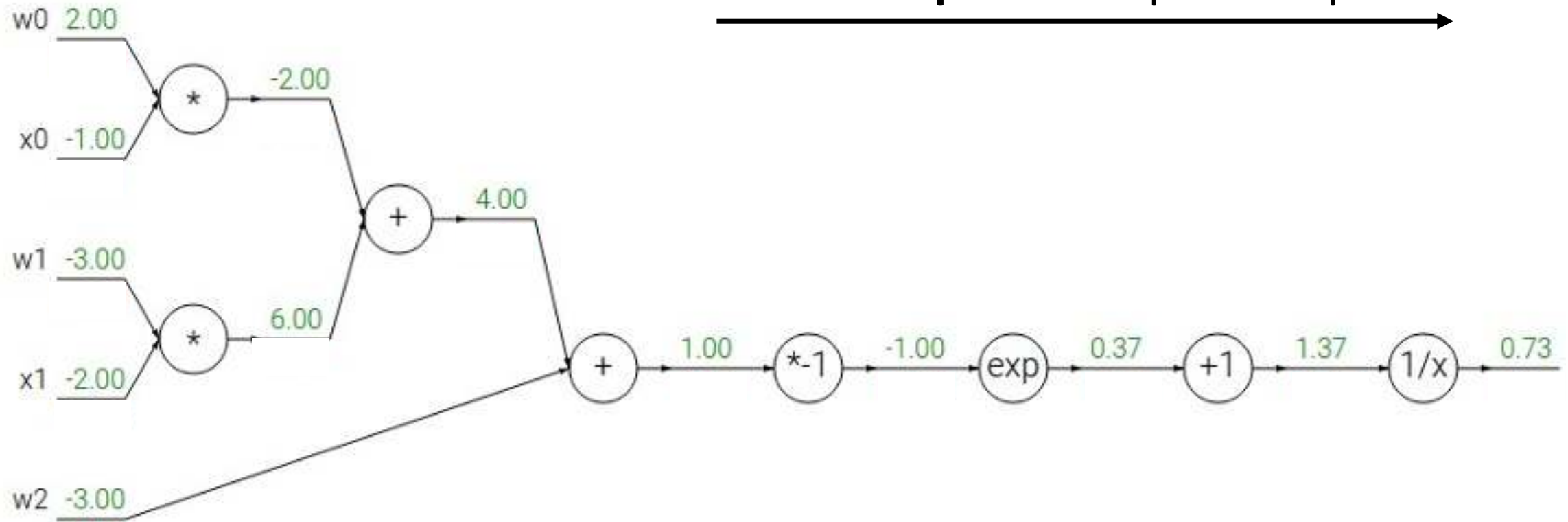


Another Example $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



Another Example $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$

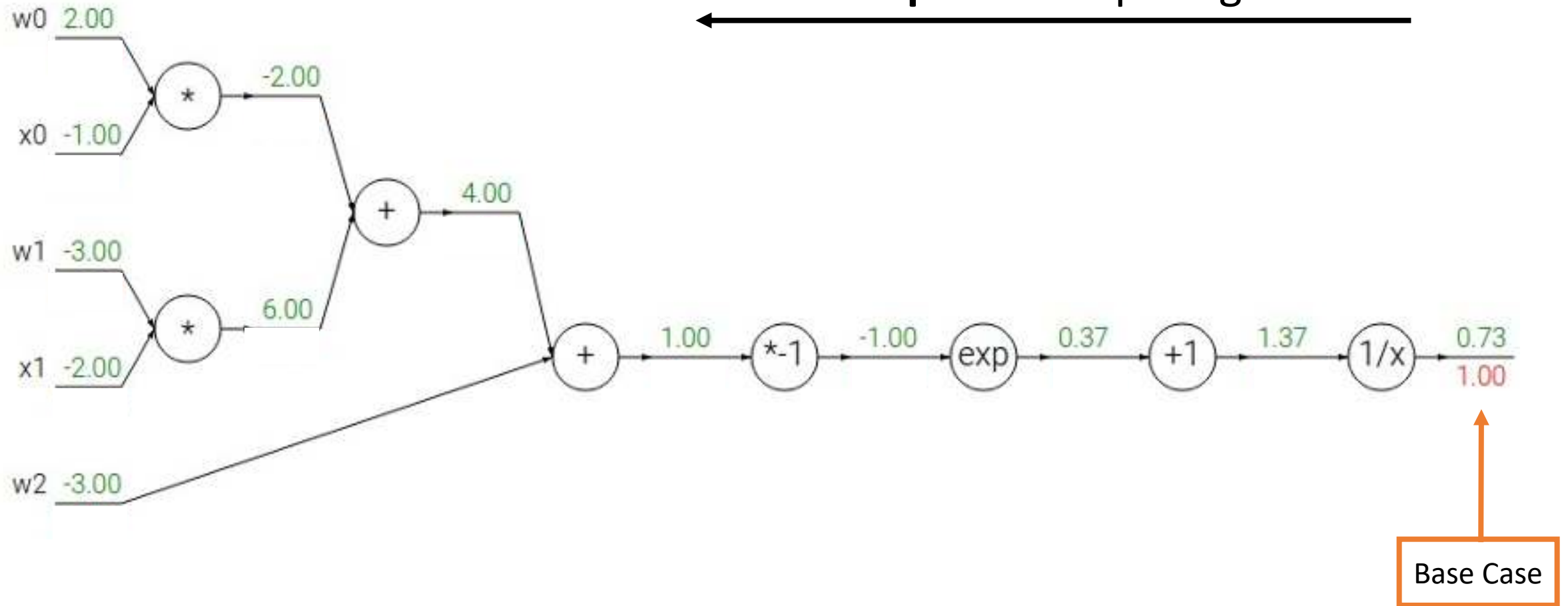
Forward pass: Compute outputs



Another Example

$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

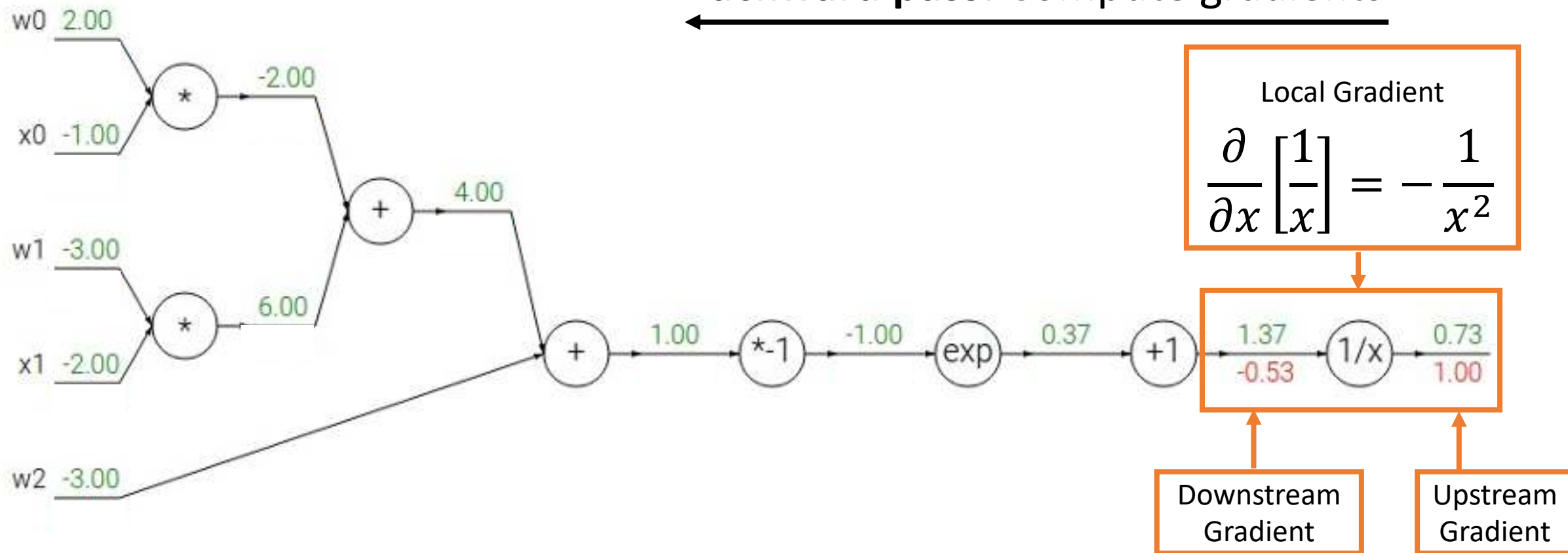
Backward pass: Compute gradients



Another Example

$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

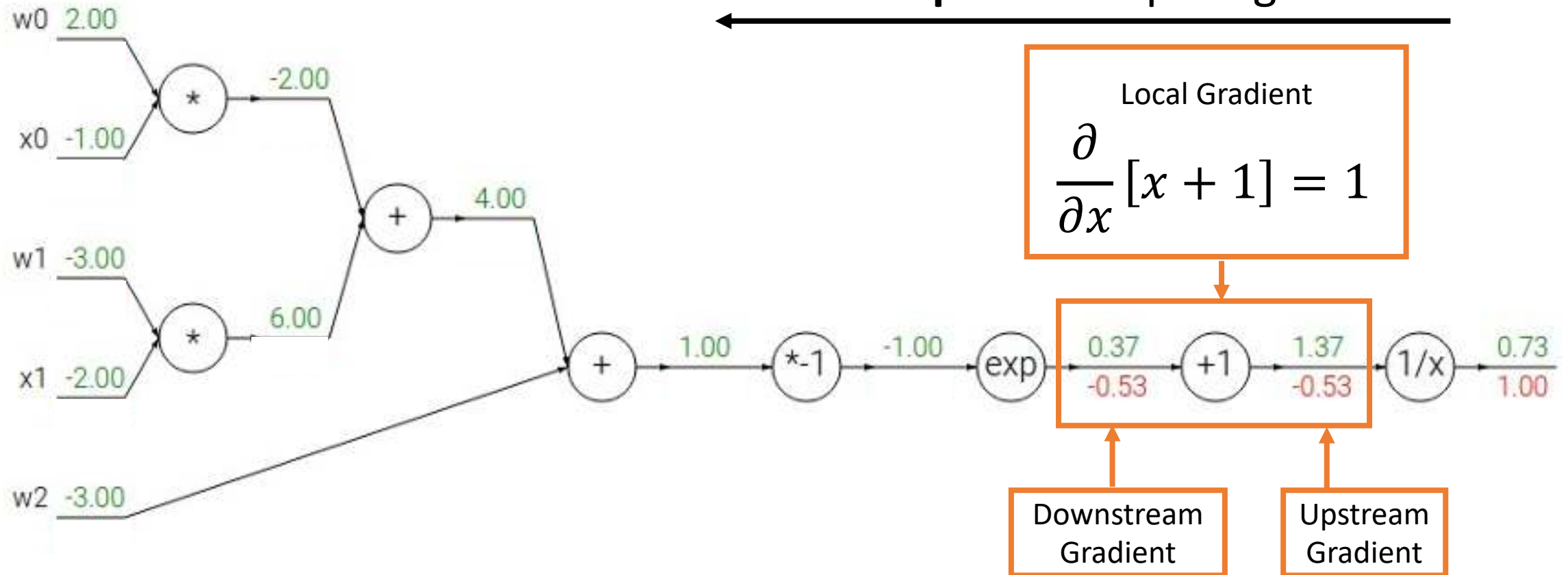
Backward pass: Compute gradients



Another Example

$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

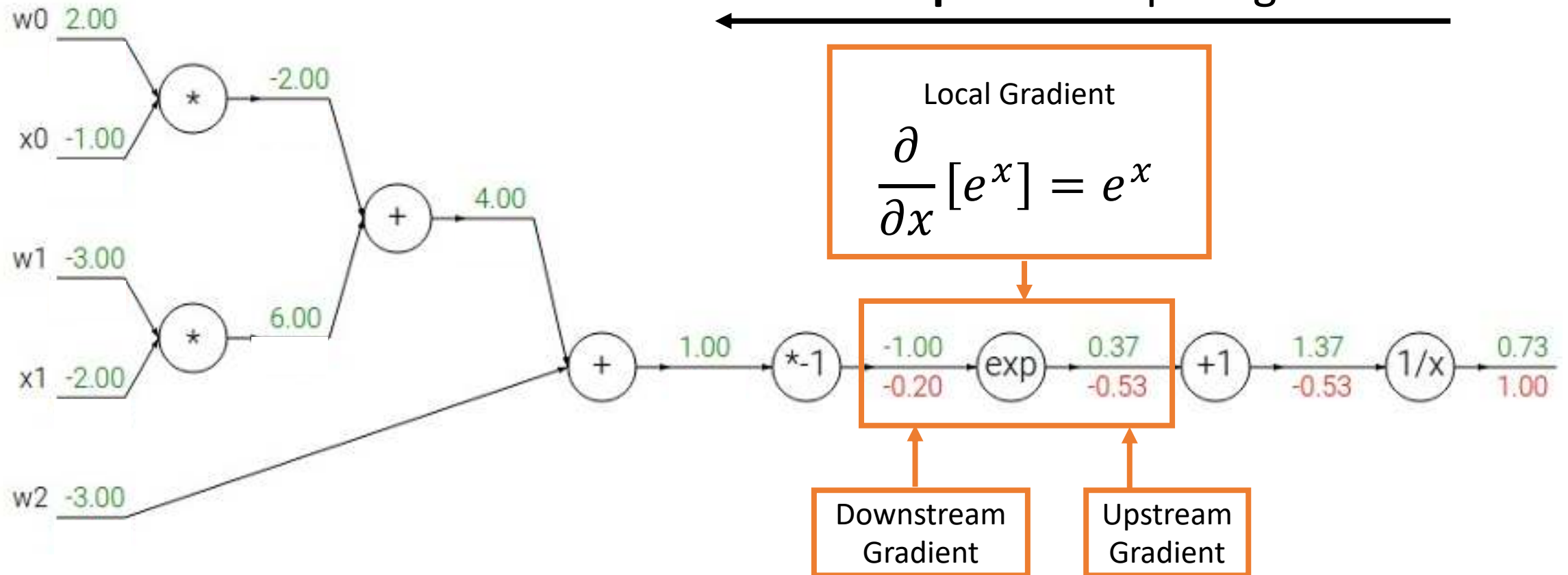
Backward pass: Compute gradients



Another Example

$$f(x, w) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

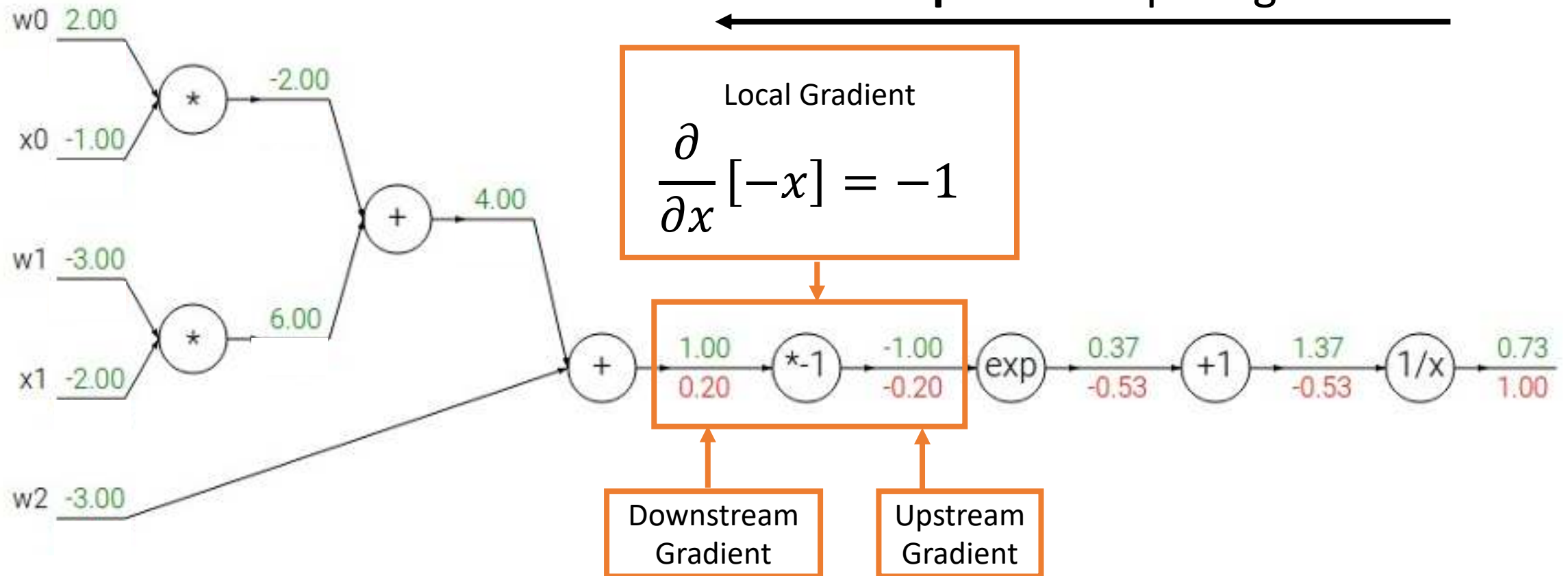
Backward pass: Compute gradients



Another Example

$$f(x, w) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

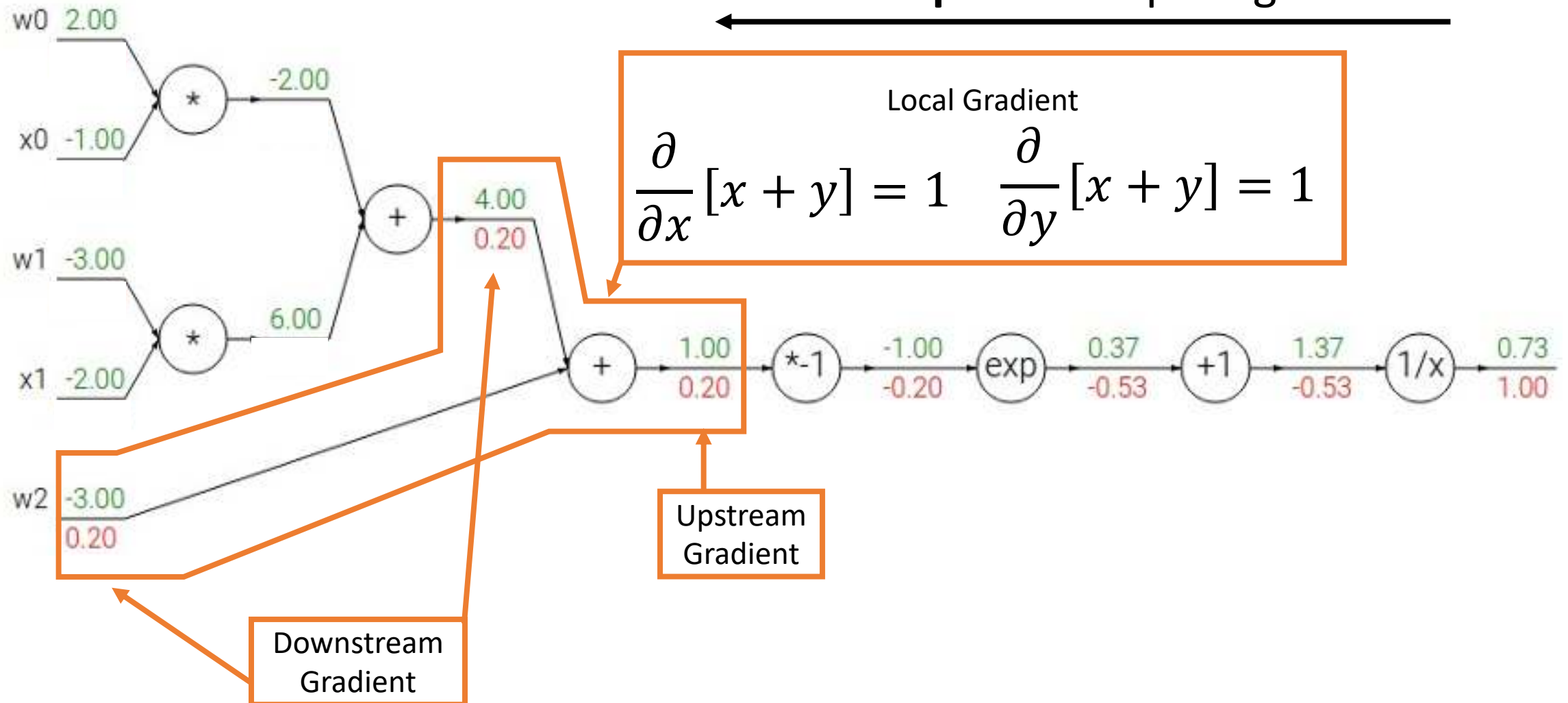
Backward pass: Compute gradients



Another Example

$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

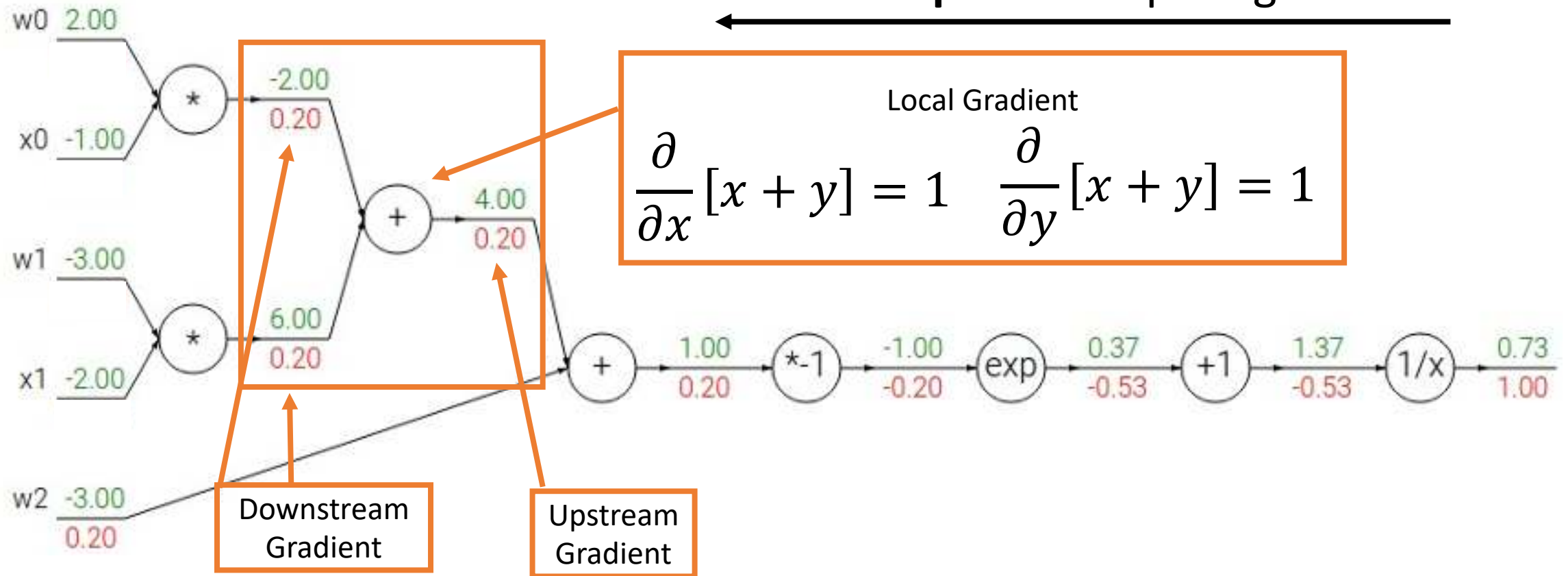
Backward pass: Compute gradients



Another Example

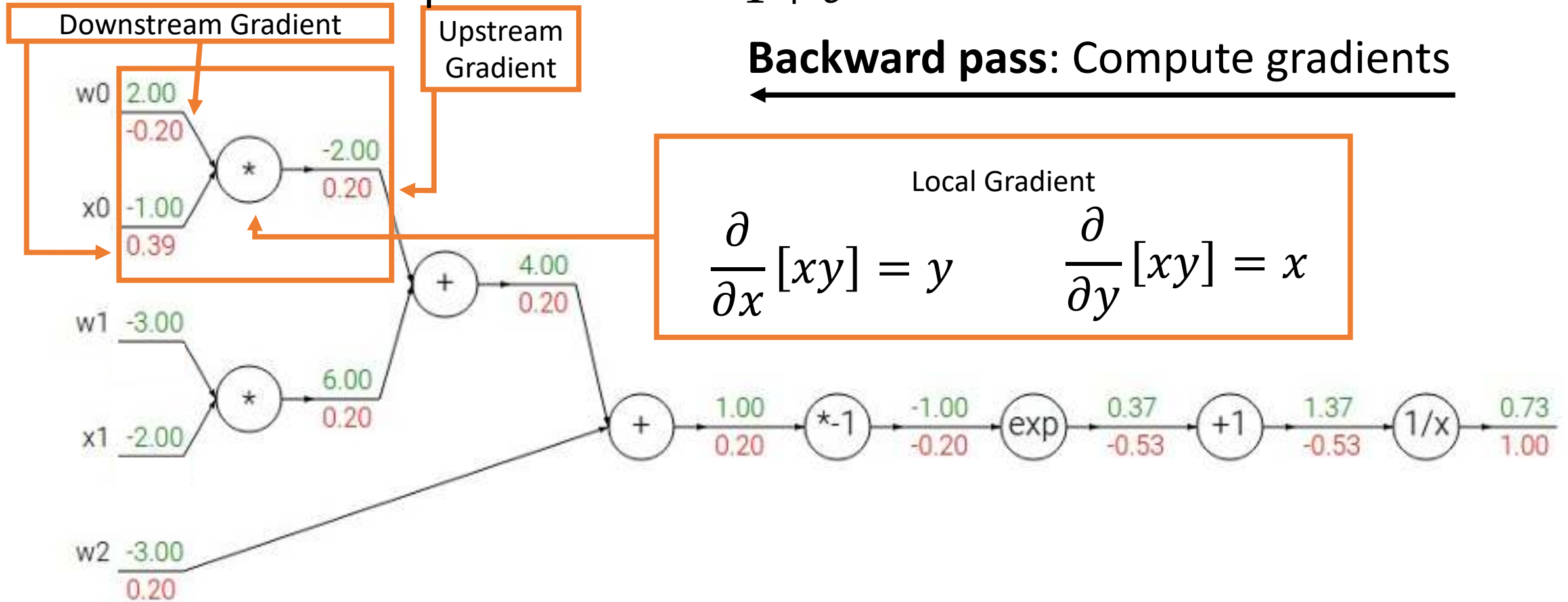
$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}}$$

Backward pass: Compute gradients



Another Example $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}}$

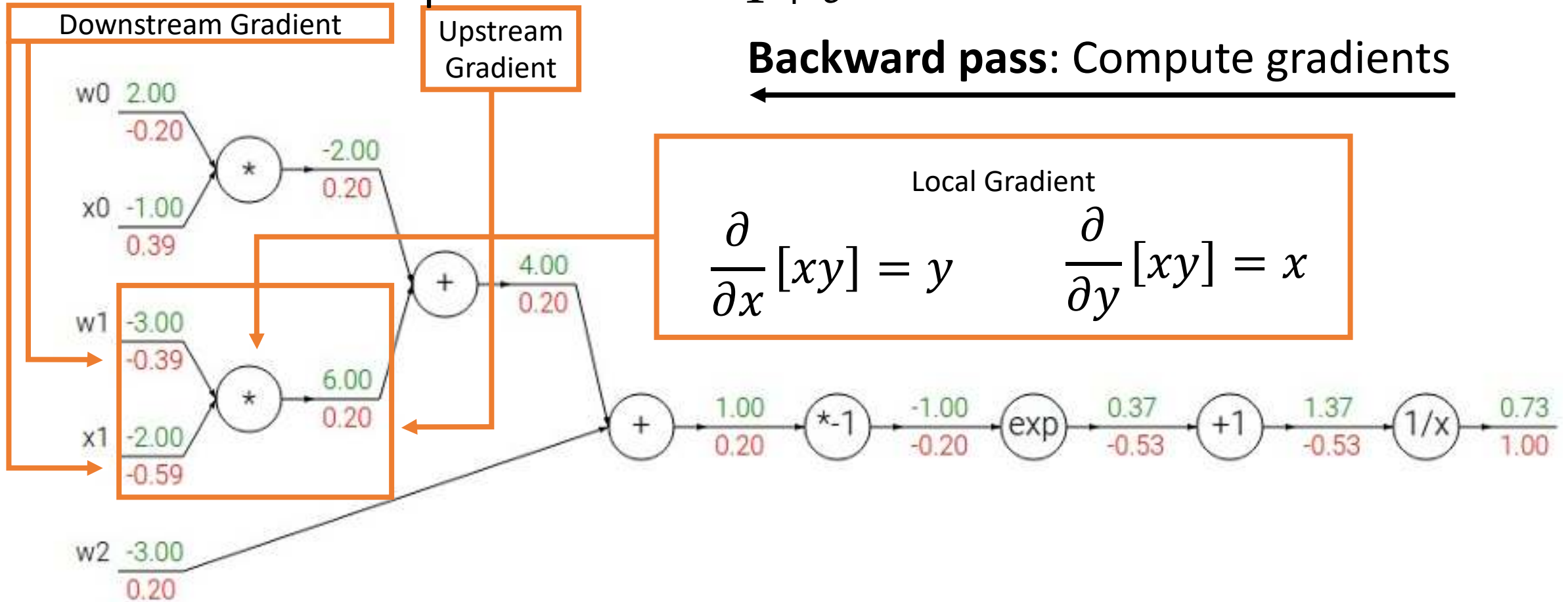
Backward pass: Compute gradients



Another Example

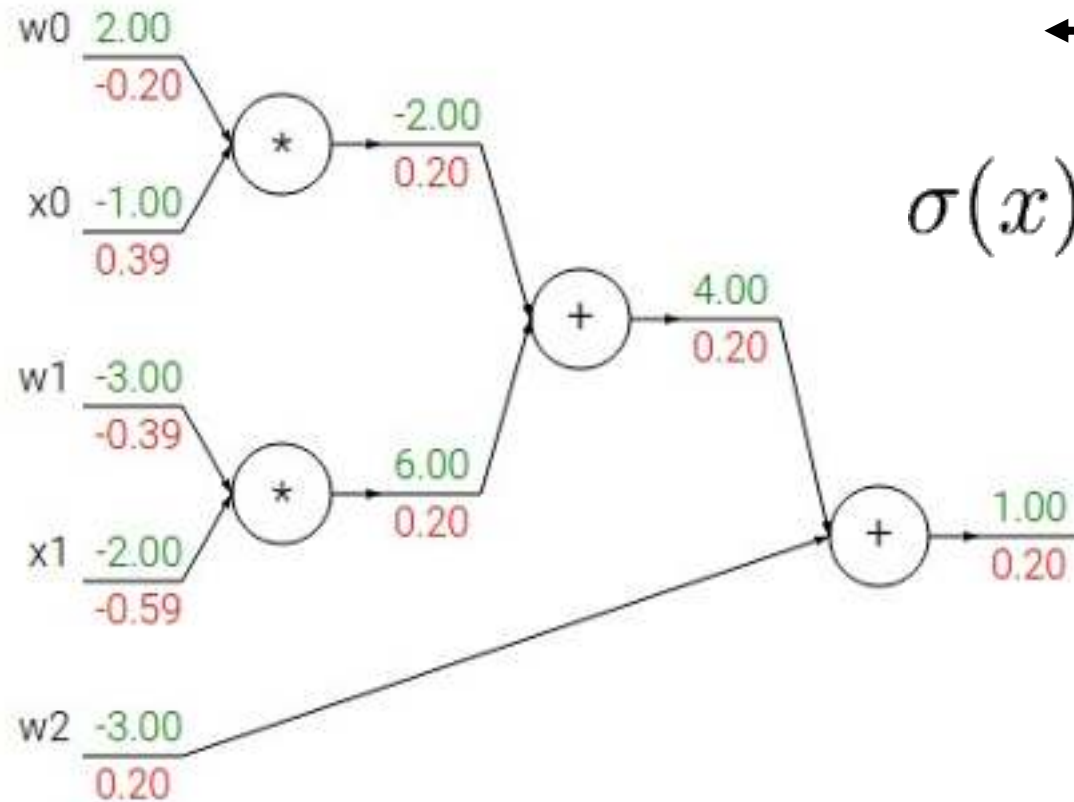
$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}}$$

Backward pass: Compute gradients



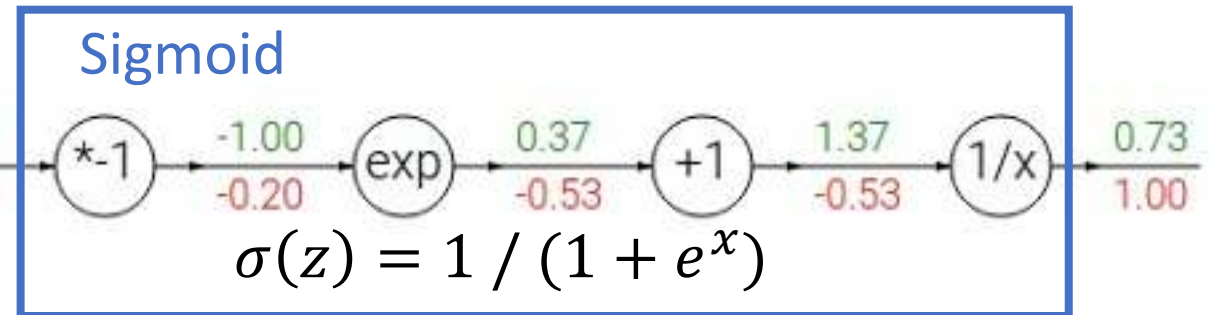
Another Example $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} = \sigma(w_0x_0 + w_1x_1 + w_2)$

Backward pass: Compute gradients



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Computational graph is not unique: we can use primitives that have simple local gradients



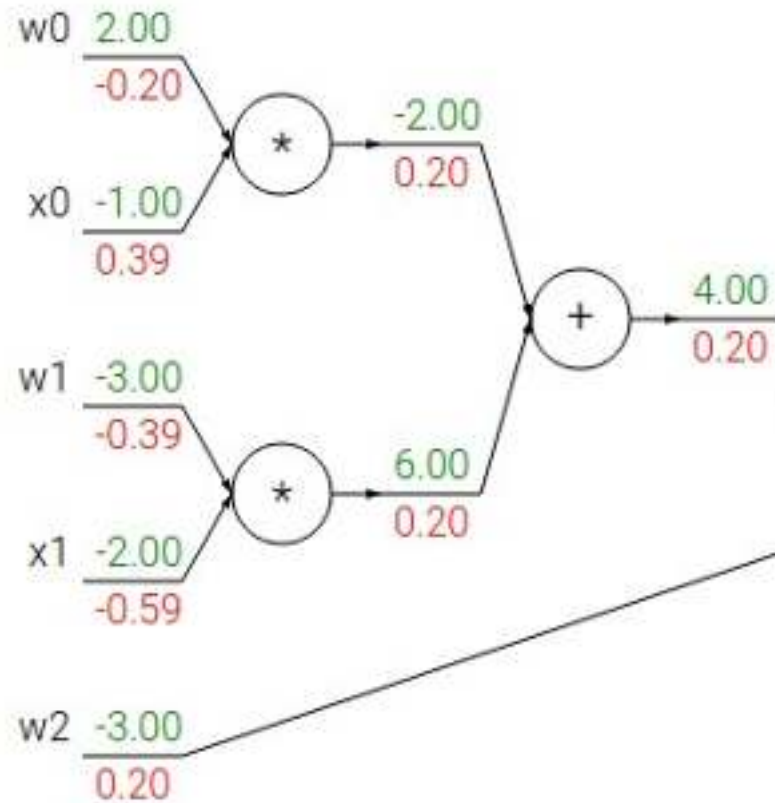
Sigmoid

$$\sigma(z) = 1 / (1 + e^z)$$

Another Example

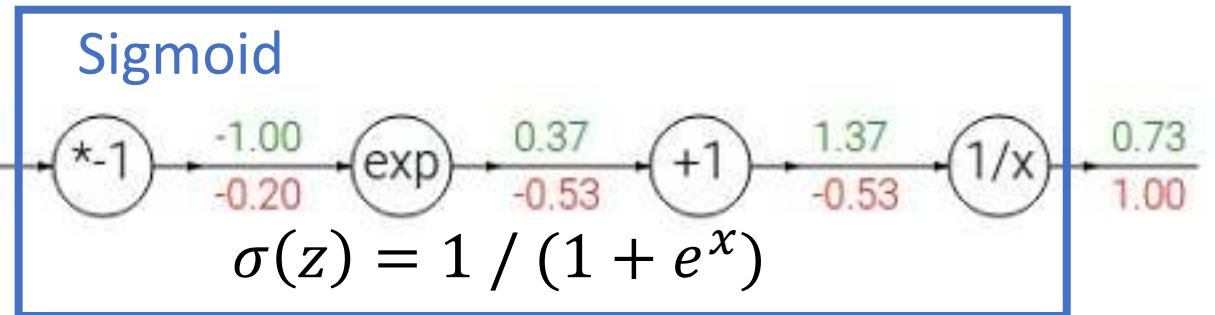
$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} = \sigma(w_0x_0 + w_1x_1 + w_2)$$

Backward pass: Compute gradients



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Computational graph is not unique: we can use primitives that have simple local gradients



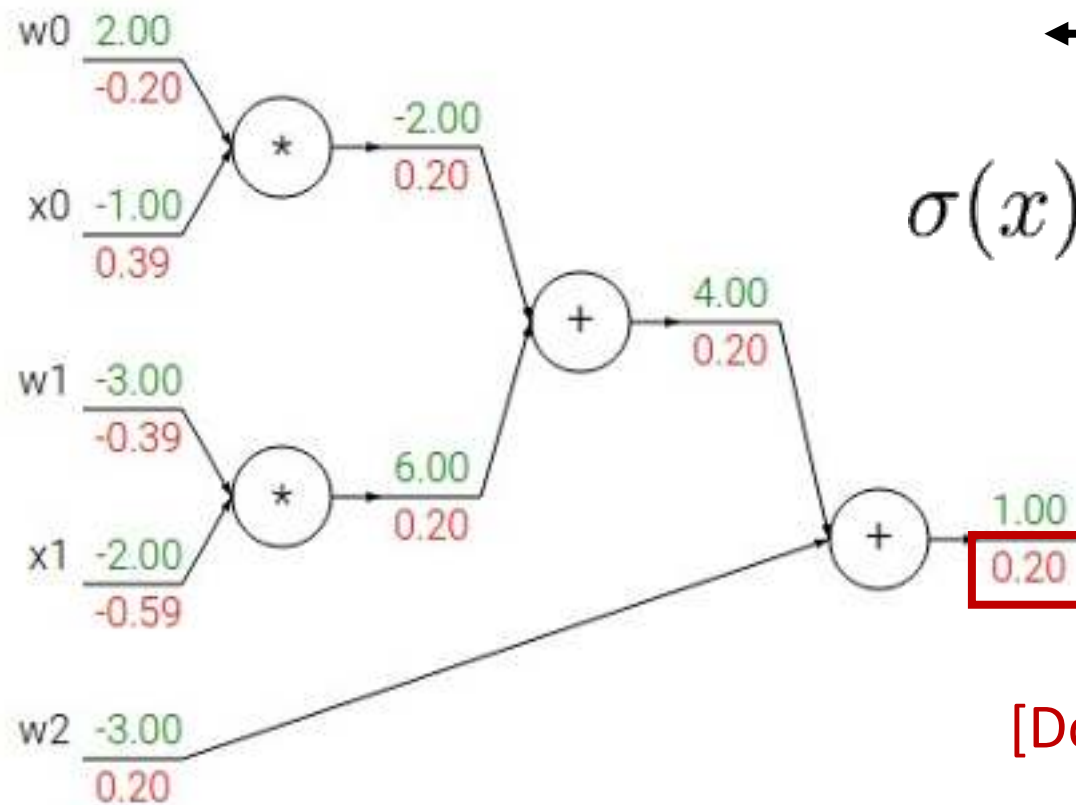
Sigmoid local gradient:

$$\frac{\partial}{\partial x} [\sigma(x)] = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$

Another Example

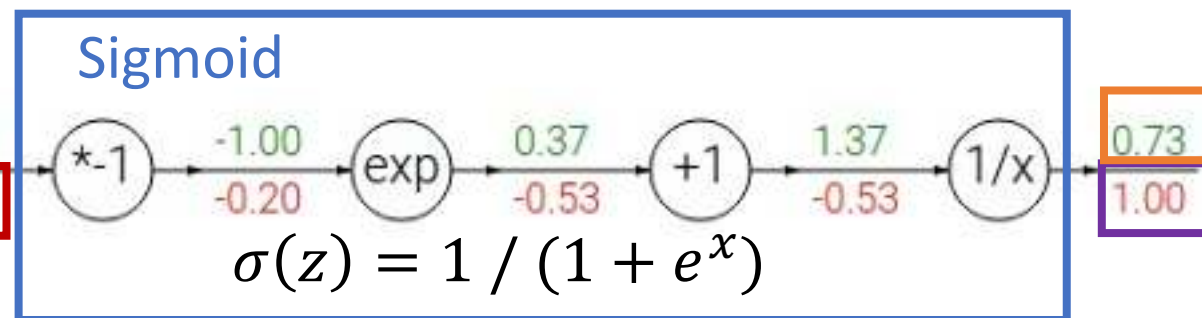
$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} = \sigma(w_0x_0 + w_1x_1 + w_2)$$

Backward pass: Compute gradients



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Computational graph is not unique: we can use primitives that have simple local gradients



$$\sigma(z) = 1 / (1 + e^z)$$

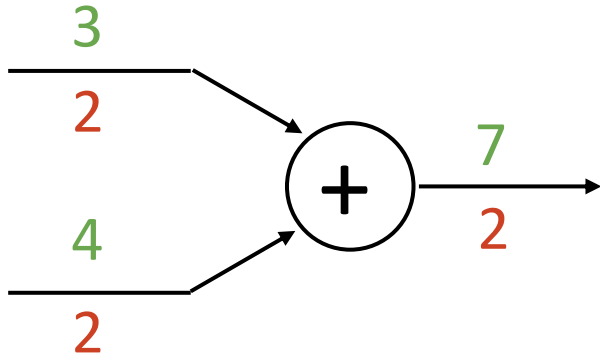
$$\begin{aligned} [\text{Downstream}] &= [\text{Local}] * [\text{Upstream}] \\ &= (1 - 0.73) * 0.73 * 1.0 = 0.2 \end{aligned}$$

Sigmoid local gradient:

$$\frac{\partial}{\partial x} [\sigma(x)] = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$

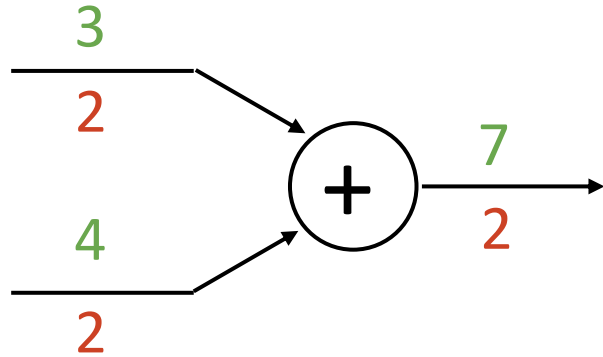
Patterns in Gradient Flow

add gate: gradient distributor

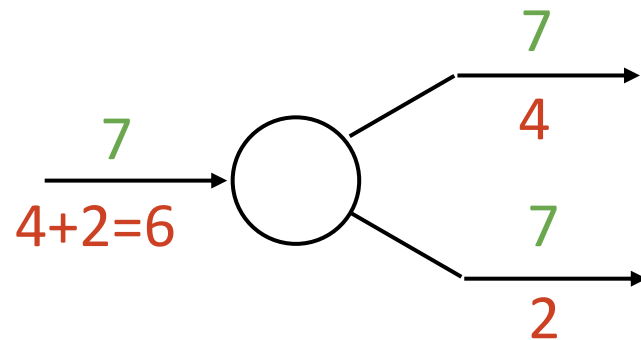


Patterns in Gradient Flow

add gate: gradient distributor

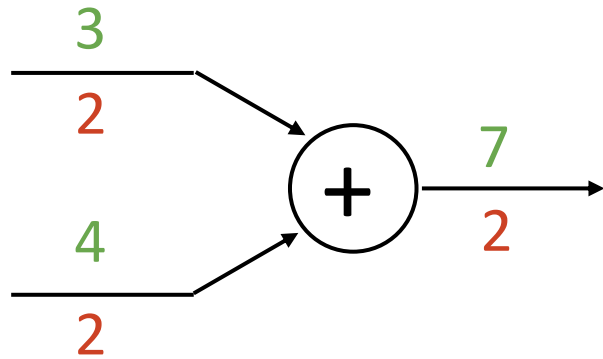


copy gate: gradient adder

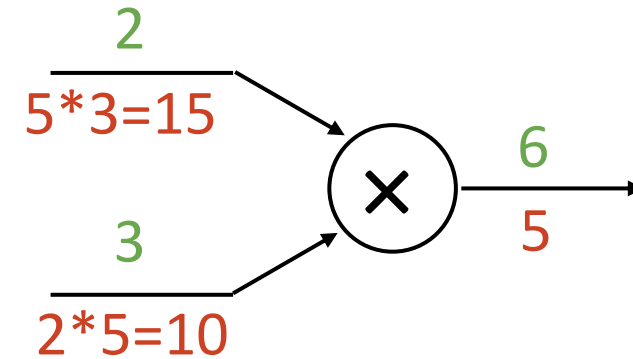


Patterns in Gradient Flow

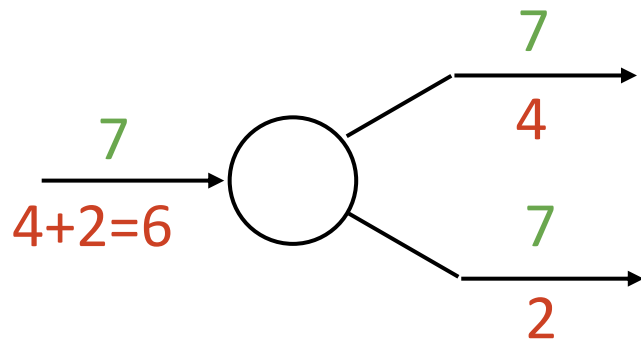
add gate: gradient distributor



mul gate: “swap multiplier”

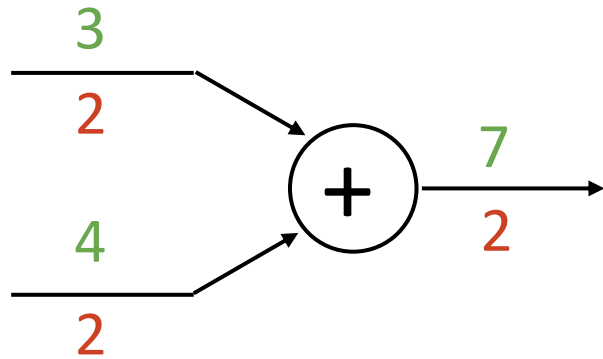


copy gate: gradient adder

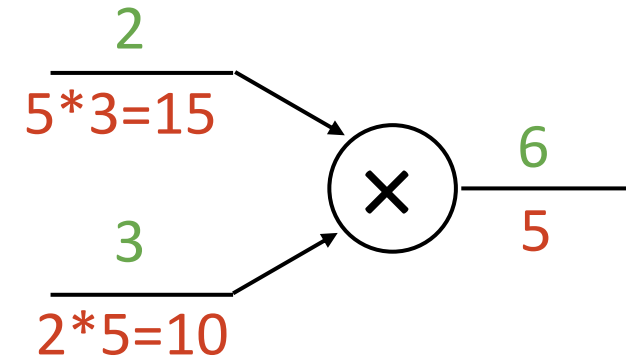


Patterns in Gradient Flow

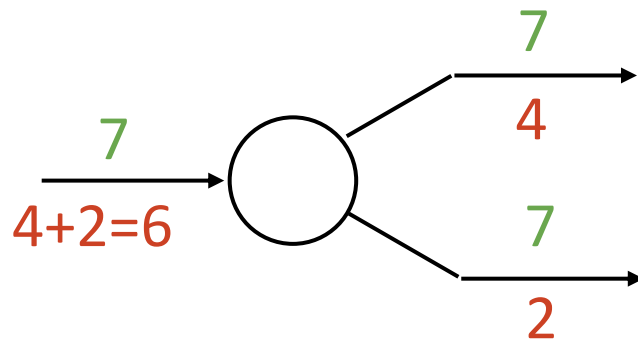
add gate: gradient distributor



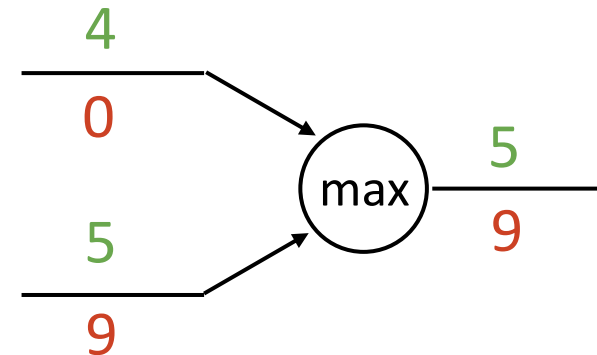
mul gate: “swap multiplier”



copy gate: gradient adder



max gate: gradient router



Backprop Implementation:

“Flat” gradient code:

Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
```

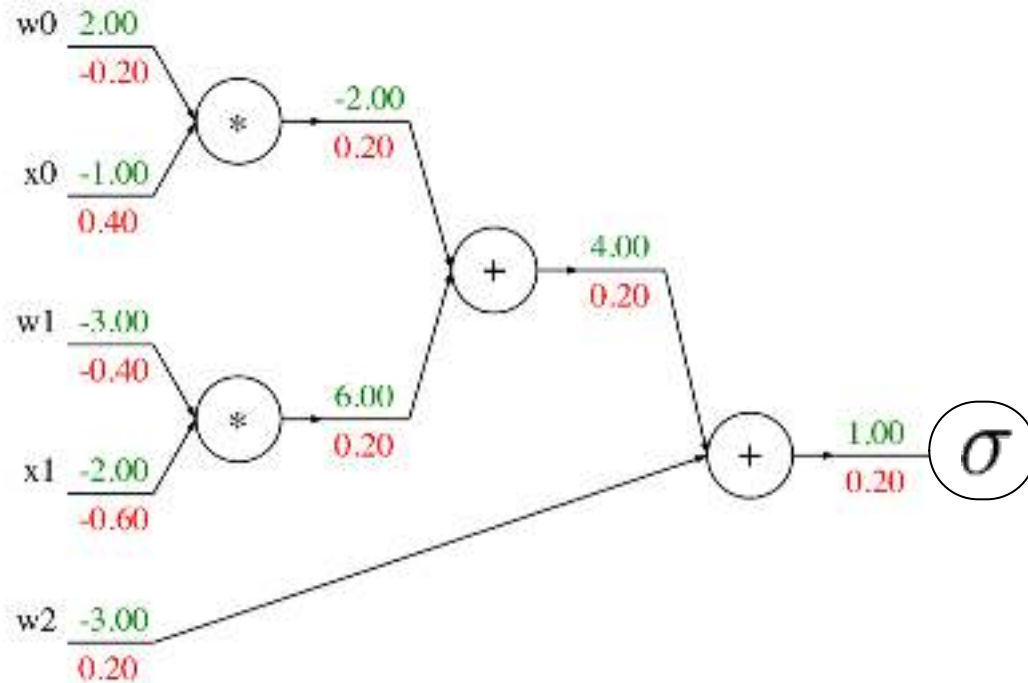
```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

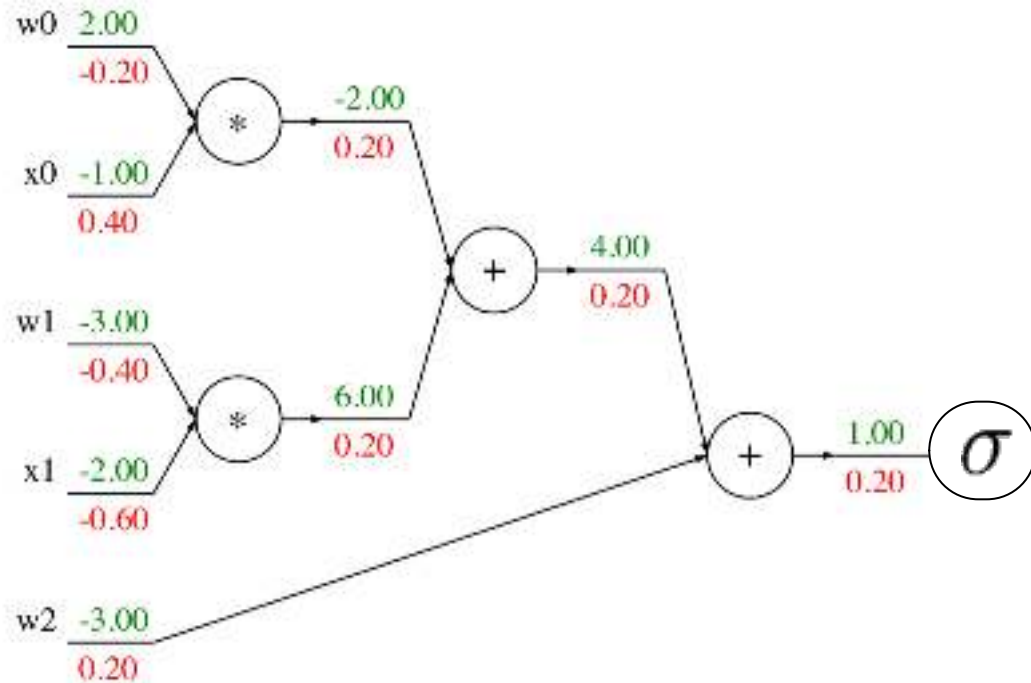


Backprop Implementation: "Flat" gradient code:

Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

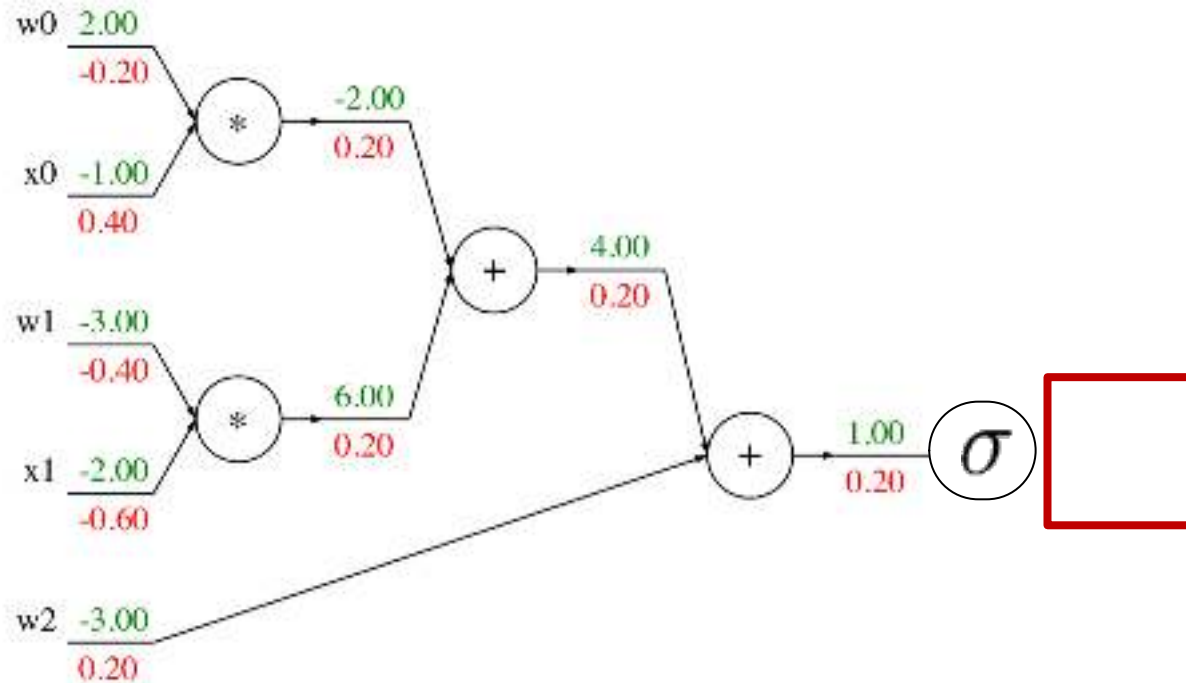


Backward pass:
Compute grads

```
    grad_L = 1.0  
    grad_s3 = grad_L * (1 - L) * L  
    grad_w2 = grad_s3  
    grad_s2 = grad_s3  
    grad_s0 = grad_s2  
    grad_s1 = grad_s2  
    grad_w1 = grad_s1 * x1  
    grad_x1 = grad_s1 * w1  
    grad_w0 = grad_s0 * x0  
    grad_x0 = grad_s0 * w0
```

Backprop Implementation: "Flat" gradient code:

Forward pass:
Compute output



Base case

```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

```
    grad_L = 1.0
```

```
    grad_s3 = grad_L * (1 - L) * L
```

```
    grad_w2 = grad_s3
```

```
    grad_s2 = grad_s3
```

```
    grad_s0 = grad_s2
```

```
    grad_s1 = grad_s2
```

```
    grad_w1 = grad_s1 * x1
```

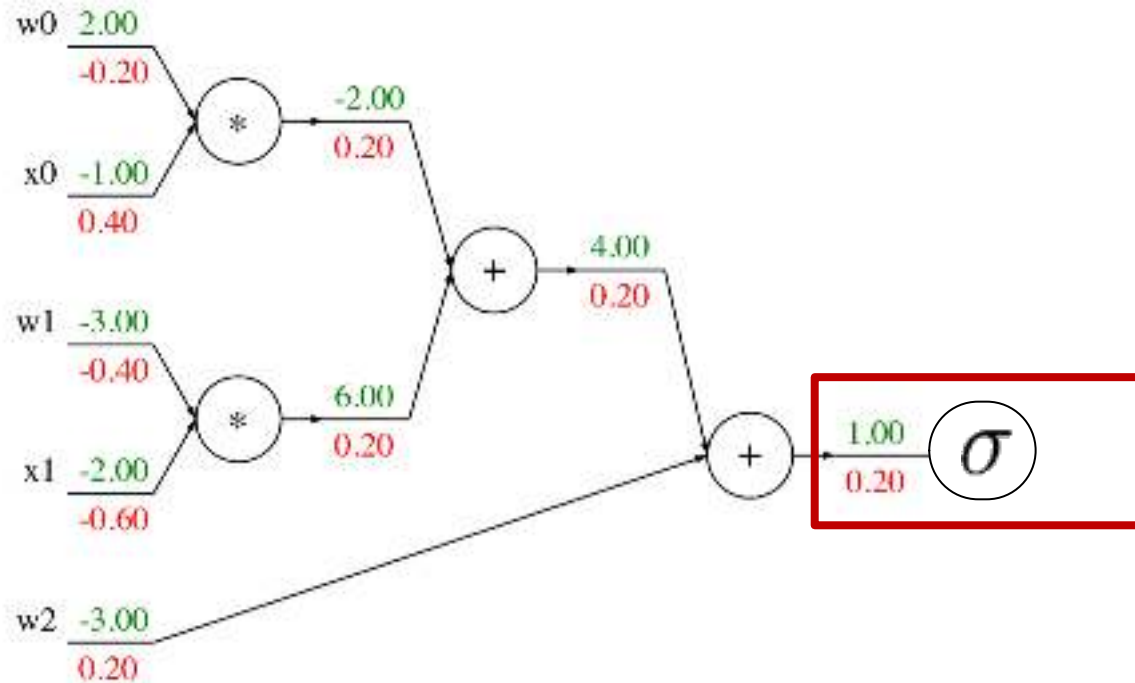
```
    grad_x1 = grad_s1 * w1
```

```
    grad_w0 = grad_s0 * x0
```

```
    grad_x0 = grad_s0 * w0
```

Backprop Implementation: "Flat" gradient code:

Forward pass:
Compute output



Sigmoid

```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

```
    grad_L = 1.0
```

```
    grad_s3 = grad_L * (1 - L) * L
```

```
    grad_w2 = grad_s3
```

```
    grad_s2 = grad_s3
```

```
    grad_s0 = grad_s2
```

```
    grad_s1 = grad_s2
```

```
    grad_w1 = grad_s1 * x1
```

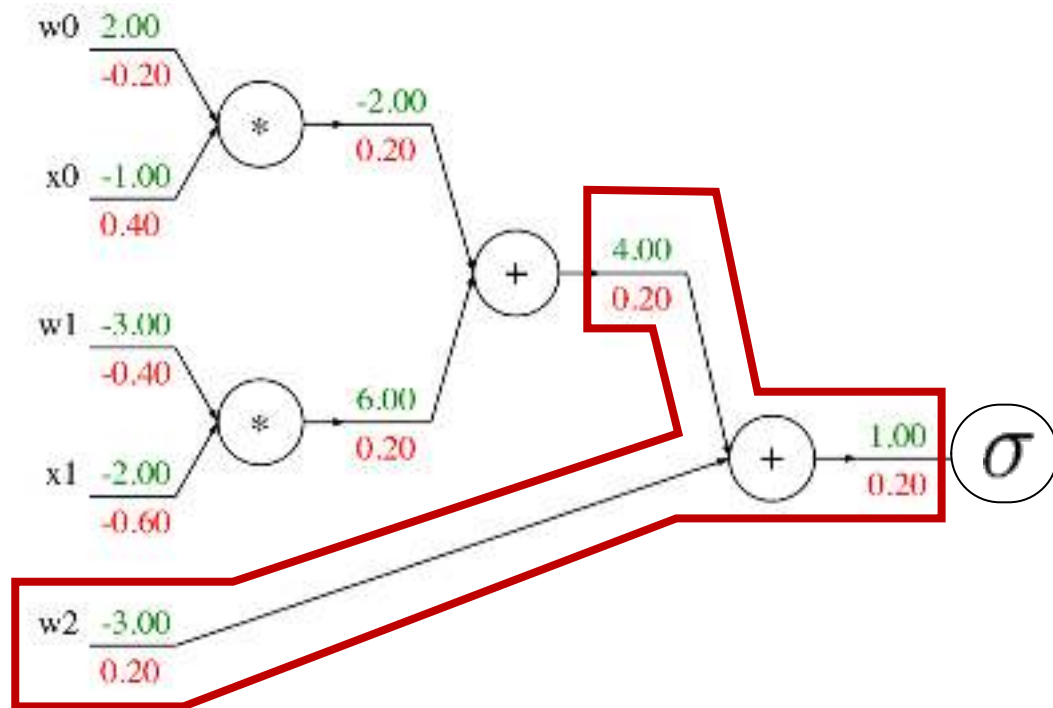
```
    grad_x1 = grad_s1 * w1
```

```
    grad_w0 = grad_s0 * x0
```

```
    grad_x0 = grad_s0 * w0
```

Backprop Implementation: "Flat" gradient code:

Forward pass:
Compute output



```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

```
    grad_L = 1.0
```

```
    grad_s3 = grad_L * (1 - L) * L
```

```
    grad_w2 = grad_s3
```

```
    grad_s2 = grad_s3
```

```
    grad_s0 = grad_s2
```

```
    grad_s1 = grad_s2
```

```
    grad_w1 = grad_s1 * x1
```

```
    grad_x1 = grad_s1 * w1
```

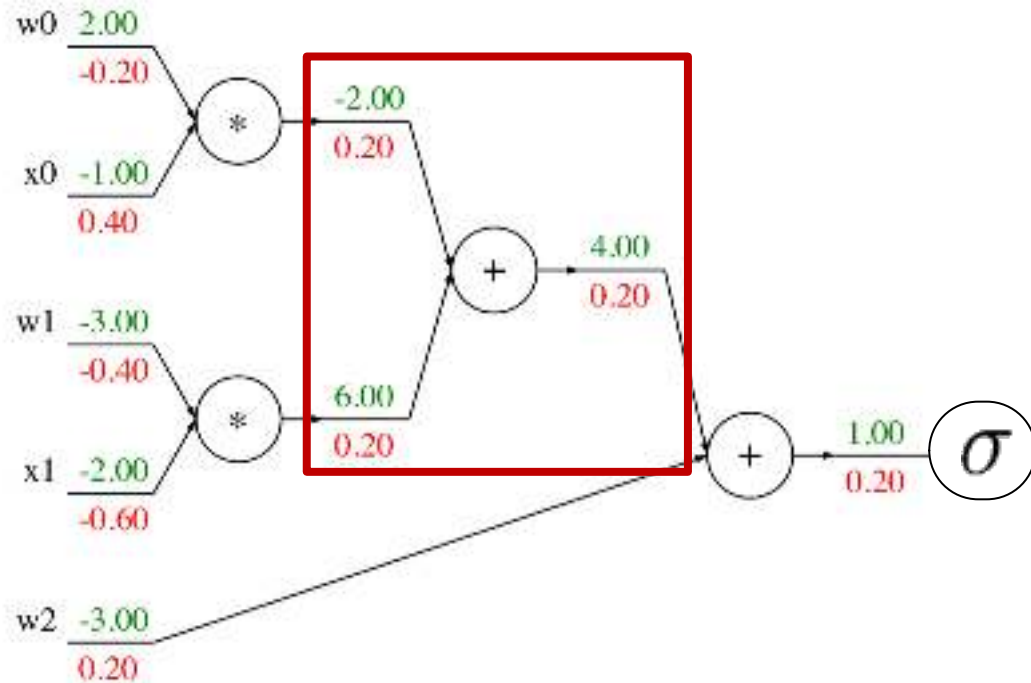
```
    grad_w0 = grad_s0 * x0
```

```
    grad_x0 = grad_s0 * w0
```

Add

Backprop Implementation: "Flat" gradient code:

Forward pass:
Compute output



```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

```
grad_L = 1.0
```

```
grad_s3 = grad_L * (1 - L) * L
```

```
grad_w2 = grad_s3
```

```
grad_s2 = grad_s3
```

```
grad_s0 = grad_s2
```

```
grad_s1 = grad_s2
```

```
grad_w1 = grad_s1 * x1
```

```
grad_x1 = grad_s1 * w1
```

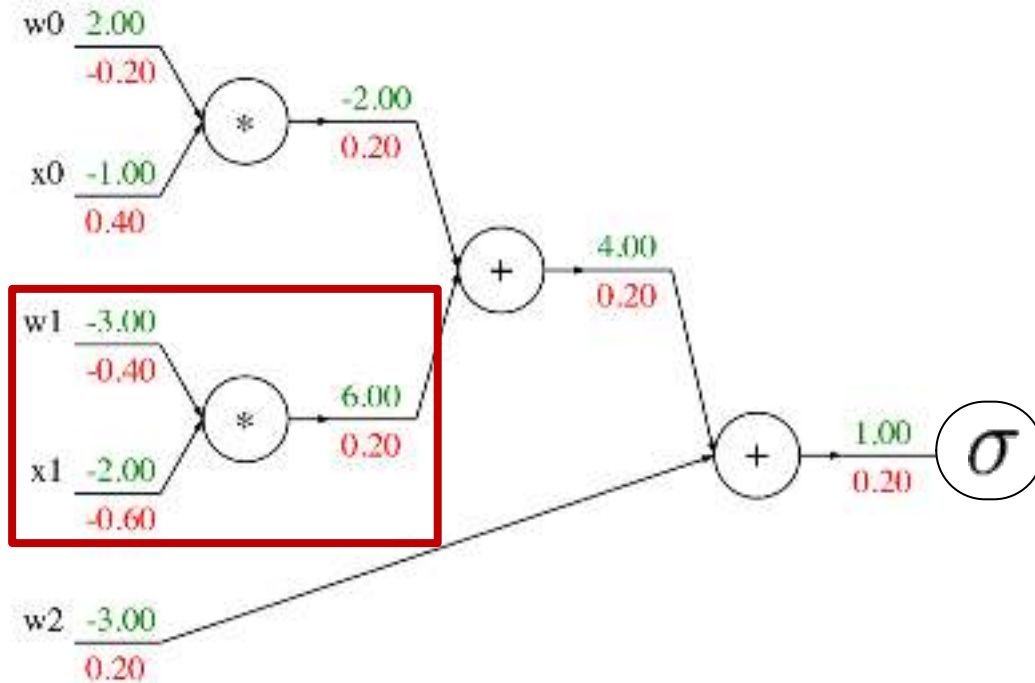
```
grad_w0 = grad_s0 * x0
```

```
grad_x0 = grad_s0 * w0
```

Add

Backprop Implementation: "Flat" gradient code:

Forward pass:
Compute output



```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

```
grad_L = 1.0
```

```
grad_s3 = grad_L * (1 - L) * L
```

```
grad_w2 = grad_s3
```

```
grad_s2 = grad_s3
```

```
grad_s0 = grad_s2
```

```
grad_s1 = grad_s2
```

```
grad_w1 = grad_s1 * x1
```

```
grad_x1 = grad_s1 * w1
```

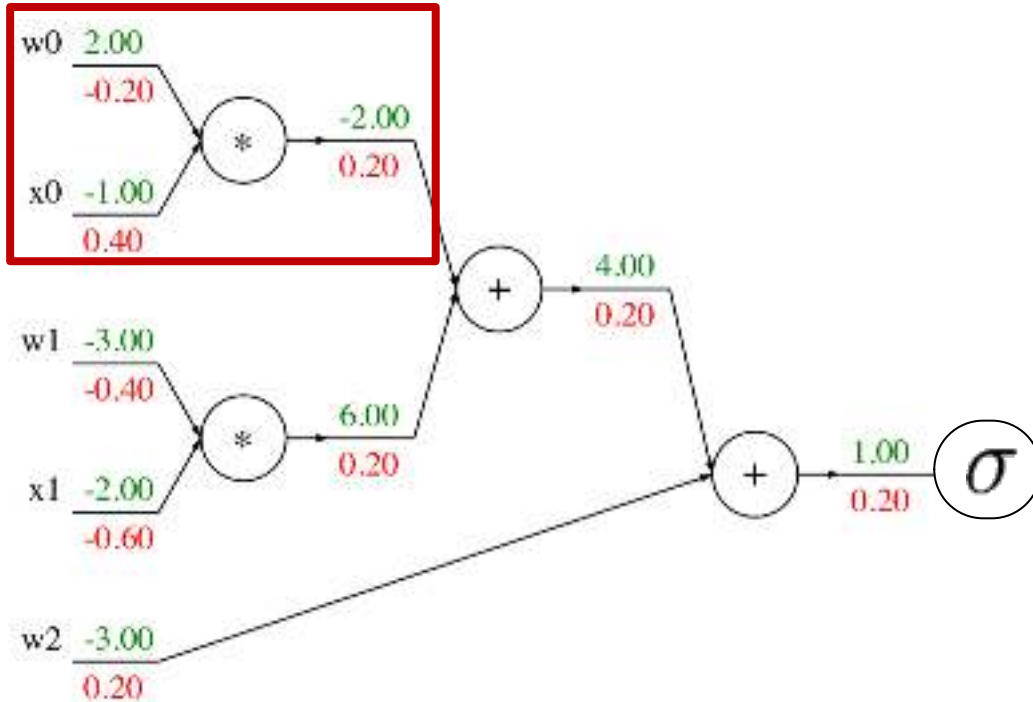
```
grad_w0 = grad_s0 * x0
```

```
grad_x0 = grad_s0 * w0
```

Multiply

Backprop Implementation: "Flat" gradient code:

Forward pass:
Compute output



```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

```
grad_L = 1.0
```

```
grad_s3 = grad_L * (1 - L) * L
```

```
grad_w2 = grad_s3
```

```
grad_s2 = grad_s3
```

```
grad_s0 = grad_s2
```

```
grad_s1 = grad_s2
```

```
grad_w1 = grad_s1 * x1
```

```
grad_x1 = grad_s1 * w1
```

```
grad_w0 = grad_s0 * x0
```

```
grad_x0 = grad_s0 * w0
```

Multiply

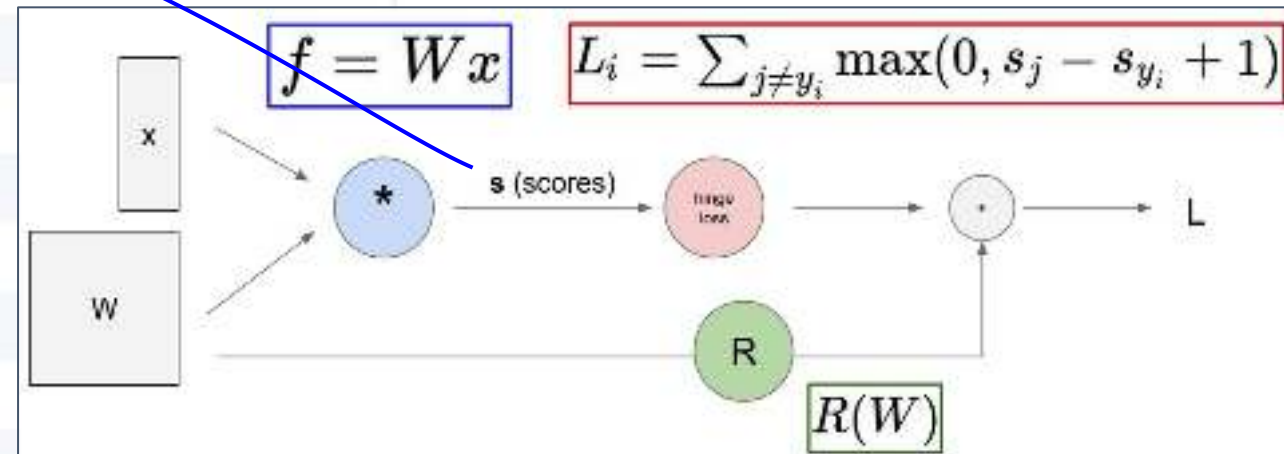
“Flat” Backprop: Do this for Assignment 2!

Your gradient code should look like a “reversed version” of your forward pass!

E.g. for the SVM:

```
# receive W (weights), X (data)
# forward pass (we have 8 lines)
scores = #...
margins = #...
data_loss = #...
reg_loss = #...
loss = data_loss + reg_loss

# backward pass (we have 5 lines)
dmargins = # ... (optionally, we go direct to dscores)
dscores = #...
dW = #...
```



“Flat” Backprop: Do this for Assignment 2!

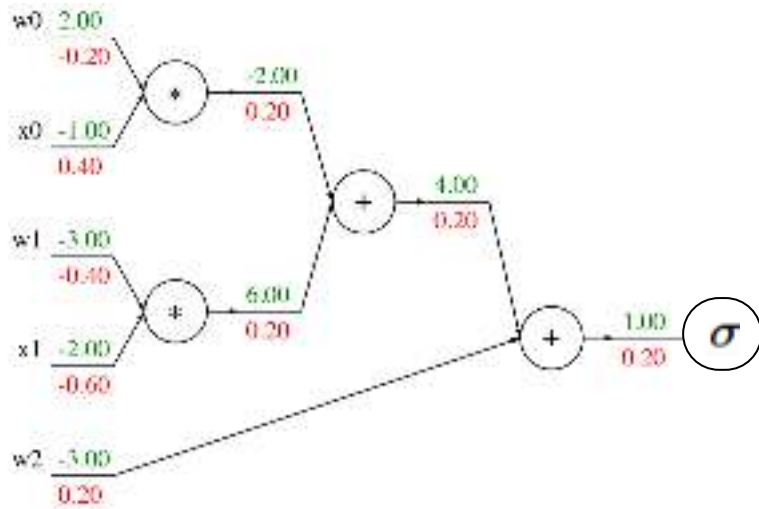
Your gradient code should look like a “reversed version” of your forward pass!

E.g. for two-layer neural net:

```
# receive W1,W2,b1,b2 (weights/biases), X (data)
# forward pass:
h1 = #... function of X,W1,b1
scores = #... function of h1,W2,b2
loss = #... (several lines of code to evaluate Softmax loss)
# backward pass:
dscores = #...
dh1,dW2,db2 = #...
dW1,db1 = #...
```

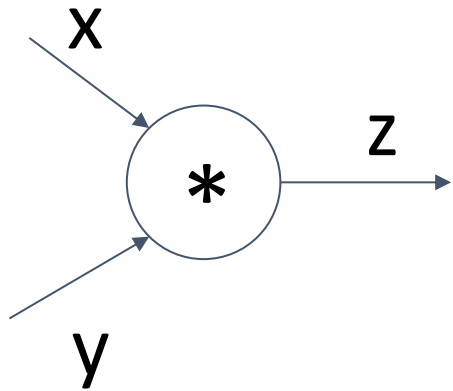
Backprop Implementation: Modular API

Graph (or Net) object *(rough pseudo code)*



```
class ComputationalGraph(object):  
    #...  
    def forward(inputs):  
        # 1. [pass inputs to input gates...]  
        # 2. forward the computational graph:  
        for gate in self.graph.nodes_topologically_sorted():  
            gate.forward()  
        return loss # the final gate in the graph outputs the loss  
    def backward():  
        for gate in reversed(self.graph.nodes_topologically_sorted()):  
            gate.backward() # little piece of backprop (chain rule applied)  
        return inputs_gradients
```


Example: PyTorch Autograd Functions



(x,y,z are scalars)

```
class Multiply(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x, y):  
        ctx.save_for_backward(x, y)  
        z = x * y  
        return z  
    @staticmethod  
    def backward(ctx, grad_z):  
        x, y = ctx.saved_tensors  
        grad_x = y * grad_z    # dz/dx * dL/dz  
        grad_y = x * grad_z    # dz/dy * dL/dz  
        return grad_x, grad_y
```

Need to stash some
values for use in
backward

Upstream
gradient

Multiply upstream
and local gradients

Example: PyTorch operators

[illegible][illegible]

PyTorch sigmoid layer

```
1  #ifndef TH_GENERIC_FILE
2  #define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
3  #else
4
5  void THNN_(Sigmoid_updateOutput)(
6      THNNState *state,
7      THTensor *input,
8      THTensor *output)
9  {
10     THTensor_(sigmoid)(output, input);
11 }
12
13 void THNN_(Sigmoid_updateGradInput)(
14     THNNState *state,
15     THTensor *gradOutput,
16     THTensor *gradInput,
17     THTensor *output)
18 {
19     THNN_CHECK_NELEMENT(output, gradOutput);
20     THTensor_(resizeAs)(gradInput, output);
21     TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
22         scalar_t z = *output_data;
23         *gradInput_data = *gradOutput_data * (1. - z) * z;
24     );
25 }
26
27 #endif
```

[Source](#)

PyTorch sigmoid layer

```
1 #ifndef TH_GENERIC_FILE
2 #define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
3 #else
```

```
5 void THNN_(Sigmoid_updateOutput)(
6     THNNState *state,
7     THTensor *input,
8     THTensor *output)
9 {
10     THTensor_(sigmoid)(output, input);
11 }
```

Forward

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

```
13 void THNN_(Sigmoid_updateGradInput)(
14     THNNState *state,
15     THTensor *gradOutput,
16     THTensor *gradInput,
17     THTensor *output)
18 {
19     THNN_CHECK_NELEMENT(output, gradOutput);
20     THTensor_(resizeAs)(gradInput, output);
21     TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
22         scalar_t z = *output_data;
23         *gradInput_data = *gradOutput_data * (1. - z) * z;
24     );
25 }
```

```
27 #endif
```

[Source](#)

PyTorch sigmoid layer

```
1 #ifndef TH_GENERIC_FILE
2 #define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
3 #else
```

```
5 void THNN_(Sigmoid_updateOutput)(
6     THNNState *state,
7     THTensor *input,
8     THTensor *output)
9 {
10     THTensor_(sigmoid)(output, input);
11 }
```

Forward

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

```
13 void THNN_(Sigmoid_updateGradInput)(
14     THNNState *state,
15     THTensor *gradOutput,
16     THTensor *gradInput,
17     THTensor *output)
18 {
19     THNN_CHECK_NELEMENT(output, gradOutput);
20     THTensor_(resizeAs)(gradInput, output);
21     TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
22         scalar_t z = *output_data;
23         *gradInput_data = *gradOutput_data * (1. - z) * z;
24     );
25 }
26
27 #endif
```

```
static void sigmoid_kernel(TensorIterator& iter) {
    AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
        unary_kernel_vec(
            iter,
            [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp((-a)))); },
            [=](Vec256<scalar_t> a) {
                a = Vec256<scalar_t>{(scalar_t){0}} - a;
                a = a.exp();
                a = Vec256<scalar_t>{(scalar_t){1}} + a;
                a = a.reciprocal();
                return a;
            });
    });
}
```

Forward actually defined [elsewhere...](#)

```
return (1 / (1 + std::exp((-a))));
```

[Source](#)

PyTorch sigmoid layer

```
1 #ifndef TH_GENERIC_FILE
2 #define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
3 #else
```

```
5 void THNN_(Sigmoid_updateOutput)(
6     THNNState *state,
7     THTensor *input,
8     THTensor *output)
9 {
10     THTensor_(sigmoid)(output, input);
11 }
```

Forward

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

```
13 void THNN_(Sigmoid_updateGradInput)(
14     THNNState *state,
15     THTensor *gradOutput,
16     THTensor *gradInput,
17     THTensor *output)
18 {
19     THNN_CHECK_NELEMENT(output, gradOutput);
20     THTensor_(resizeAs)(gradInput, output);
21     TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
22         scalar_t z = *output_data;
23         *gradInput_data = *gradOutput_data * (1. - z) * z;
24     );
25 }
```

Backward

$$(1 - \sigma(x)) \sigma(x)$$

[Source](#)

So far: backprop with scalars

What about vector-valued functions?

Recap: Vector Derivatives

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

Recap: Vector Derivatives

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

$$x \in \mathbb{R}^N, y \in \mathbb{R}$$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N, \\ \left(\frac{\partial y}{\partial x} \right)_i = \frac{\partial y}{\partial x_i}$$

For each element of x , if it changes by a small amount then how much will y change?

Recap: Vector Derivatives

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

$$x \in \mathbb{R}^N, y \in \mathbb{R}$$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N, \\ \left(\frac{\partial y}{\partial x} \right)_i = \frac{\partial y}{\partial x_i}$$

For each element of x , if it changes by a small amount then how much will y change?

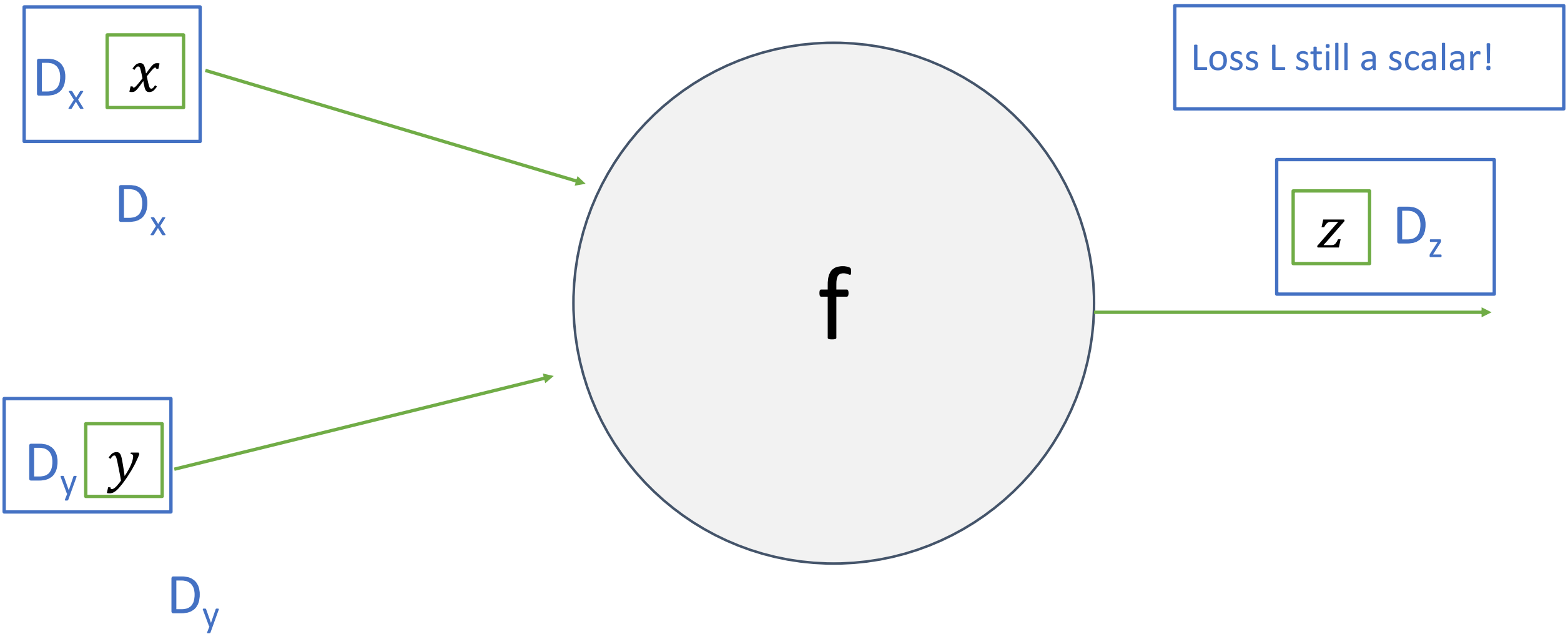
$$x \in \mathbb{R}^N, y \in \mathbb{R}^M$$

Derivative is **Jacobian**:

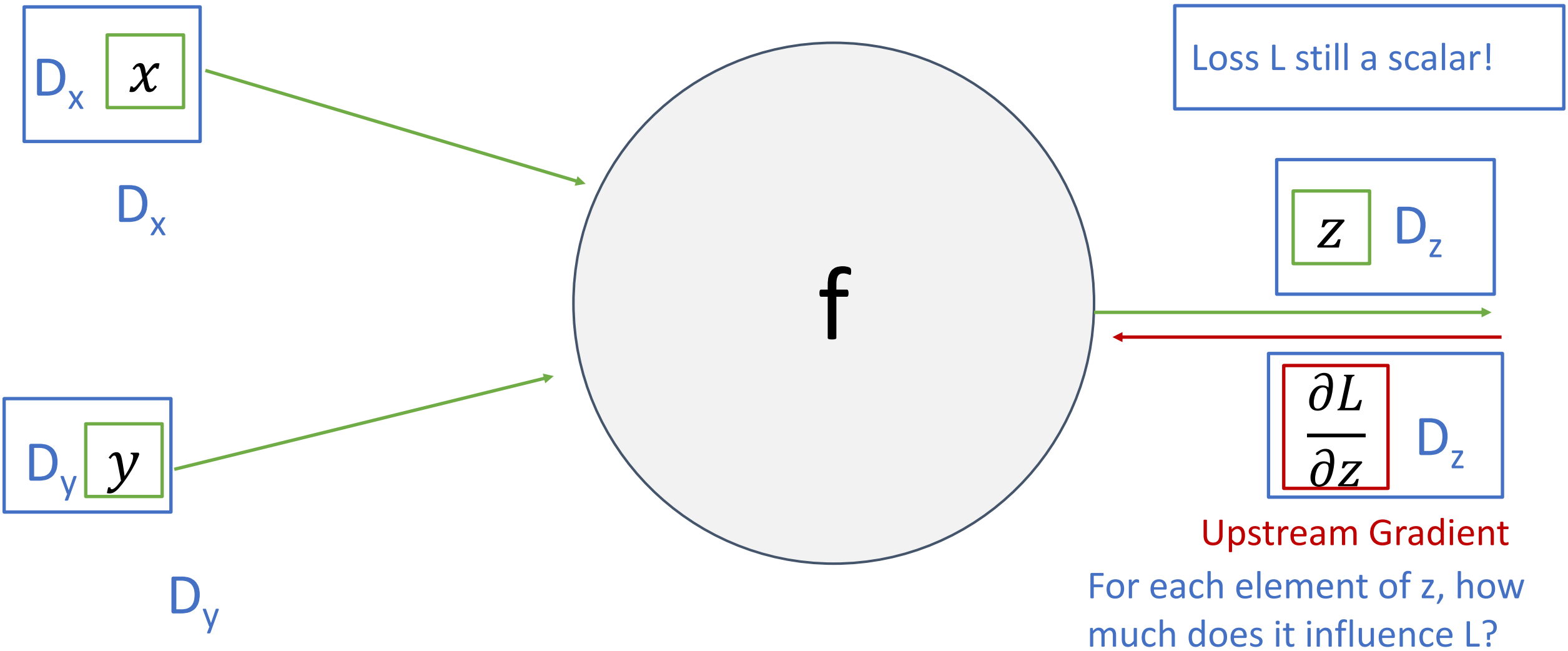
$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \\ \left(\frac{\partial y}{\partial x} \right)_{i,j} = \frac{\partial y_j}{\partial x_i}$$

For each element of x , if it changes by a small amount then how much will each element of y change?

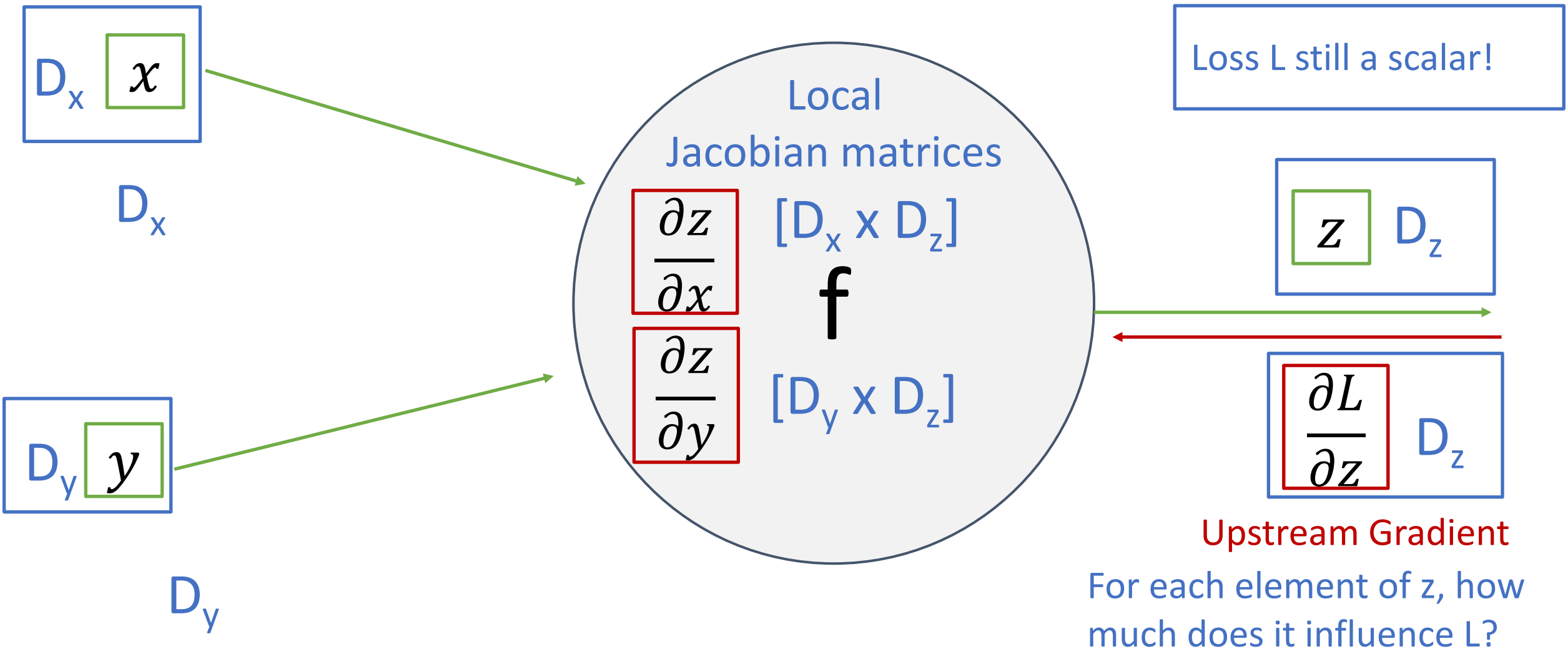
Backprop with Vectors



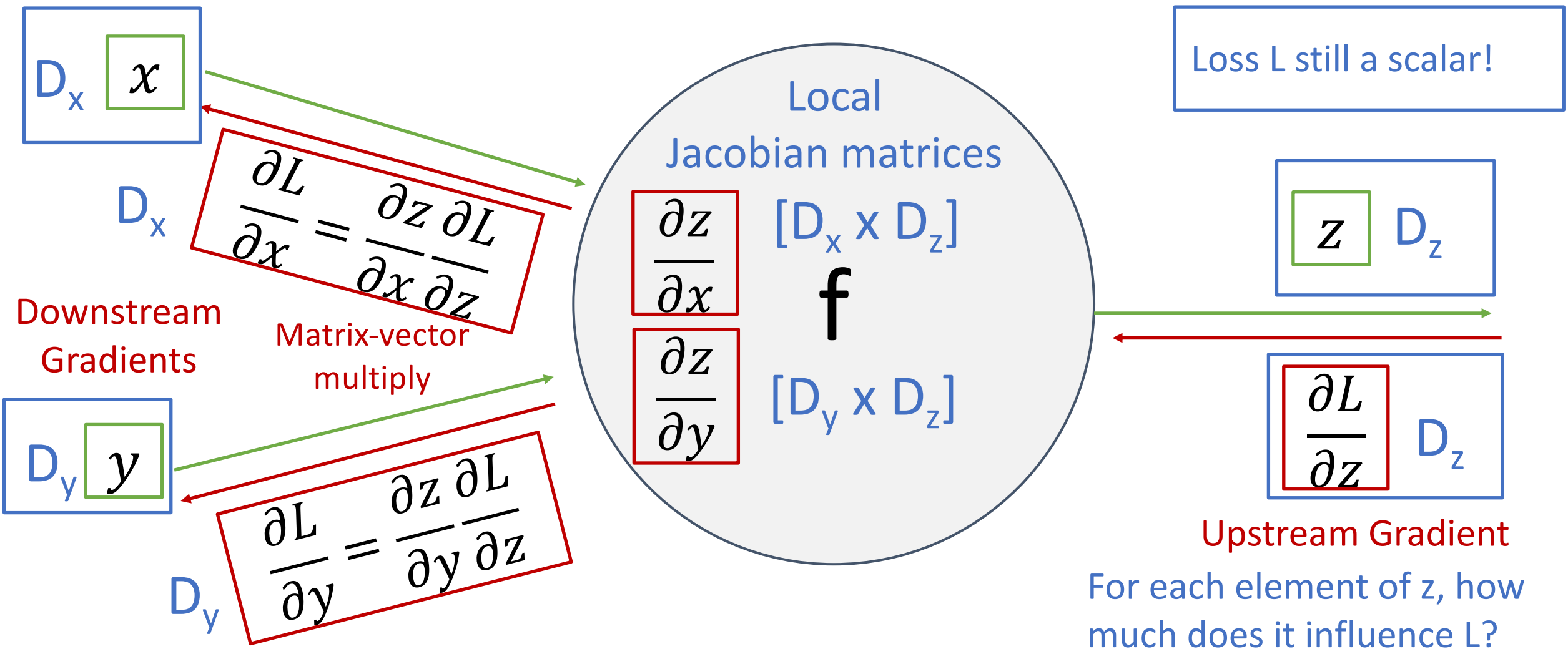
Backprop with Vectors



Backprop with Vectors



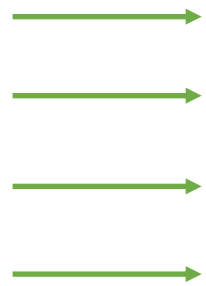
Backprop with Vectors



Backprop with Vectors

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

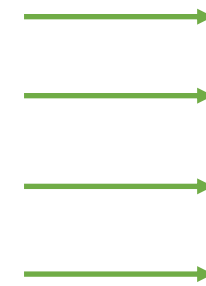


$$f(x) = \max(0, x)$$

(elementwise)

4D output y:

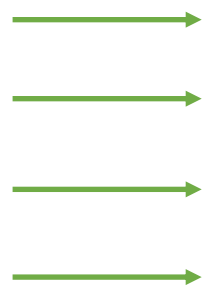
$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$



Backprop with Vectors

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

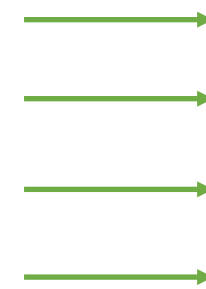


$$f(x) = \max(0, x)$$

(elementwise)

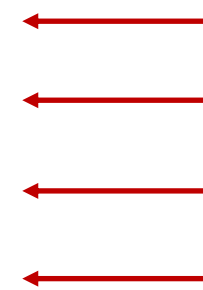
4D output y:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$



4D dL/dy:

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

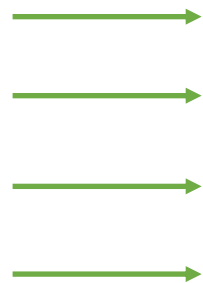


Upstream
gradient

Backprop with Vectors

4D input x:

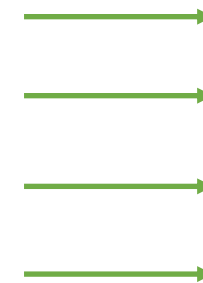
$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$



$f(x) = \max(0, x)$
(*elementwise*)

4D output y:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$



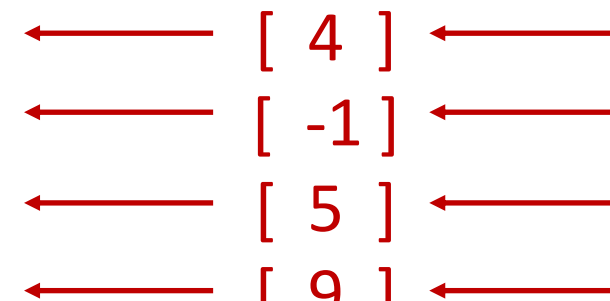
Jacobian dy/dx

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

4D dL/dy :

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

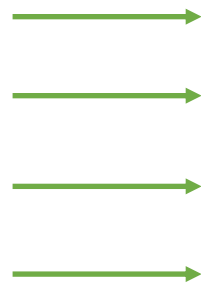
Upstream
gradient



Backprop with Vectors

4D input x:

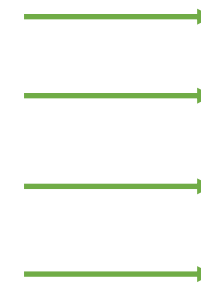
$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$



$f(x) = \max(0, x)$
(*elementwise*)

4D output y:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$



$\begin{bmatrix} dy/dx & dL/dy \end{bmatrix}$

$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix}$

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

$\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 5 \end{bmatrix}$

$\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 9 \end{bmatrix}$

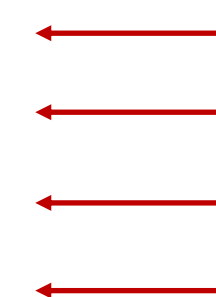
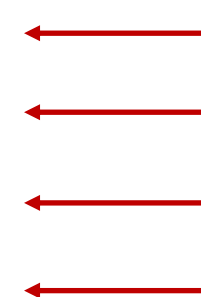
4D dL/dy:

$\begin{bmatrix} 4 \end{bmatrix}$

$\begin{bmatrix} -1 \end{bmatrix}$

$\begin{bmatrix} 5 \end{bmatrix}$

$\begin{bmatrix} 9 \end{bmatrix}$

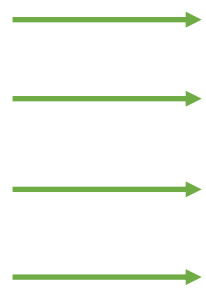


Upstream
gradient

Backprop with Vectors

4D input x:

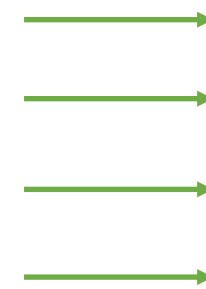
$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$



$f(x) = \max(0, x)$
(*elementwise*)

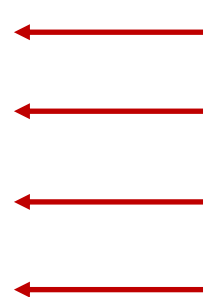
4D output y:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$



4D dL/dx :

$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix}$

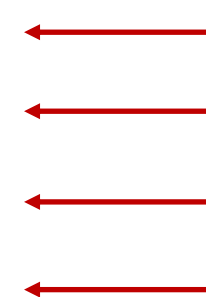


$[dy/dx] [dL/dy]$

$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix}$
 $\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \end{bmatrix}$
 $\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 5 \end{bmatrix}$
 $\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 9 \end{bmatrix}$

4D dL/dy :

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$



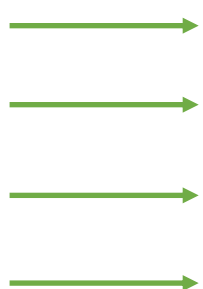
Upstream
gradient

Backprop with Vectors

Jacobian is **sparse**: off-diagonal entries all zero! Never **explicitly** form Jacobian; instead use **implicit** multiplication

4D input x:

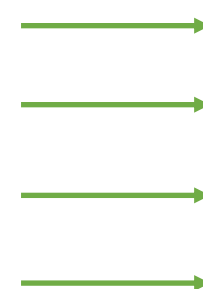
$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$



$f(x) = \max(0, x)$
(*elementwise*)

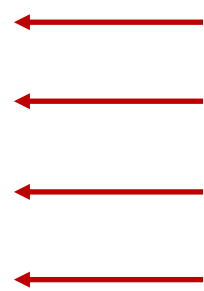
4D output y:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$



4D dL/dx:

$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix}$

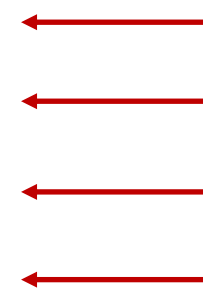


$\begin{bmatrix} dy/dx & dL/dy \end{bmatrix}$

$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

4D dL/dy:

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$



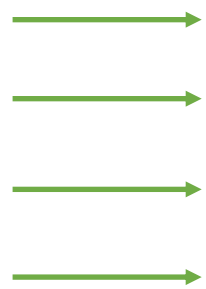
Upstream
gradient

Backprop with Vectors

Jacobian is **sparse**: off-diagonal entries all zero! Never **explicitly** form Jacobian; instead use **implicit** multiplication

4D input x:

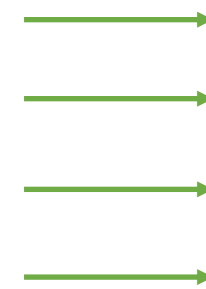
$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$



$$f(x) = \max(0, x) \\ (\text{elementwise})$$

4D output y:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$



4D dL/dx:

$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix}$

$$\left(\frac{\partial L}{\partial x}\right)_i = \begin{cases} \left(\frac{\partial L}{\partial y}\right)_i, & \text{if } x_i > 0 \\ 0, & \text{otherwise} \end{cases}$$

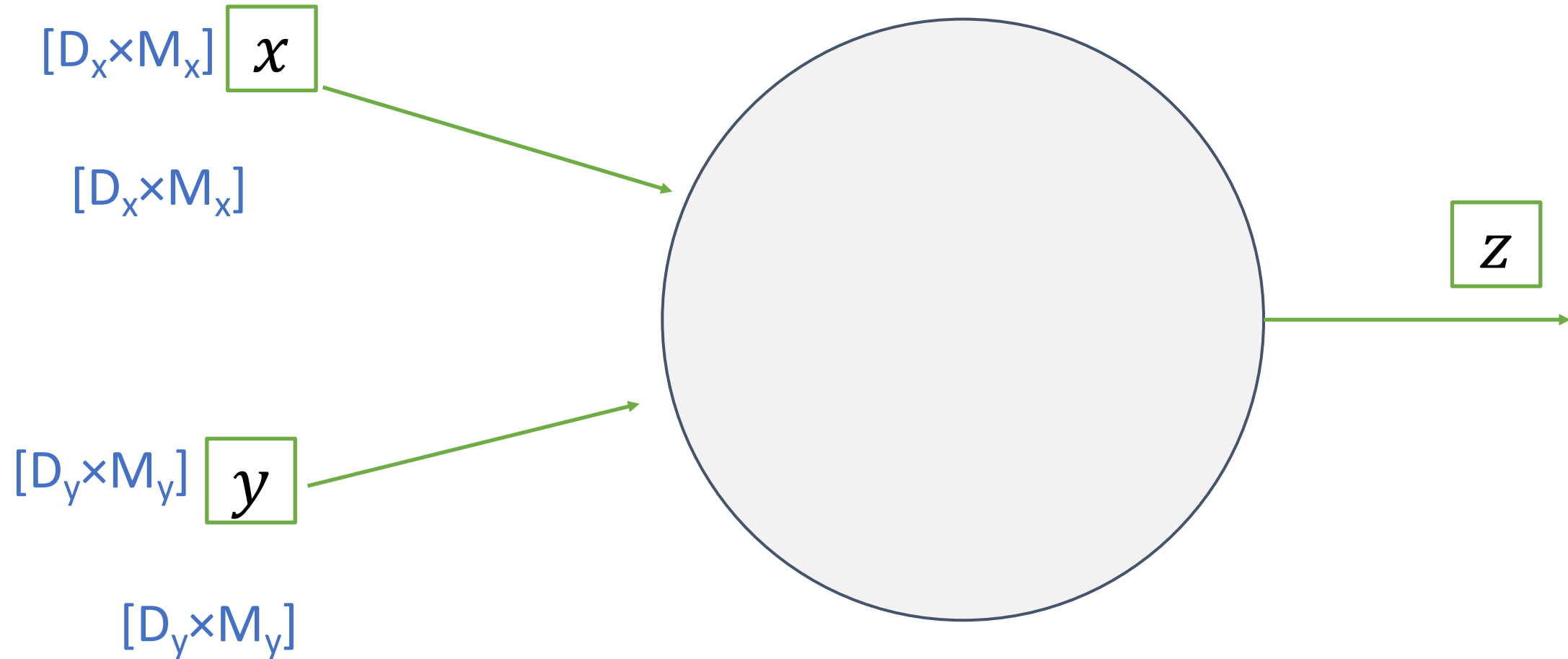
$[dy/dx] [dL/dy]$

4D dL/dy:

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

Upstream
gradient

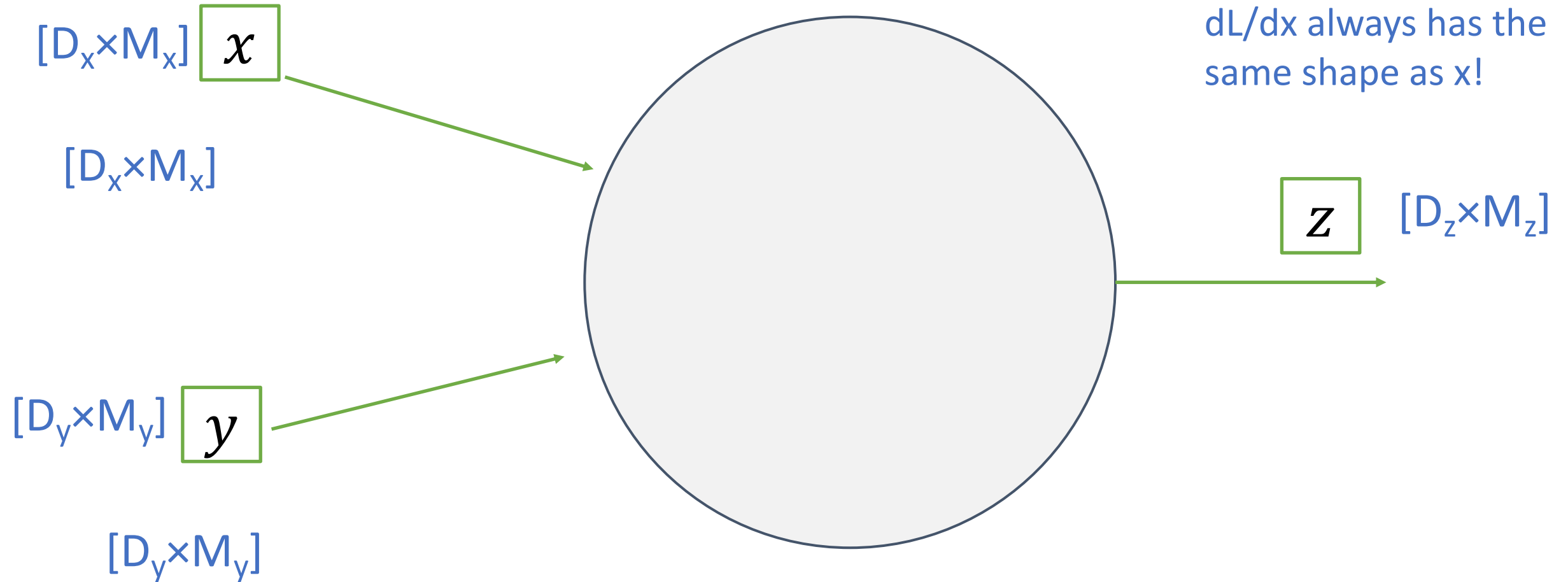
Backprop with Matrices (or Tensors):



Backprop with Matrices (or Tensors):

Loss L still a scalar!

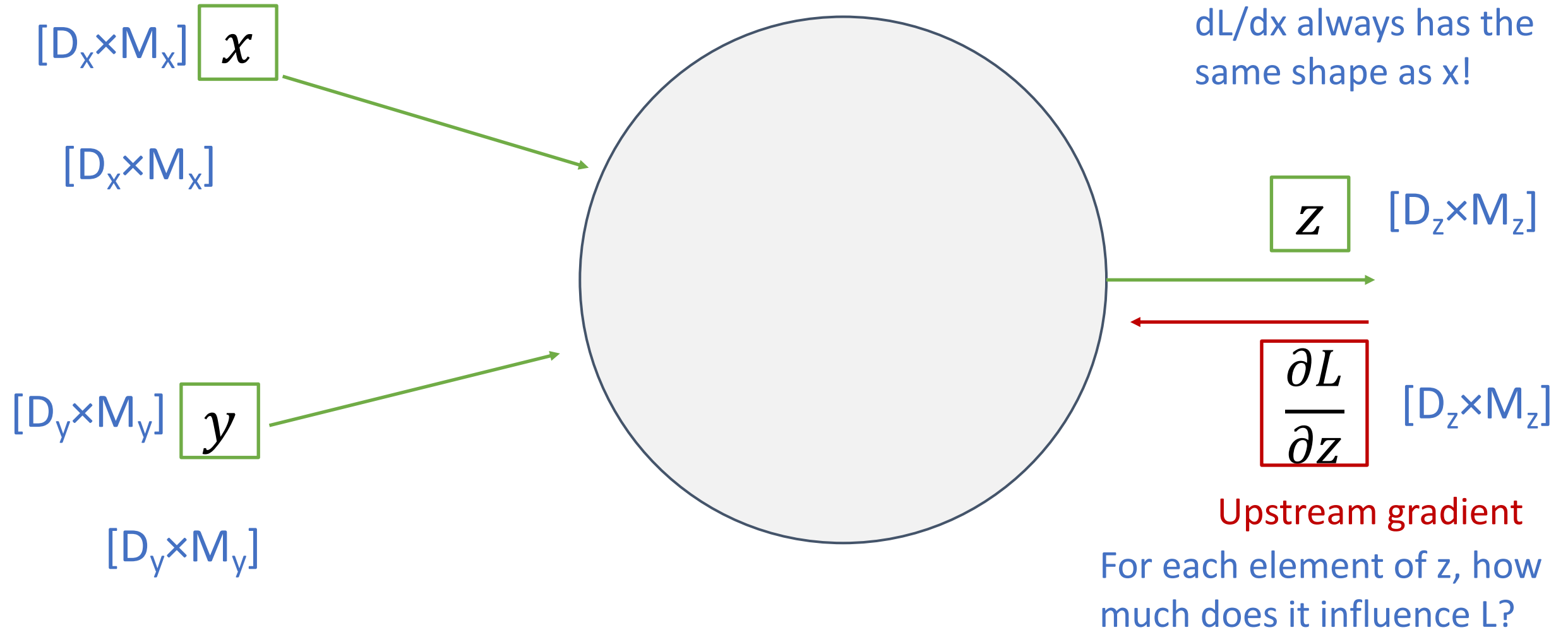
dL/dx always has the same shape as x !



Backprop with Matrices (or Tensors):

Loss L still a scalar!

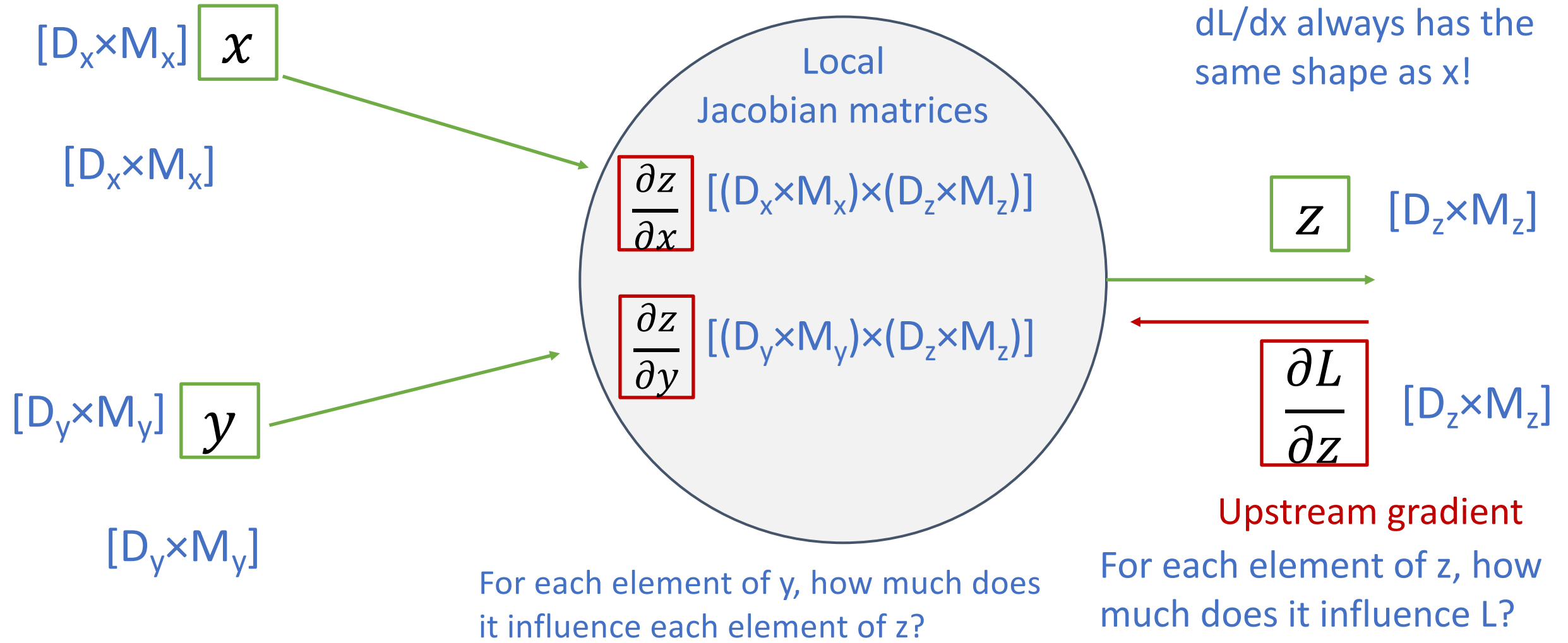
dL/dx always has the same shape as x !



Backprop with Matrices (or Tensors):

Loss L still a scalar!

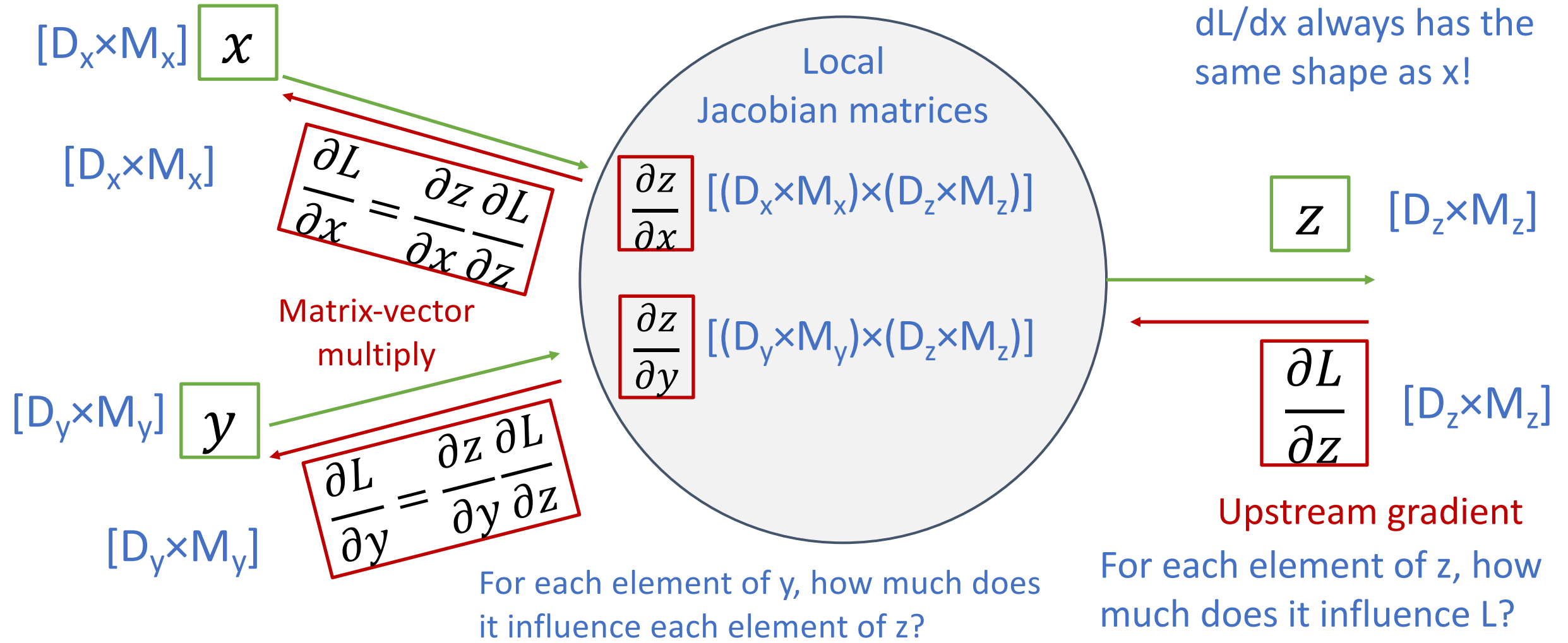
dL/dx always has the same shape as x !



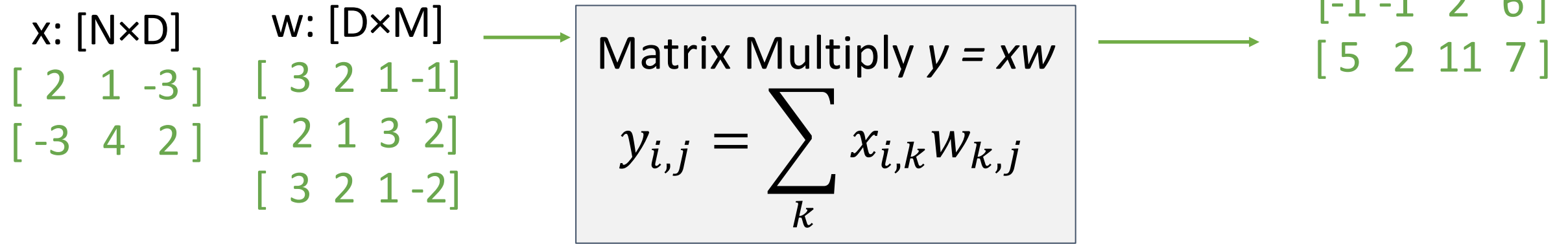
Backprop with Matrices (or Tensors):

Loss L still a scalar!

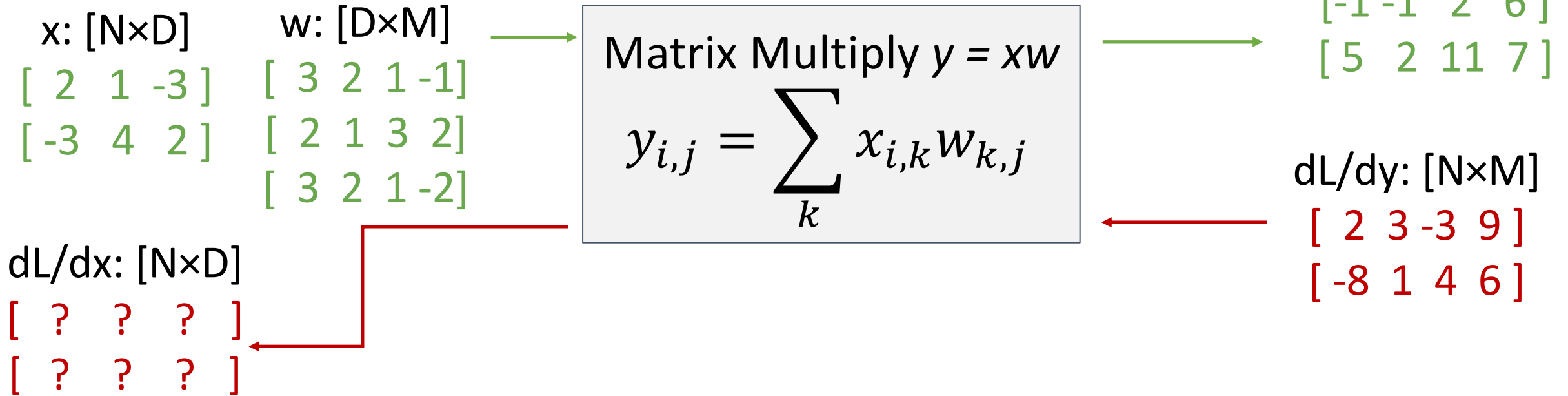
dL/dx always has the same shape as x !



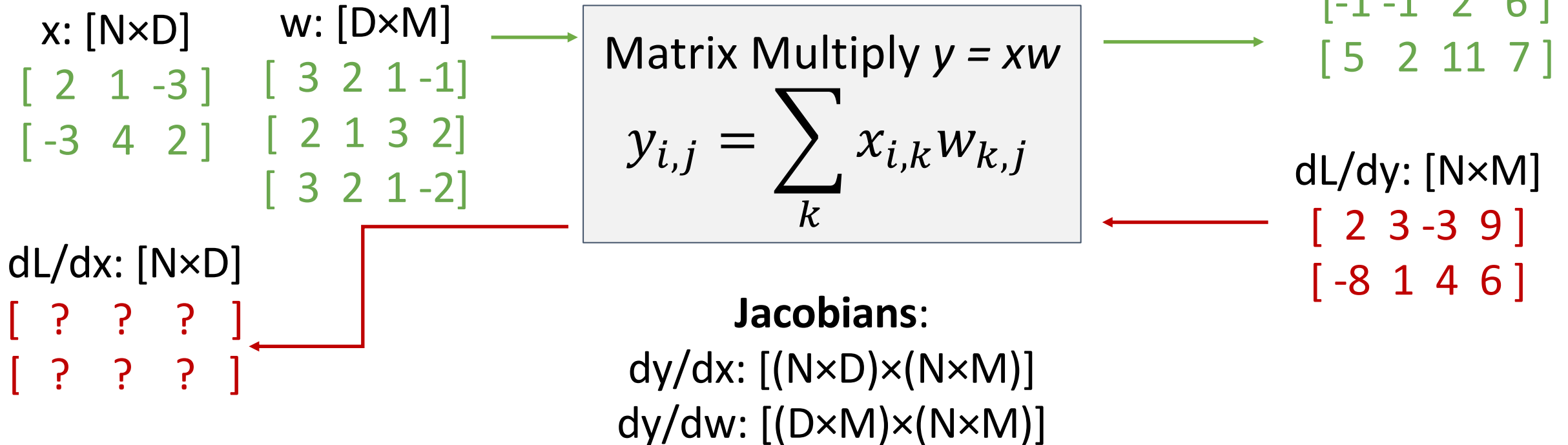
Example: Matrix Multiplication



Example: Matrix Multiplication



Example: Matrix Multiplication

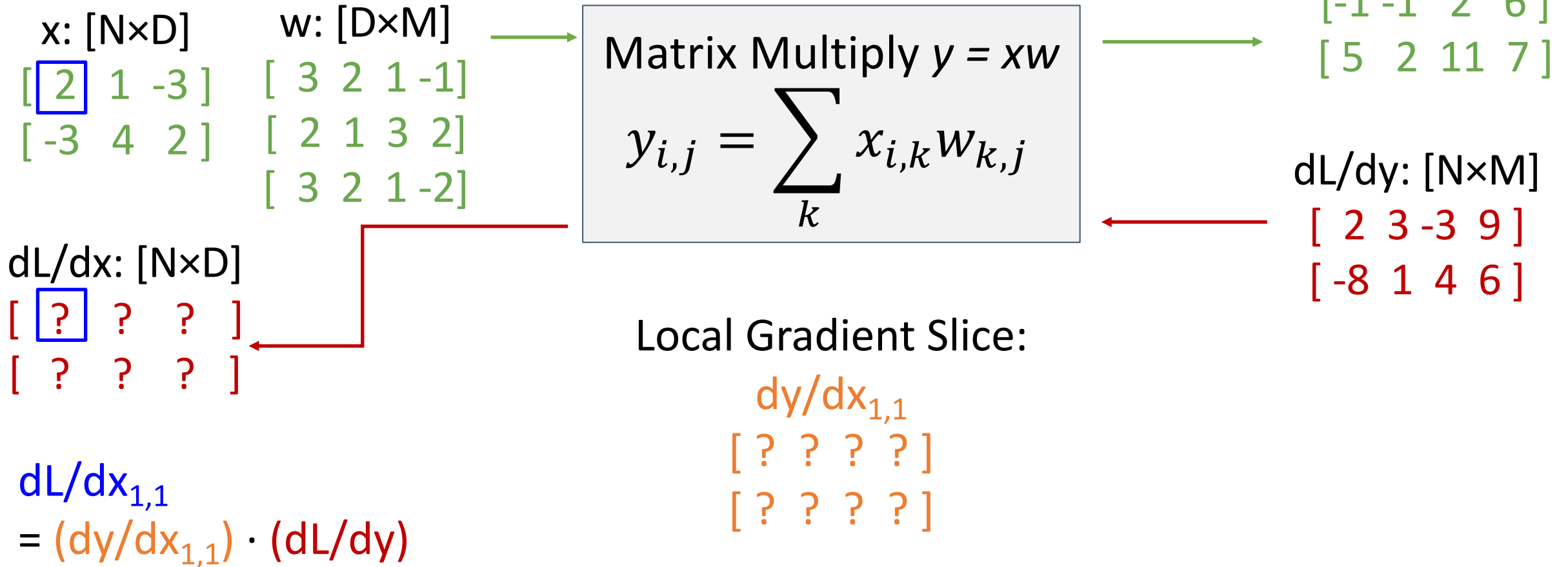


For a neural net we may have

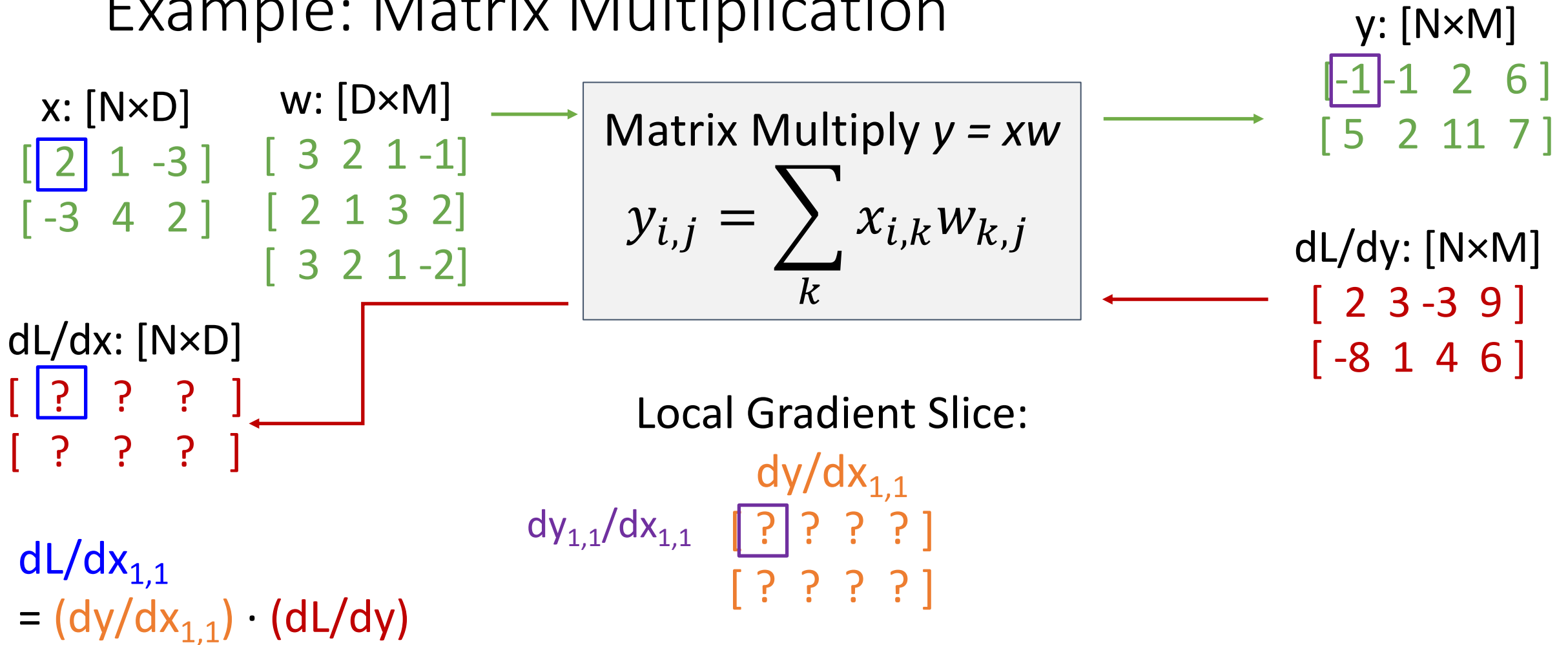
$N=64, D=M=4096$

Each Jacobian takes 256 GB of memory! Must work with them implicitly!

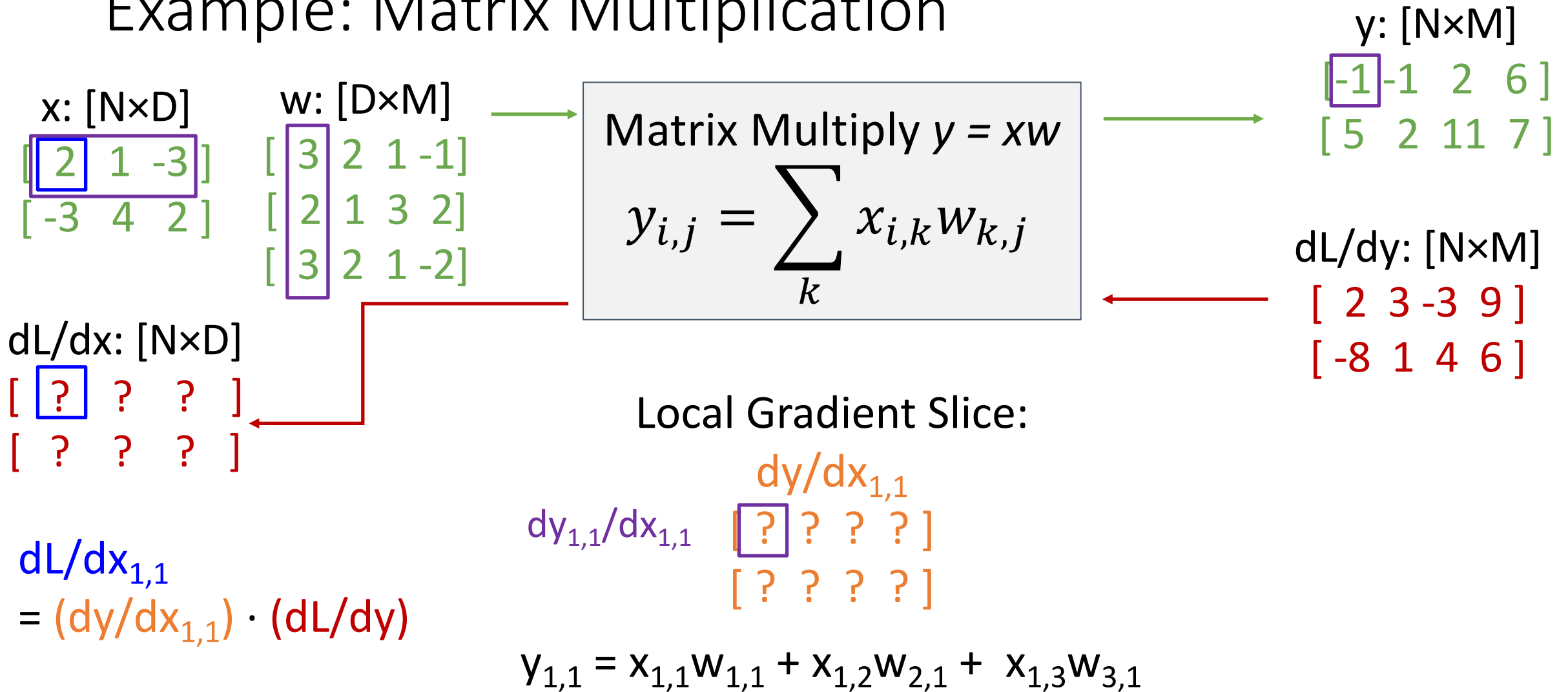
Example: Matrix Multiplication



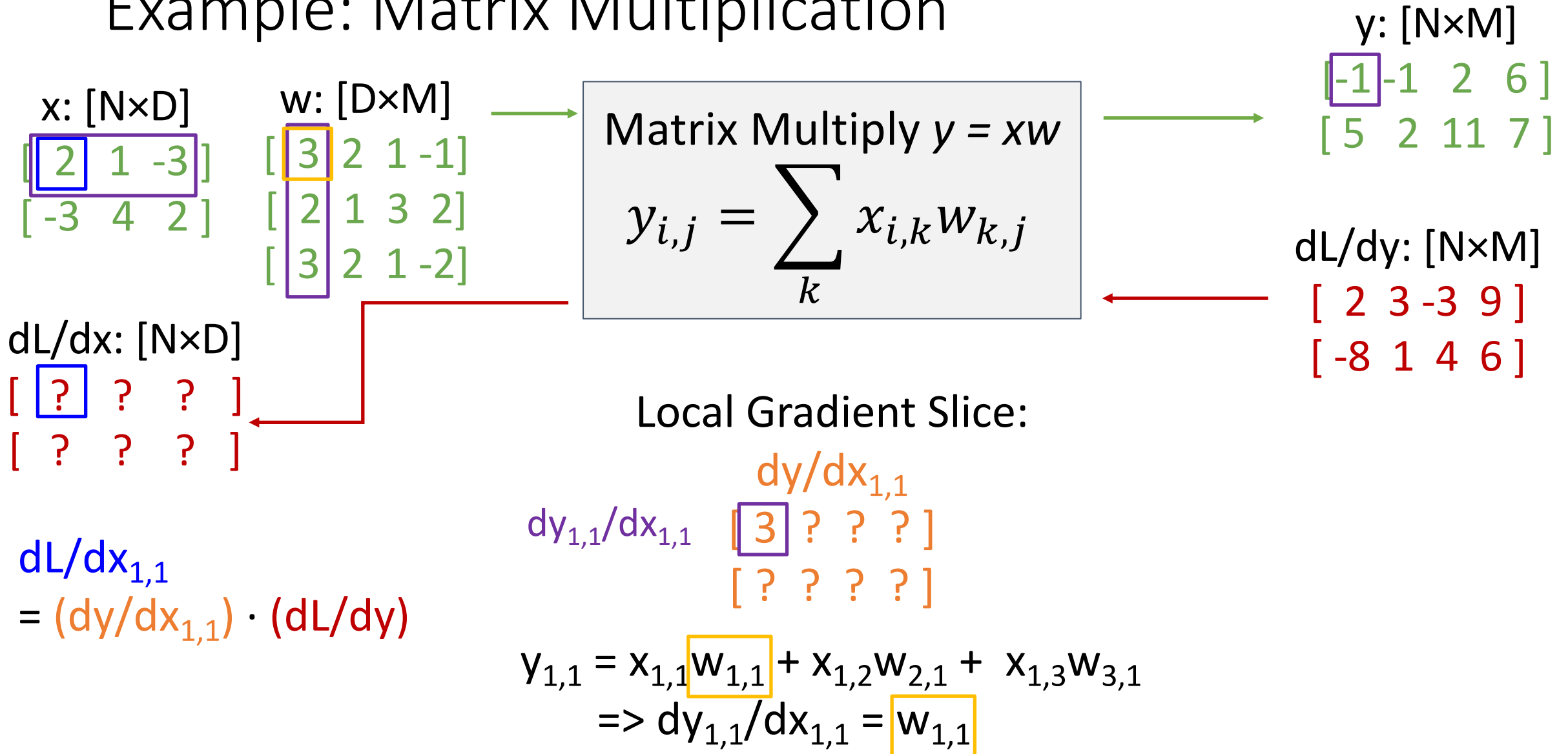
Example: Matrix Multiplication



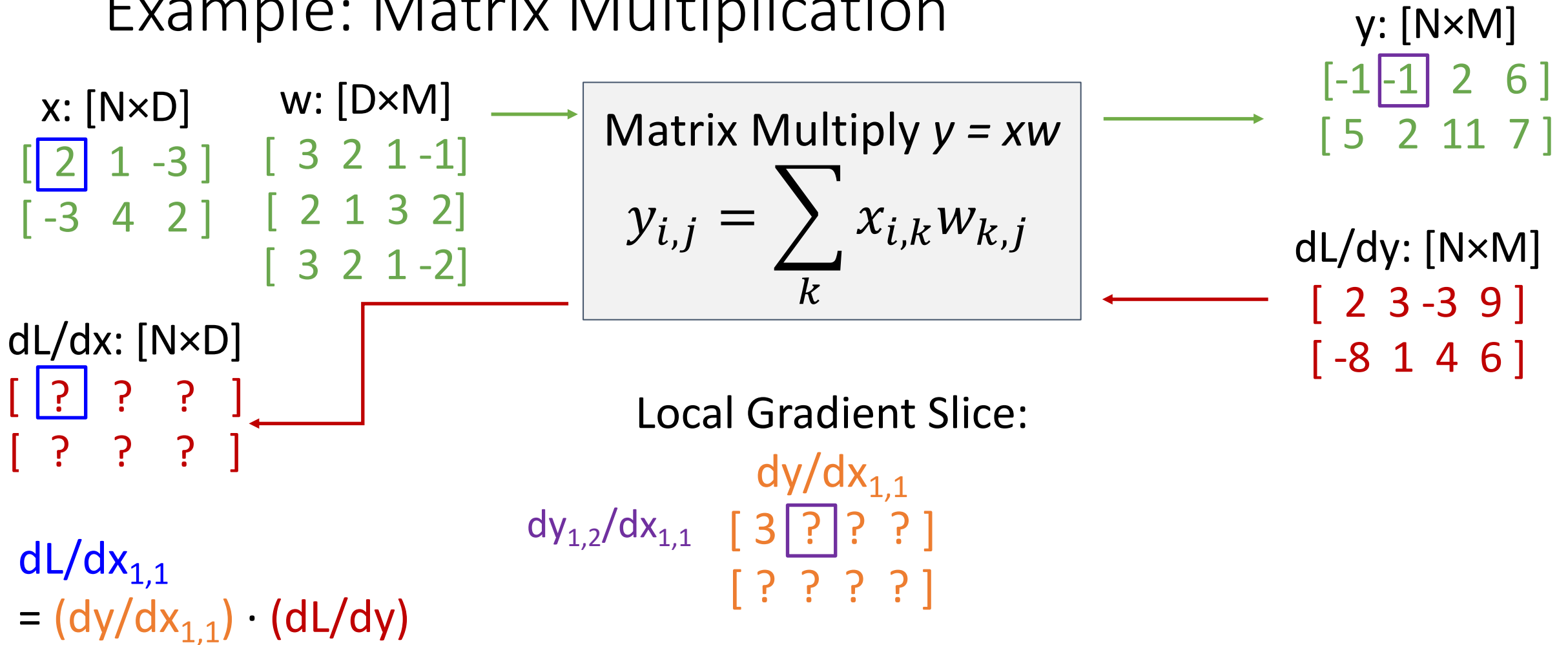
Example: Matrix Multiplication



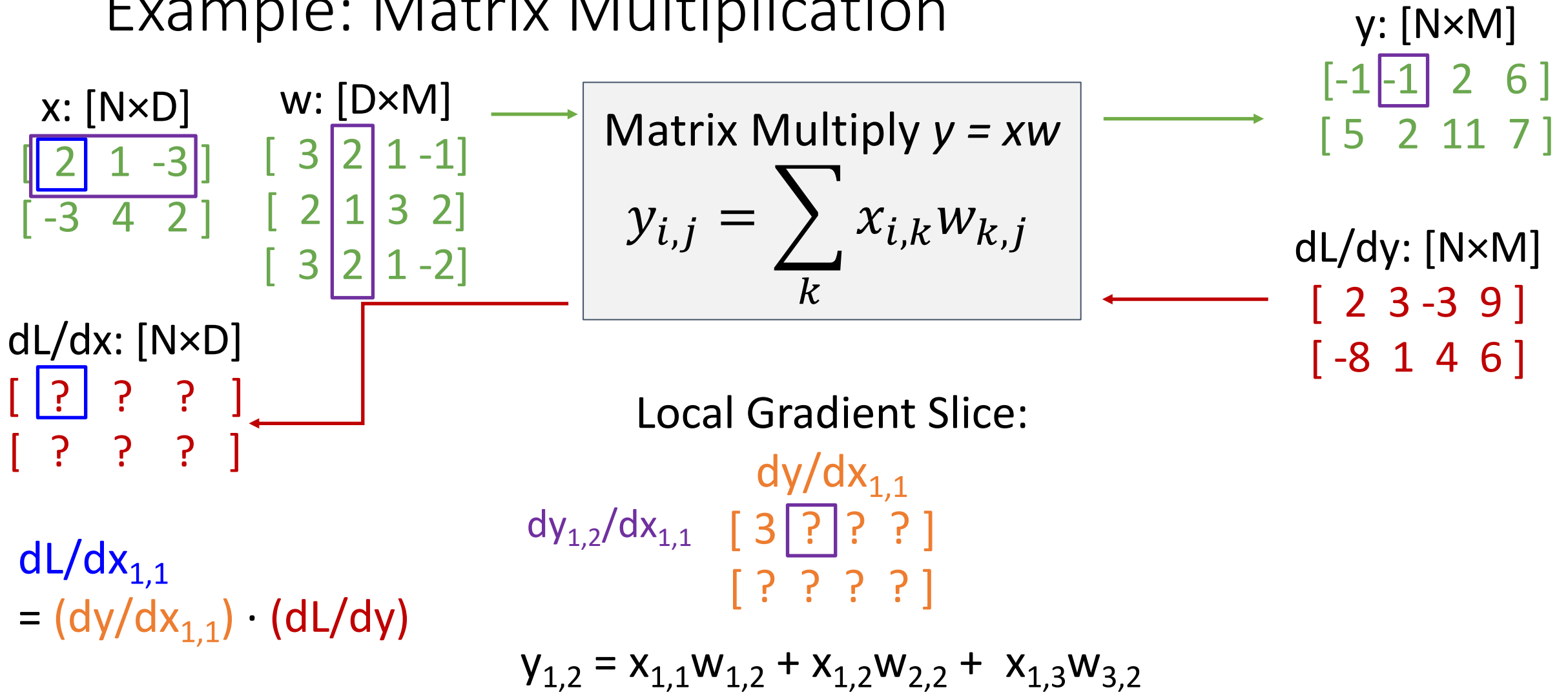
Example: Matrix Multiplication



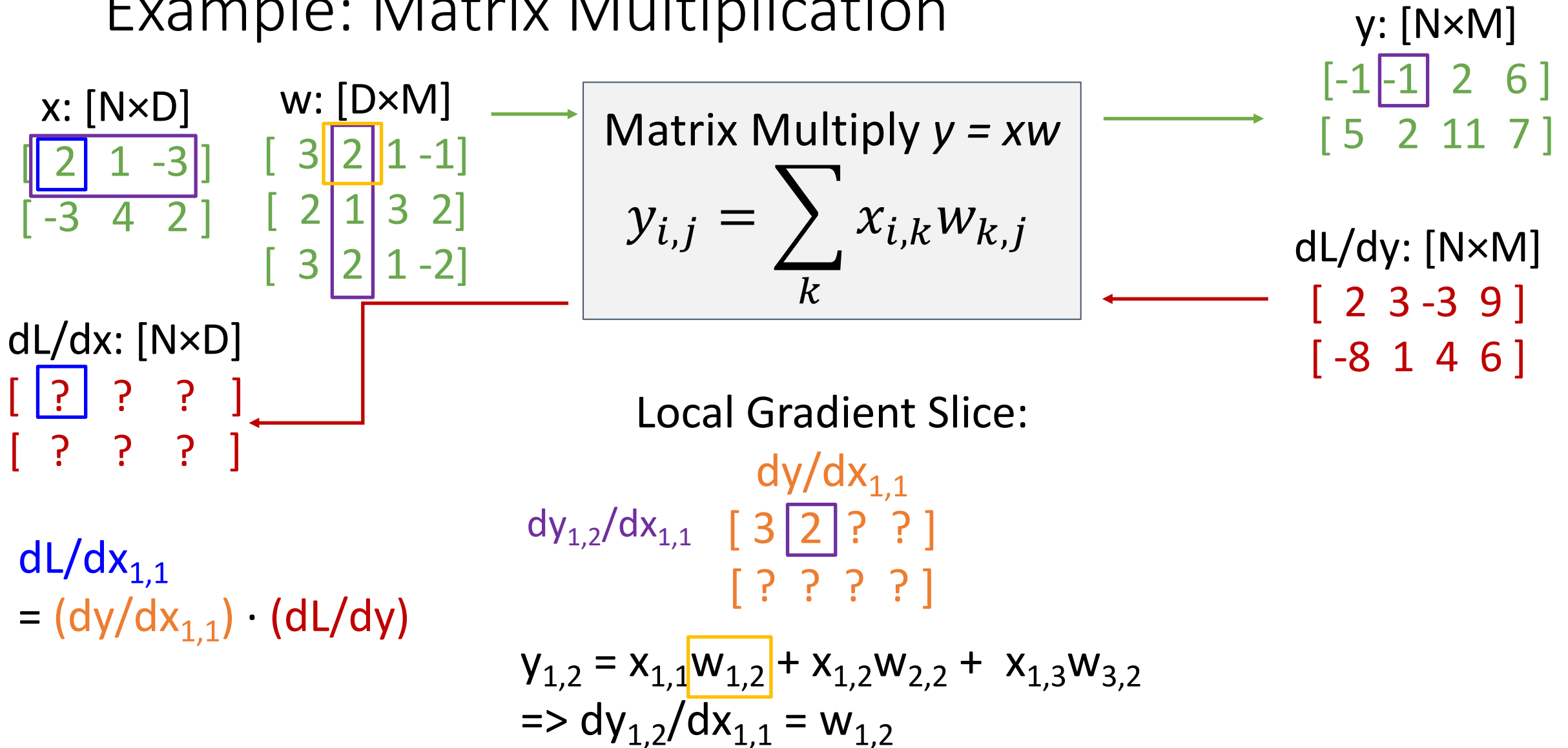
Example: Matrix Multiplication



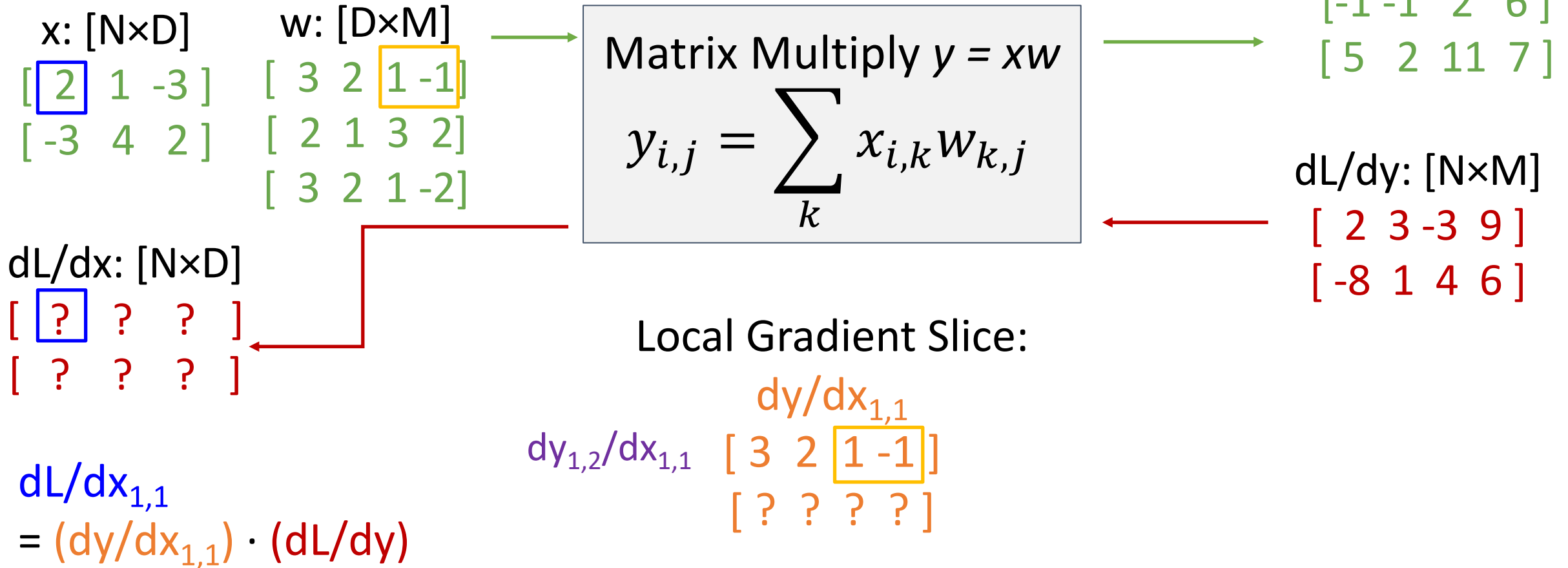
Example: Matrix Multiplication



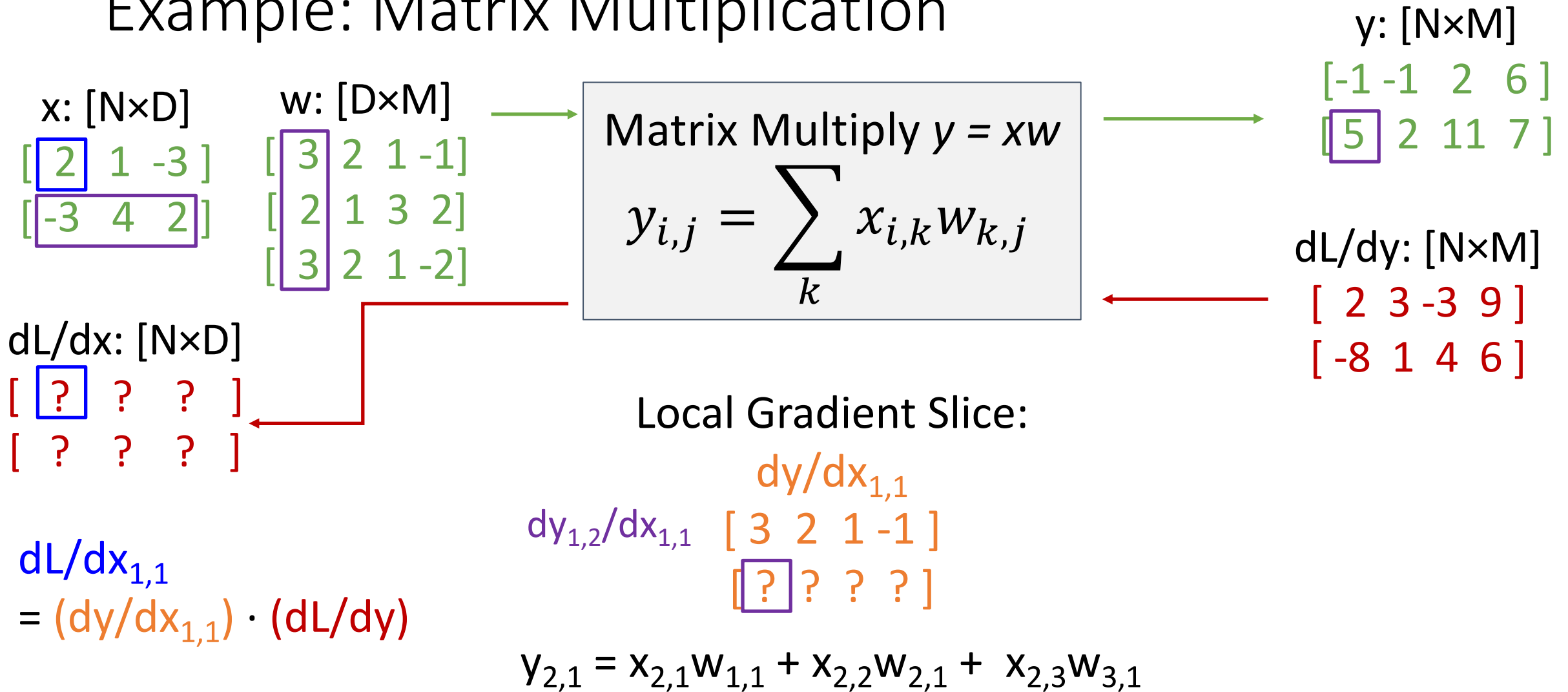
Example: Matrix Multiplication



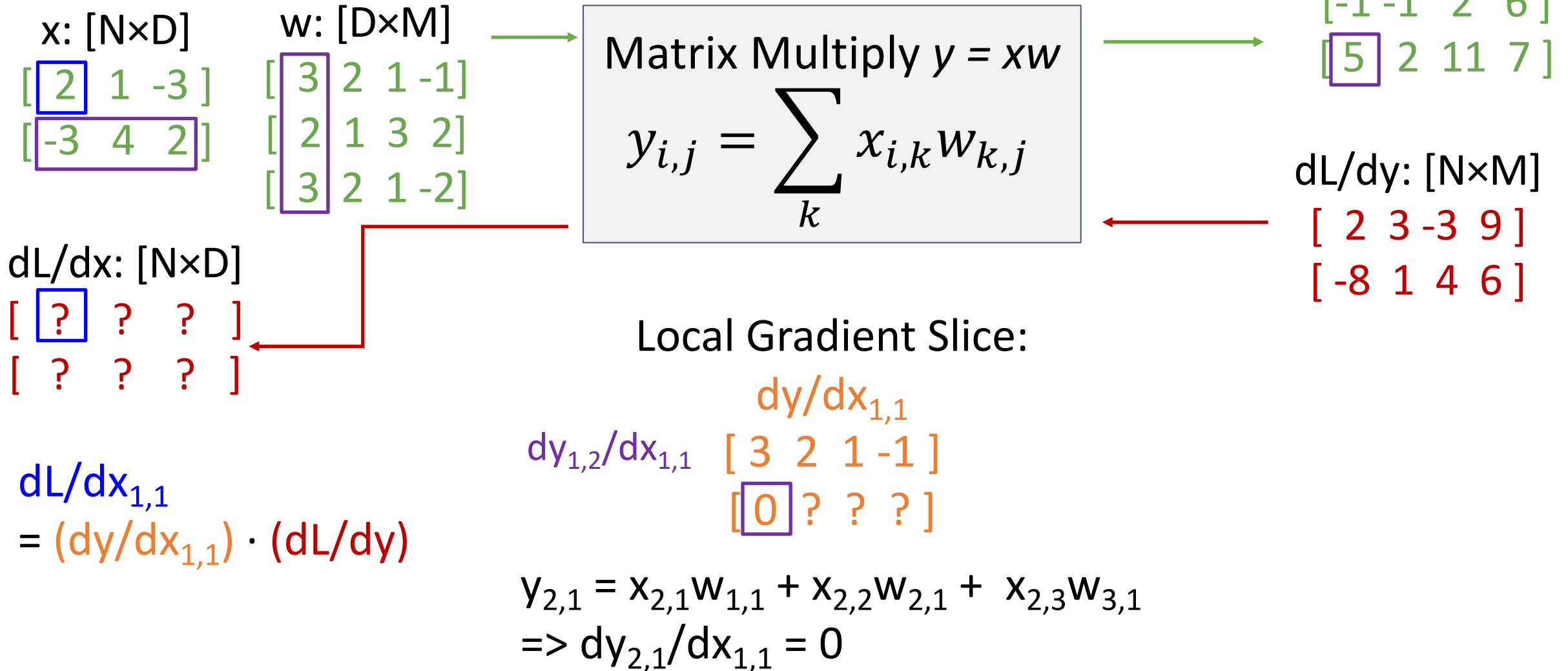
Example: Matrix Multiplication



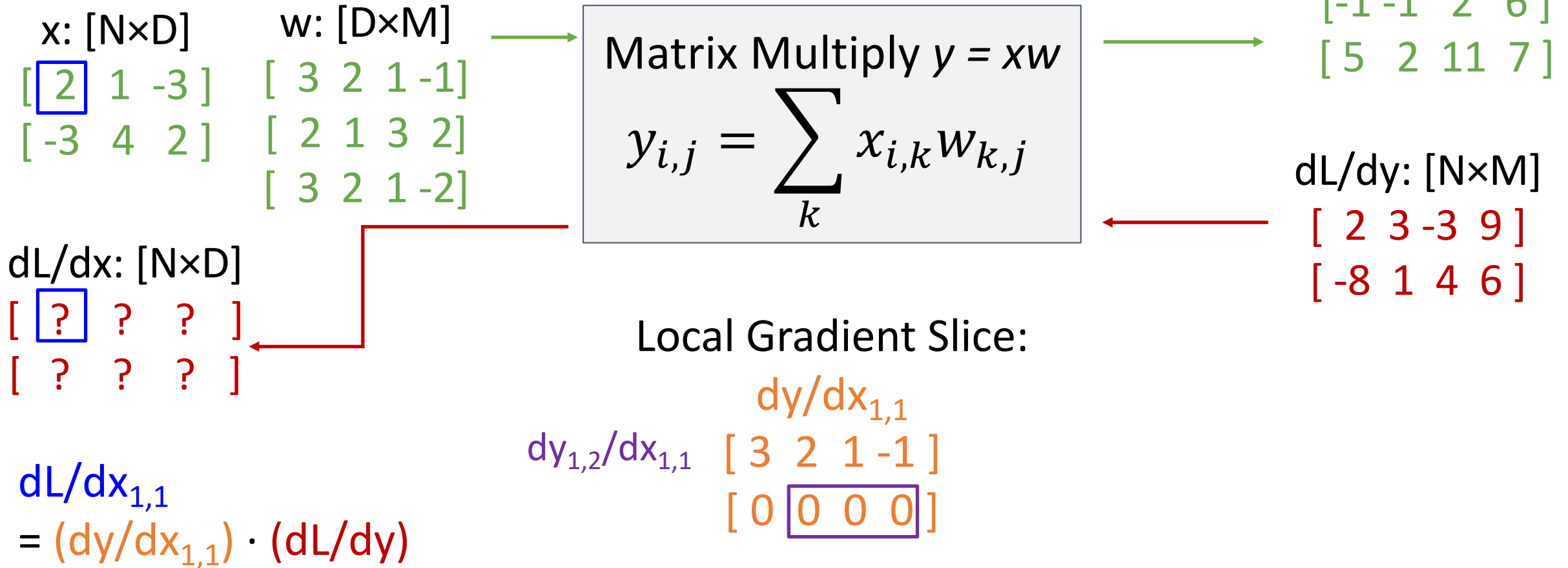
Example: Matrix Multiplication



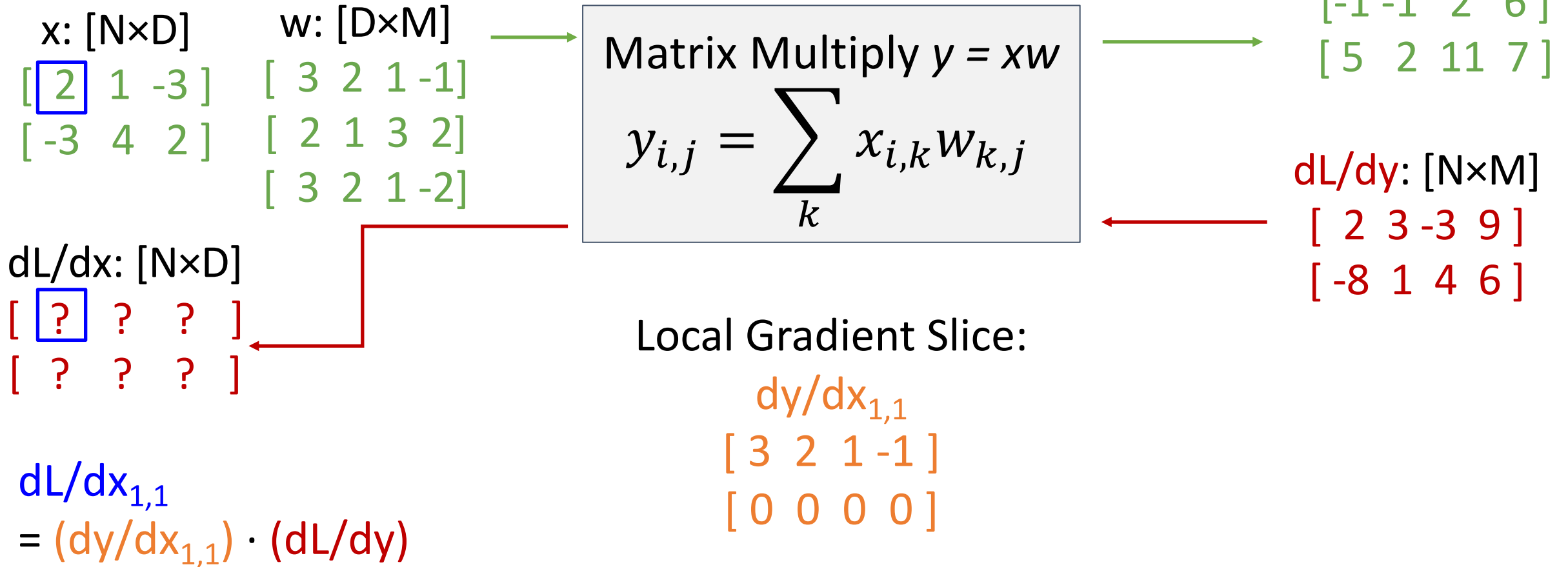
Example: Matrix Multiplication



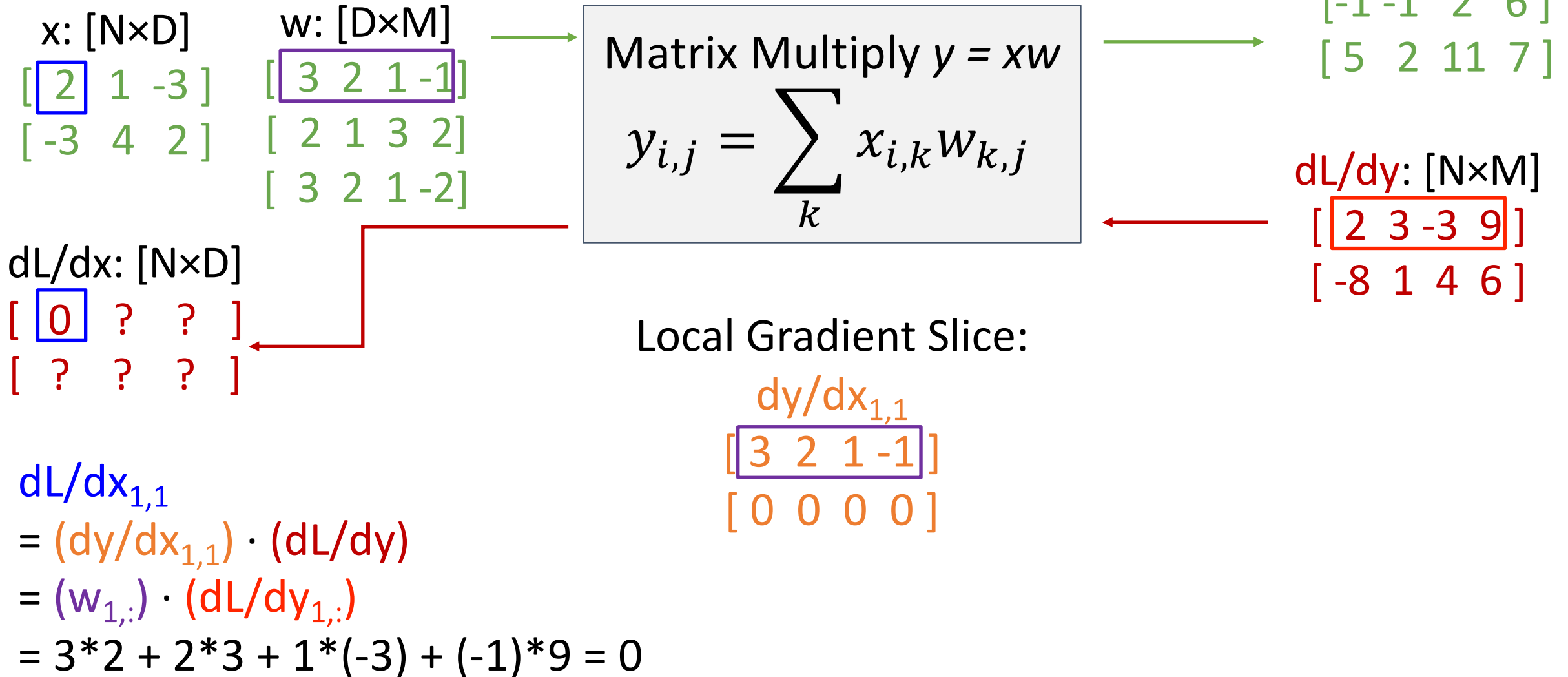
Example: Matrix Multiplication



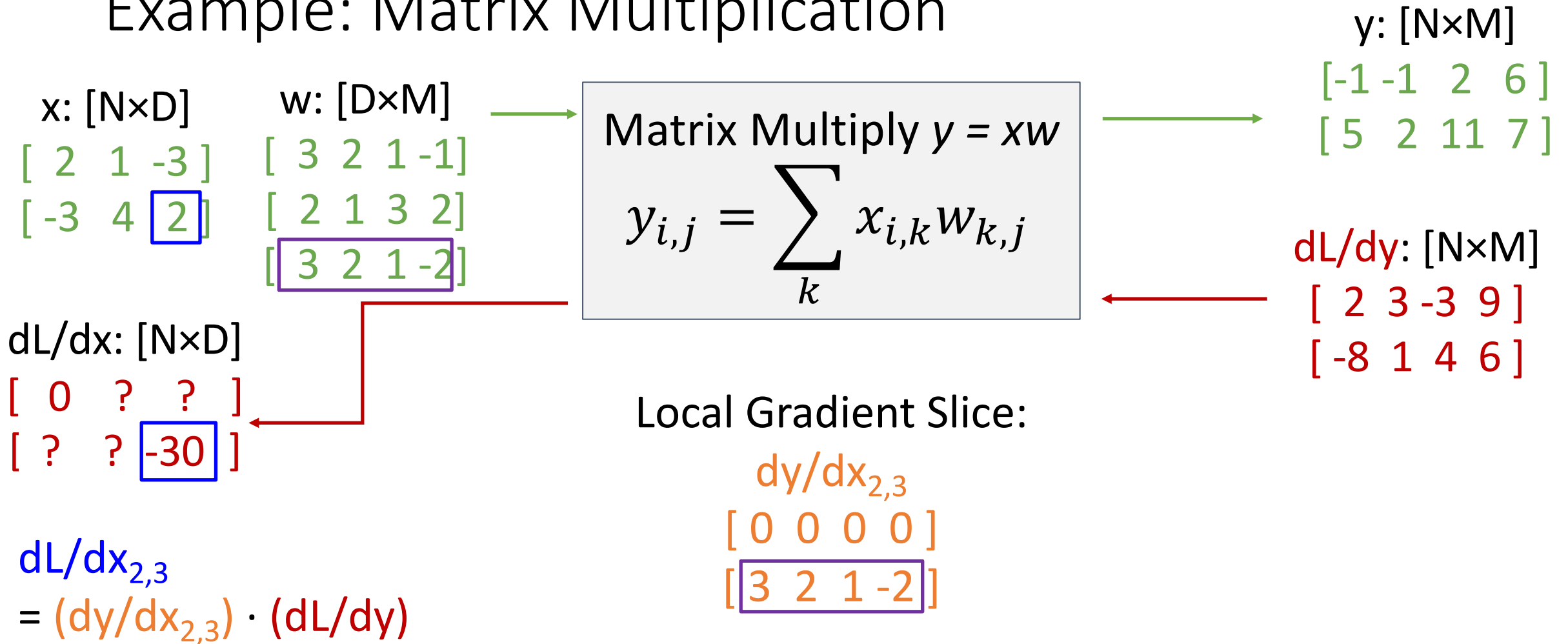
Example: Matrix Multiplication



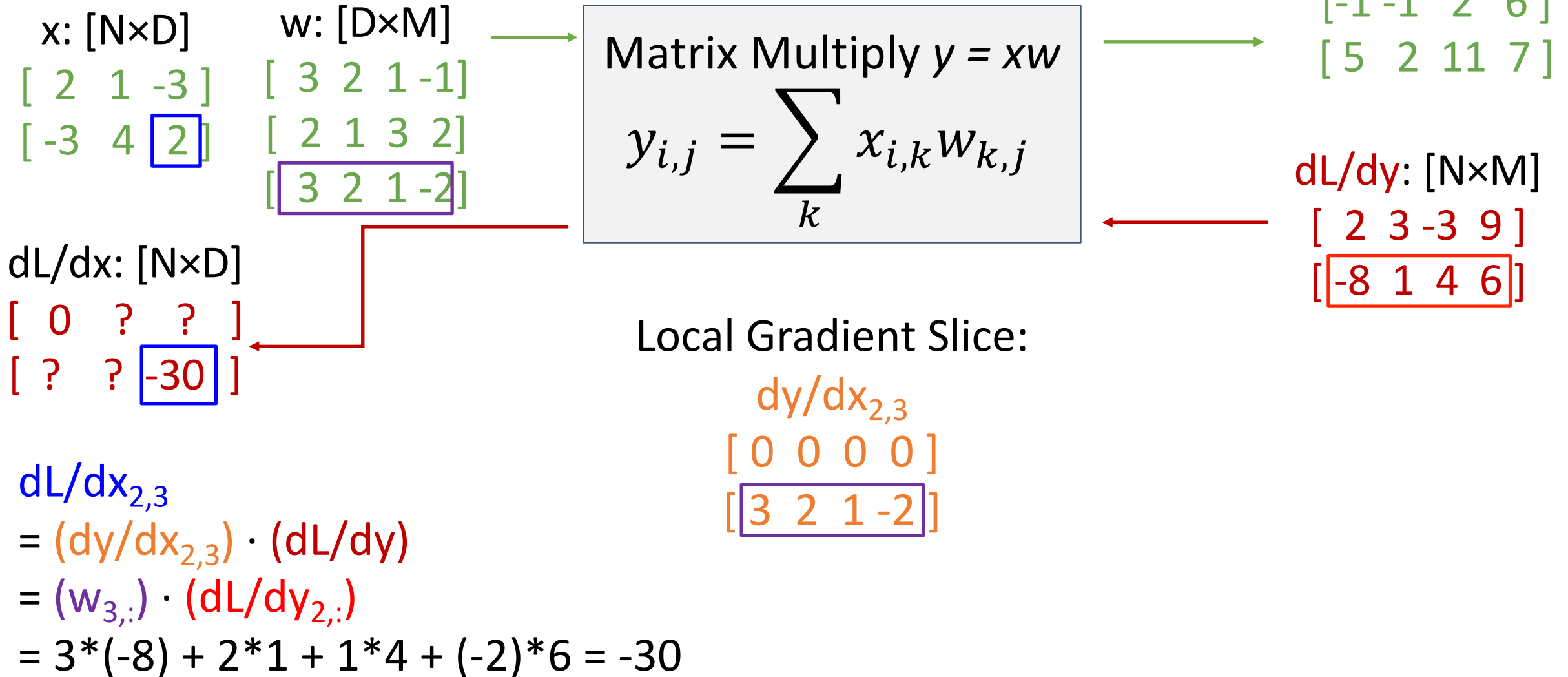
Example: Matrix Multiplication



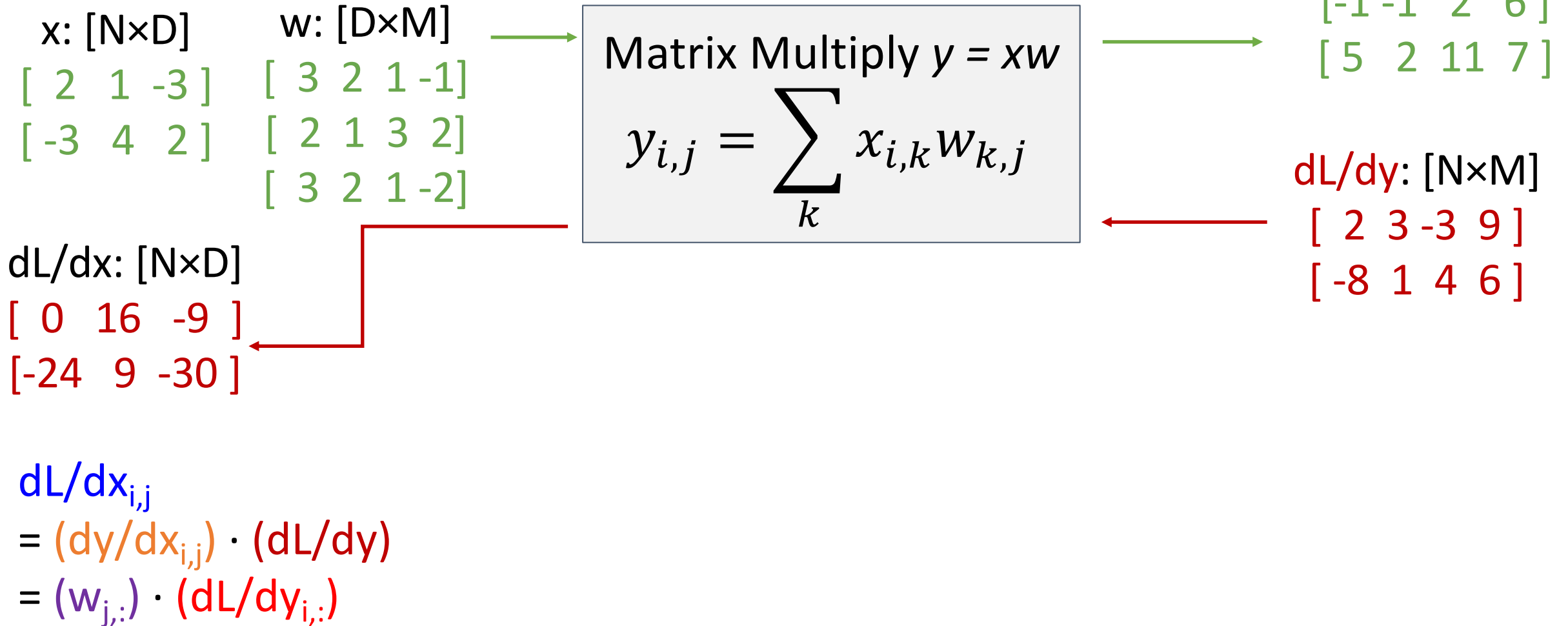
Example: Matrix Multiplication



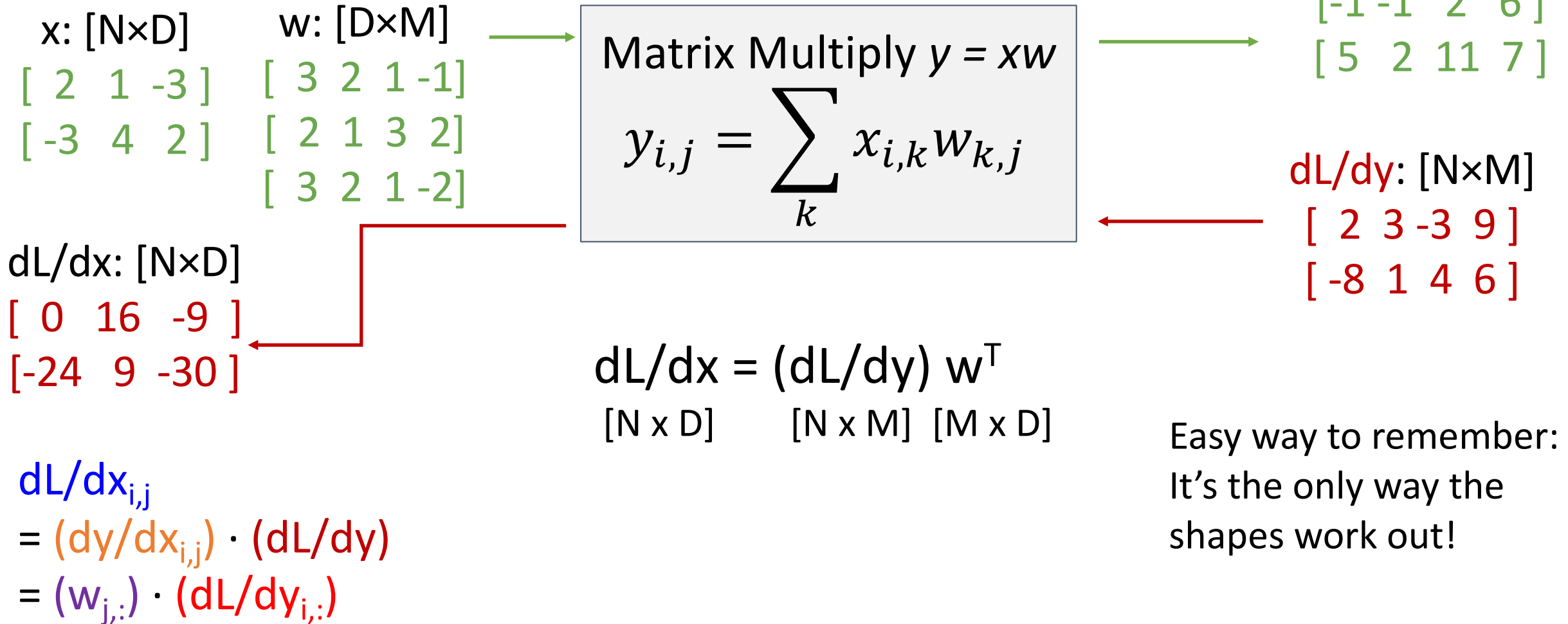
Example: Matrix Multiplication



Example: Matrix Multiplication

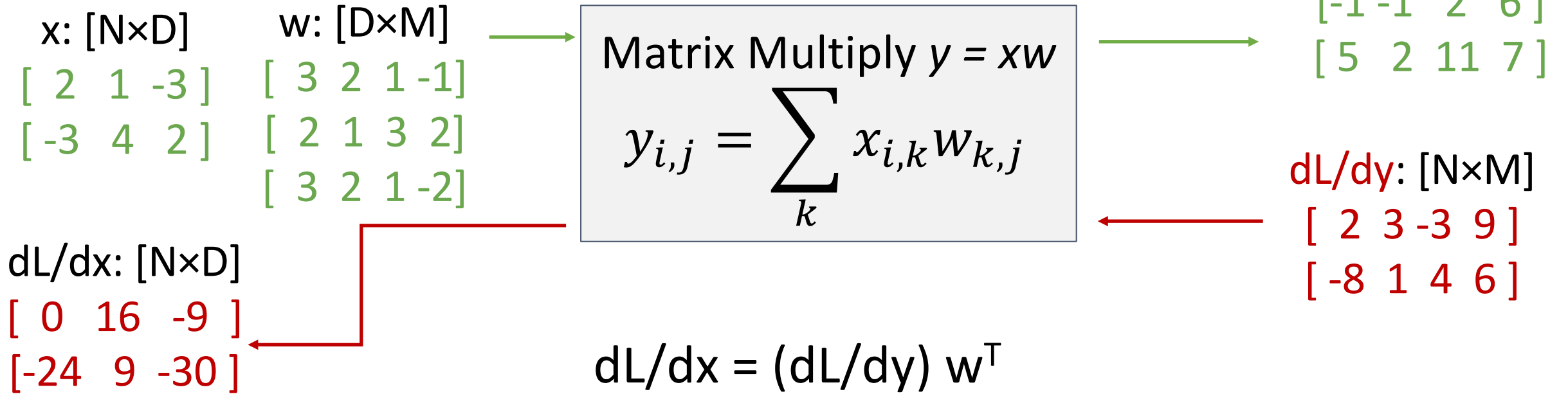


Example: Matrix Multiplication



Easy way to remember:
It's the only way the
shapes work out!

Example: Matrix Multiplication



$$dL/dx = (dL/dy) w^T$$

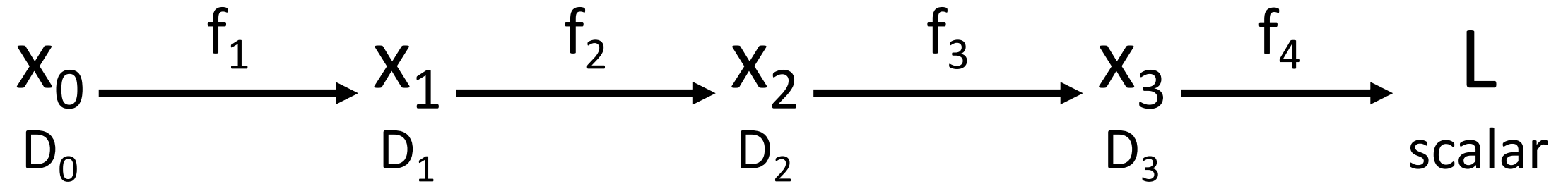
$[N \times D] \quad [N \times M] \quad [M \times D]$

$$dL/dw = x^T (dL/dy)$$

$[D \times M] \quad [D \times N] \quad [N \times M]$

Easy way to remember:
It's the only way the
shapes work out!

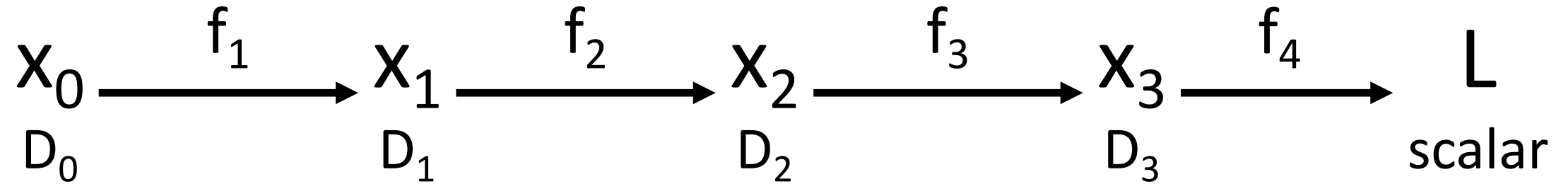
Backpropagation: Another View



Chain rule

$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0} \right) \left(\frac{\partial x_2}{\partial x_1} \right) \left(\frac{\partial x_3}{\partial x_2} \right) \left(\frac{\partial L}{\partial x_3} \right)$$

Backpropagation: Another View

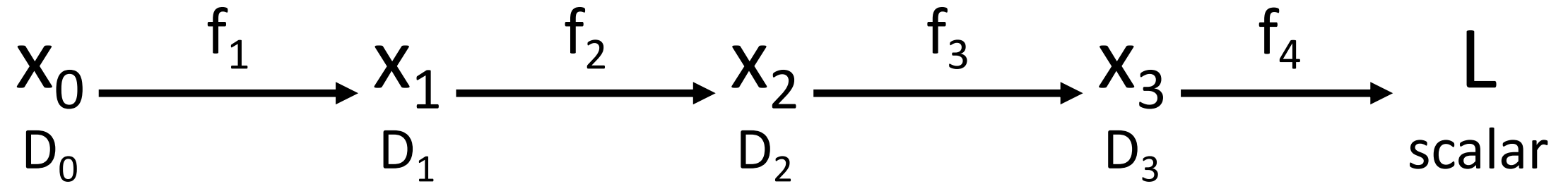


Matrix multiplication is **associative**: we can compute products in any order

Chain rule

$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0} \right) \left(\frac{\partial x_2}{\partial x_1} \right) \left(\frac{\partial x_3}{\partial x_2} \right) \left(\frac{\partial L}{\partial x_3} \right)$$
$$[D_0 \times D_1] \quad [D_1 \times D_2] \quad [D_2 \times D_3] \quad [D_3]$$

Reverse-Mode Automatic Differentiation



Matrix multiplication is **associative**: we can compute products in any order
Computing products right-to-left avoids matrix-matrix products; only needs matrix-vector

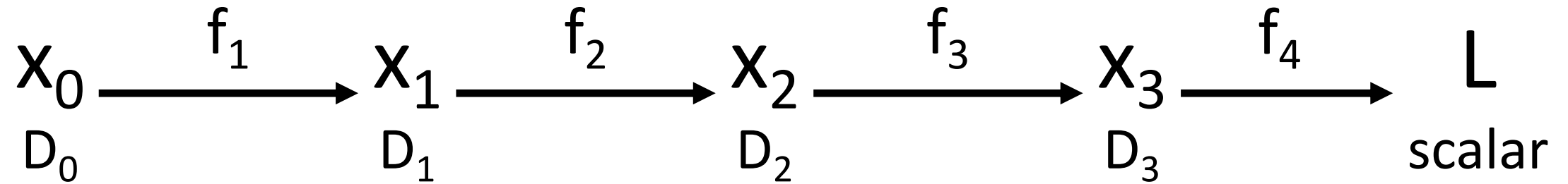
Chain rule

$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0} \right) \left(\frac{\partial x_2}{\partial x_1} \right) \left(\frac{\partial x_3}{\partial x_2} \right) \left(\frac{\partial L}{\partial x_3} \right)$$

$[D_0 \times D_1] \quad [D_1 \times D_2] \quad [D_2 \times D_3] \quad [D_3]$

A long arrow points from the rightmost term $(\frac{\partial L}{\partial x_3})$ to the leftmost term $(\frac{\partial x_1}{\partial x_0})$, indicating the reverse flow of the chain rule.

Reverse-Mode Automatic Differentiation



Matrix multiplication is **associative**: we can compute products in any order
 Computing products right-to-left avoids matrix-matrix products; only needs matrix-vector

Chain rule

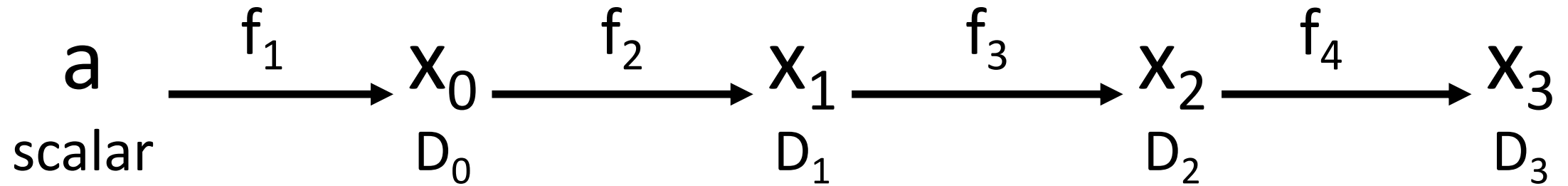
$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0} \right) \left(\frac{\partial x_2}{\partial x_1} \right) \left(\frac{\partial x_3}{\partial x_2} \right) \left(\frac{\partial L}{\partial x_3} \right)$$

$[D_0 \times D_1] \quad [D_1 \times D_2] \quad [D_2 \times D_3] \quad [D_3]$

What if we want grads of scalar input w/respect to vector outputs?

Compute grad of scalar output w/respect to all vector inputs

Forward-Mode Automatic Differentiation

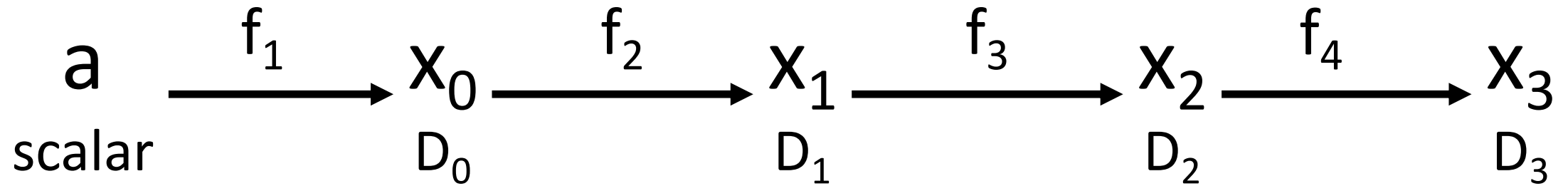


Chain rule

$$\frac{\partial x_3}{\partial a} = \left(\frac{\partial x_0}{\partial a} \right) \left(\frac{\partial x_1}{\partial x_0} \right) \left(\frac{\partial x_2}{\partial x_1} \right) \left(\frac{\partial x_3}{\partial x_2} \right)$$

$[D_0] \quad [D_0 \times D_1] \quad [D_1 \times D_2] \quad [D_2 \times D_3]$

Forward-Mode Automatic Differentiation



Computing products left-to-right avoids matrix-matrix products; only needs matrix-vector

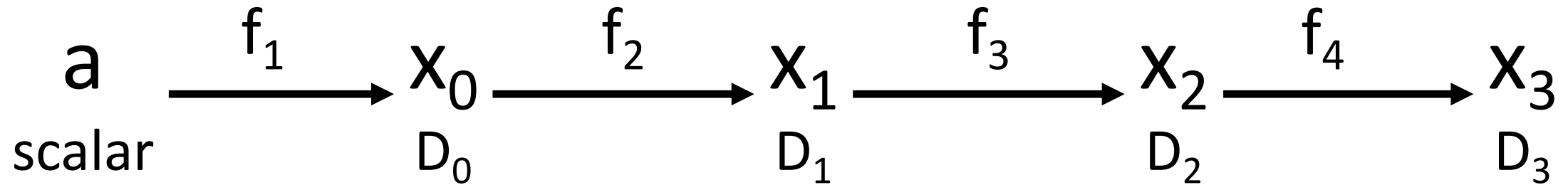
Chain rule

$$\frac{\partial x_3}{\partial a} = \left(\frac{\partial x_0}{\partial a} \right) \left(\frac{\partial x_1}{\partial x_0} \right) \left(\frac{\partial x_2}{\partial x_1} \right) \left(\frac{\partial x_3}{\partial x_2} \right)$$

$[D_0] \quad [D_0 \times D_1] \quad [D_1 \times D_2] \quad [D_2 \times D_3]$

A long horizontal arrow points from the first term $\left(\frac{\partial x_0}{\partial a} \right)$ to the last term $\left(\frac{\partial x_3}{\partial x_2} \right)$ in the product.

Forward-Mode Automatic Differentiation



Computing products left-to-right avoids matrix-matrix products; only needs matrix-vector

Beta implementation in PyTorch! https://pytorch.org/tutorials/intermediate/forward_ad_usage.html

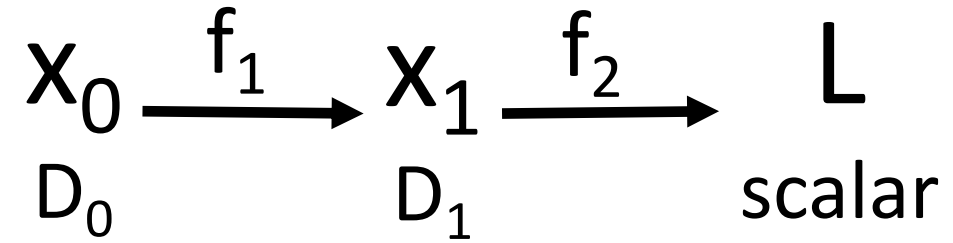
Chain
rule

$$\frac{\partial x_3}{\partial a} = \left(\frac{\partial x_0}{\partial a} \right) \left(\frac{\partial x_1}{\partial x_0} \right) \left(\frac{\partial x_2}{\partial x_1} \right) \left(\frac{\partial x_3}{\partial x_2} \right)$$

$[D_0] \quad [D_0 \times D_1] \quad [D_1 \times D_2] \quad [D_2 \times D_3]$

You can also
implement
forward-mode AD
using [two calls to
reverse-mode AD!](#)
(Inefficient but
elegant)

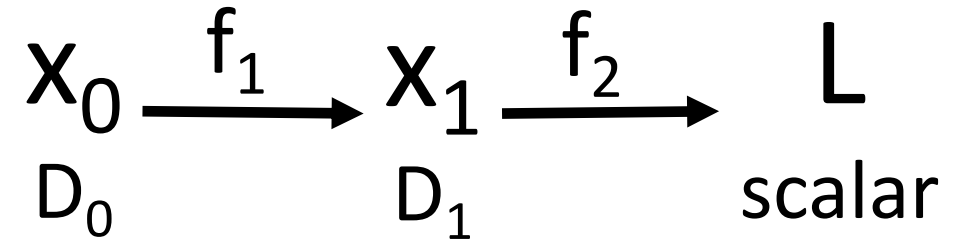
Backprop: Higher-Order Derivatives



$$\frac{\partial^2 L}{\partial x_0^2}$$

Hessian matrix
H of second
derivatives.
[$D_0 \times D_0$]

Backprop: Higher-Order Derivatives



Hessian / vector multiply

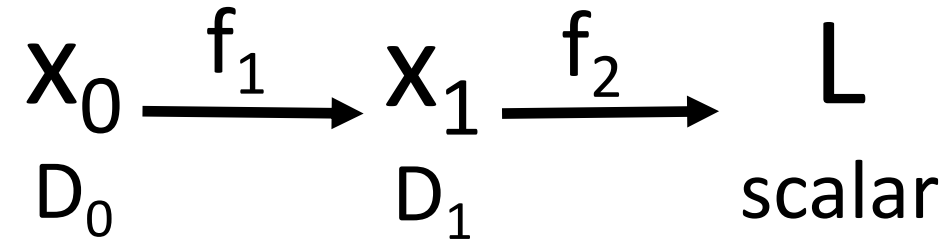
$$\frac{\partial^2 L}{\partial x_0^2}$$

Hessian matrix
H of second derivatives.
[$D_0 \times D_0$]

$$\frac{\partial^2 L}{\partial x_0^2} v$$

[$D_0 \times D_0$] [D_0]

Backprop: Higher-Order Derivatives



Hessian / vector multiply

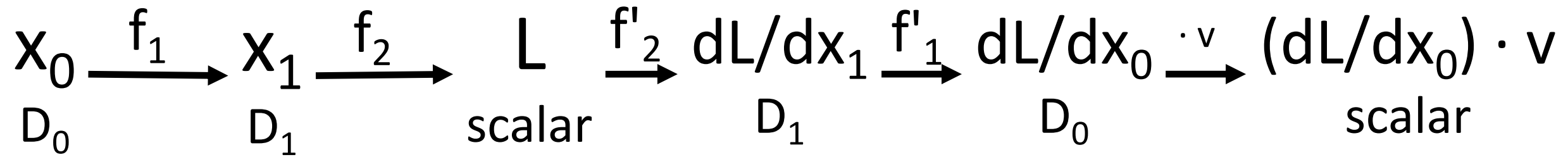
$$\frac{\partial^2 L}{\partial x_0^2}$$

Hessian matrix
H of second derivatives.
[D₀ x D₀]

$$\frac{\partial^2 L}{\partial x_0^2} v = \frac{\partial}{\partial x_0} \left[\frac{\partial L}{\partial x_0} \cdot v \right]$$

(if v doesn't depend on x₀)
[D₀ x D₀] [D₀]

Backprop: Higher-Order Derivatives



Hessian / vector multiply

$$\frac{\partial^2 L}{\partial x_0^2} \quad \text{Hessian matrix}$$

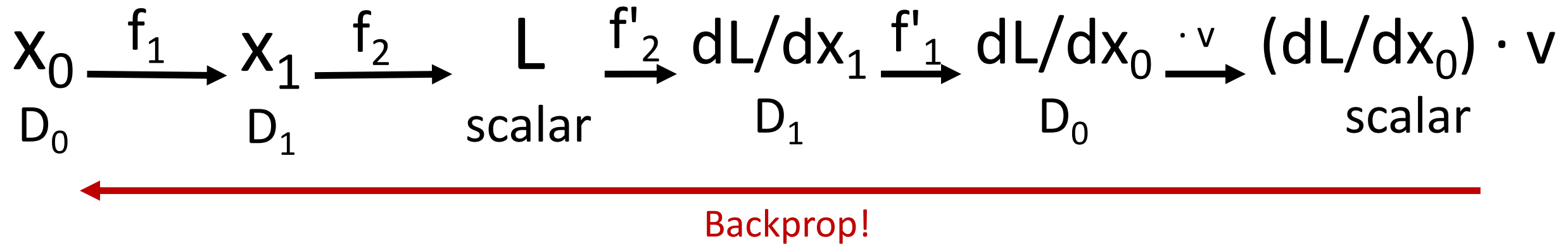
H of second derivatives.

$[D_0 \times D_0]$

$$\frac{\partial^2 L}{\partial x_0^2} v = \frac{\partial}{\partial x_0} \left[\frac{\partial L}{\partial x_0} \cdot v \right] \quad (\text{if } v \text{ doesn't depend on } x_0)$$

$[D_0 \times D_0] [D_0]$

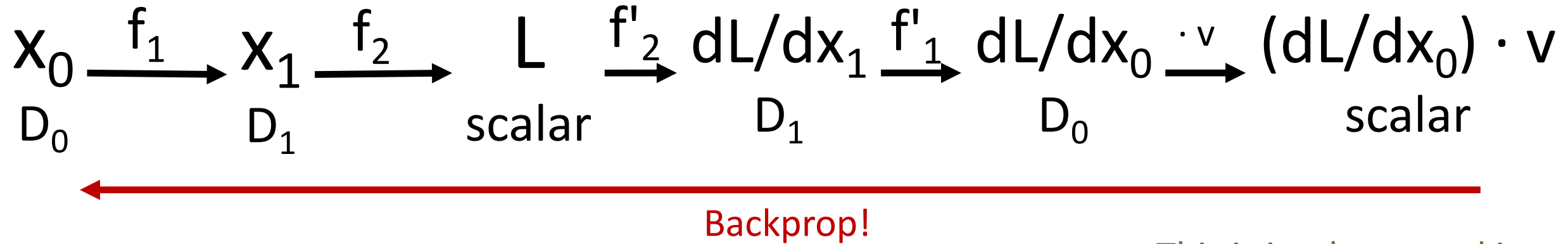
Backprop: Higher-Order Derivatives



Hessian matrix
 $\frac{\partial^2 L}{\partial x_0^2}$ H of second derivatives.
 $[D_0 \times D_0]$

Hessian / vector multiply
 $\frac{\partial^2 L}{\partial x_0^2} v = \frac{\partial}{\partial x_0} \left[\frac{\partial L}{\partial x_0} \cdot v \right]$ (if v doesn't depend on x_0)
 $[D_0 \times D_0] [D_0]$

Backprop: Higher-Order Derivatives



This is implemented in
PyTorch / Tensorflow!

$$\frac{\partial^2 L}{\partial x_0^2}$$

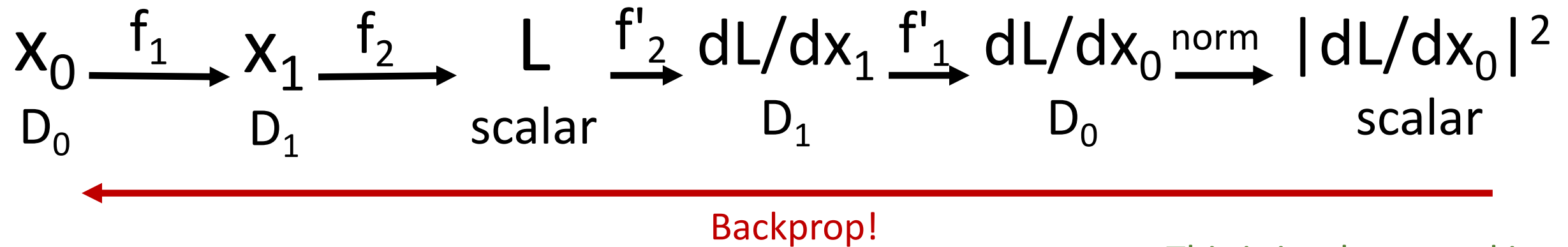
Hessian matrix
H of second
derivatives.
[$D_0 \times D_0$]

Hessian / vector multiply

$$\frac{\partial^2 L}{\partial x_0^2} v = \frac{\partial}{\partial x_0} \left[\frac{\partial L}{\partial x_0} \cdot v \right]$$

(if v doesn't
depend on x_0)
[$D_0 \times D_0$] [D_0]

Backprop: Higher-Order Derivatives



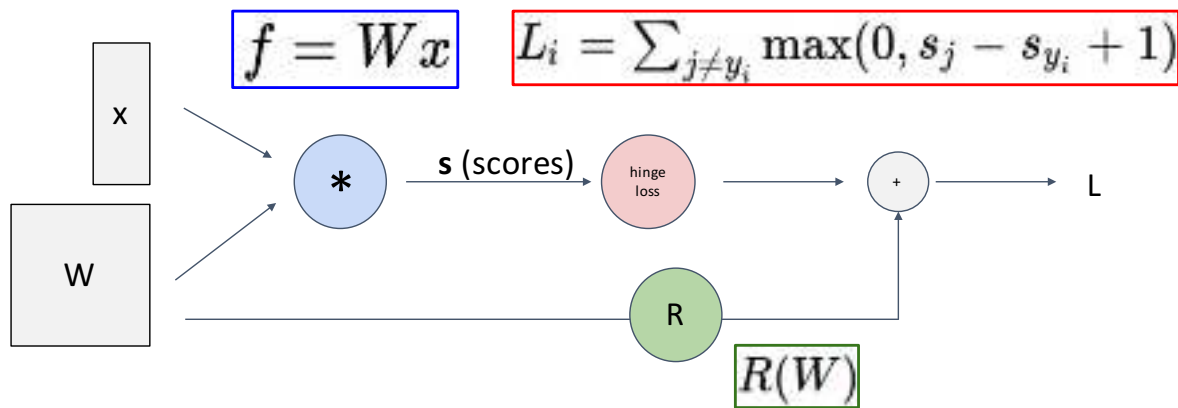
This is implemented in
PyTorch / Tensorflow!

Example: Regularization to penalize the norm of the gradient

$$R(W) = \left\| \frac{\partial L}{\partial W} \right\|_2^2 = \left(\frac{\partial L}{\partial W} \right) \cdot \left(\frac{\partial L}{\partial W} \right) \quad \frac{\partial}{\partial x_0} [R(W)] = 2 \left(\frac{\partial^2 L}{\partial x_0^2} \right) \left(\frac{\partial L}{\partial x_0} \right)$$

Summary

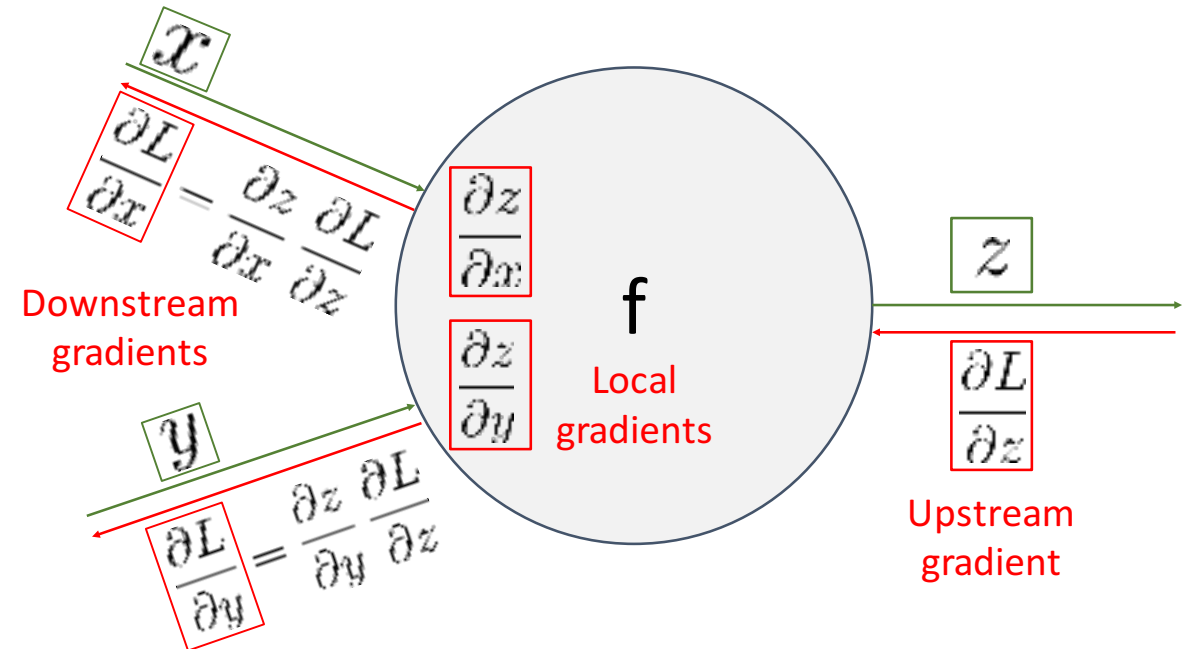
Represent complex expressions
as **computational graphs**



Forward pass computes outputs

Backward pass computes gradients

During the backward pass, each node in the graph receives **upstream gradients** and multiplies them by **local gradients** to compute **downstream gradients**



Summary

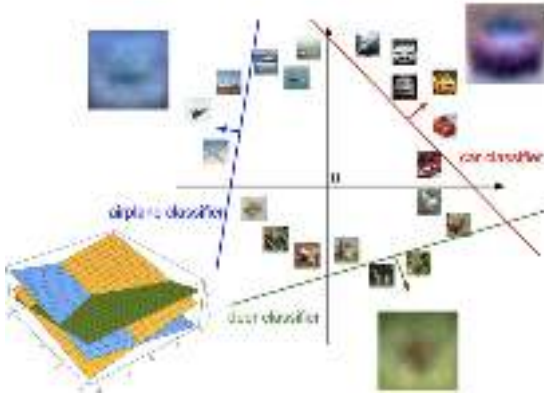
Backprop can be implemented with “flat” code where the backward pass looks like forward pass reversed (Use this for A2!)

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)  
  
    grad_L = 1.0  
    grad_s3 = grad_L * (1 - L) * L  
    grad_w2 = grad_s3  
    grad_s2 = grad_s3  
    grad_s0 = grad_s2  
    grad_s1 = grad_s2  
    grad_w1 = grad_s1 * x1  
    grad_x1 = grad_s1 * w1  
    grad_w0 = grad_s0 * x0  
    grad_x0 = grad_s0 * w0
```

Backprop can be implemented with a modular API, as a set of paired forward/backward functions (We will do this on A3!)

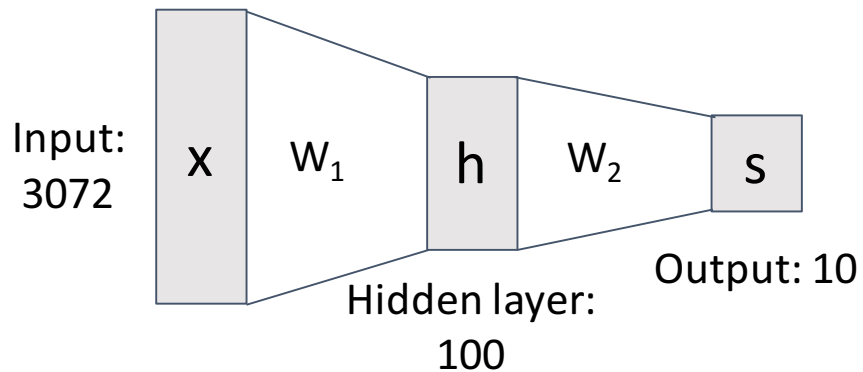
```
class Multiply(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x, y):  
        ctx.save_for_backward(x, y)  
        z = x * y  
        return z  
    @staticmethod  
    def backward(ctx, grad_z):  
        x, y = ctx.saved_tensors  
        grad_x = y * grad_z # dz/dx * dL/dz  
        grad_y = x * grad_z # dz/dy * dL/dz  
        return grad_x, grad_y
```

$$f(x, W) = Wx$$

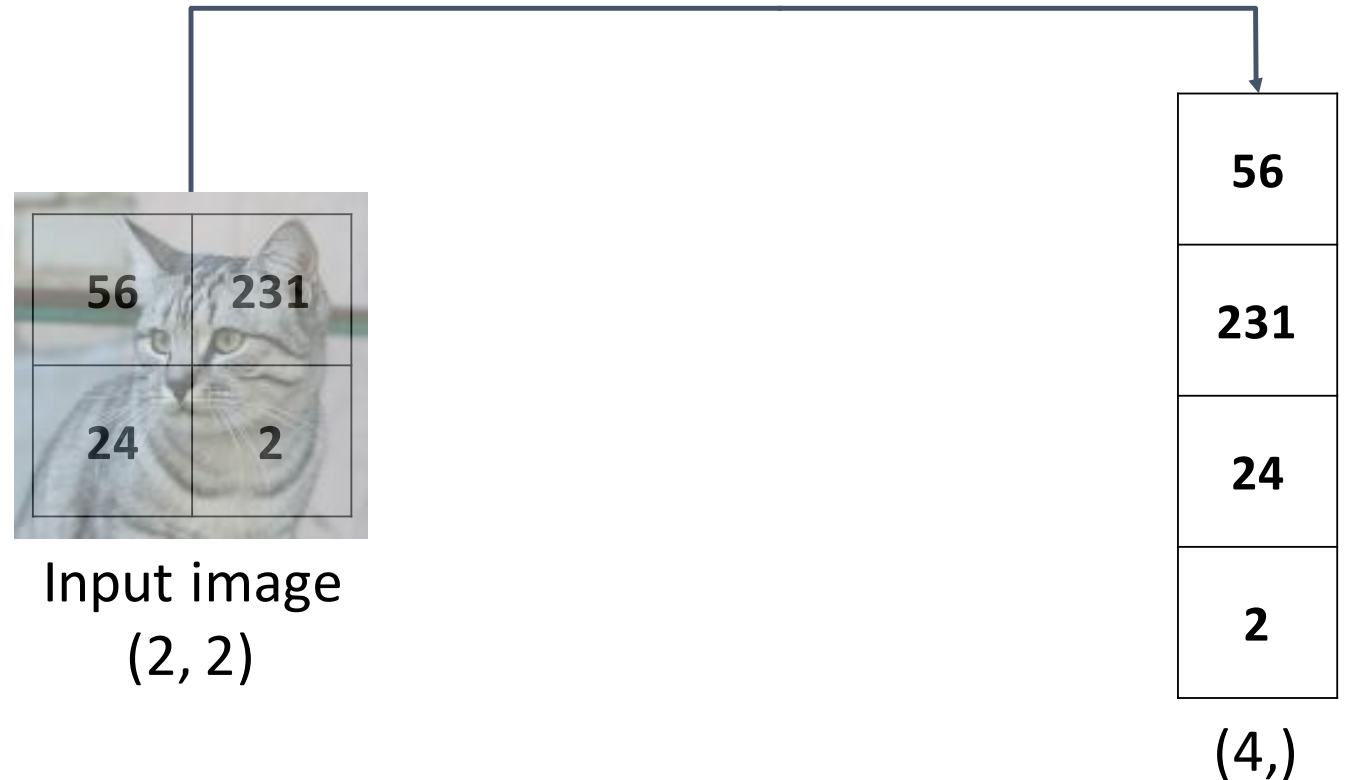


Problem: So far our classifiers don't respect the spatial structure of images!

$$f = W_2 \max(0, W_1 x)$$



Stretch pixels into column



Next time: Convolutional Neural Networks