# ROB 430/599: Deep Learning for Robot Perception and Manipulation (DeepRob)

Lecture 6: Backpropagation

01/28/2026

ROBOTICS

# Today

- Feedback and Recap (5min)
- Backpropagation
  - P1 - Linear Classifier gradients (25min)
  - Computational Graph: Examples of calculating backprop (30min)
  - Backprop with vectors (25min)
- Summary and Takeaways (5min)

# P1 Hints

- `.view` vs. `.reshape`
  - `.view` more memory efficient, but only works with contiguous memory tensor. Preferred in our P1.
  - `.reshape` works with both contiguous and non-contiguous memory tensor. May return a view or a copy.
- `torch.chunk(tensor, NumChuncks, dim)`: split a tensor in chunks - useful in cross validation
- `Compute_distances_no_loops` - good place to debug RAM issue (see next two slides)
  - Hint: Euclidean distance $dist = \sqrt{(p-q)^2} = p^2 + q^2 - 2p \cdot q$

# P1 - Debugging RAM issue

This is likely due to your implementation of compute_distances_no_loops, so start debugging from there.

Debugging tips:

- **Don't save intermediate tensor operations** -- this will allocate additional memory for those variables
  - Replace those variables with the operation itself in your implementation
  - Simple example of what this might look like:

```
# Additional memory allocation for variable 'mult'
mult = torch.mm(x1, x2)
return mult

# Replace with single line
return torch.mm(x1, x2)
```

run code snippet Visit Manage Class to disable runnable code snippets

- Check CPU allocation for tensors: tensor_name.element_size() * tensor_name.nelement()
- Check GPU allocation for tensors: torch.cuda.memory_allocated()
- Run the Pytorch profiler for your function: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html
  - Replace "model_inference" with "compute_distances_no_loops" to check usage for your function
- Kernel management:
  - Run only certain cells in the notebook (see @46)
  - Re-run cells that include variables and functions you are debugging and restart kernel to reset
    - Colab will save all of your variables in a single runtime, so they will retain values unless reassigned and still have memory allocated

ROBOTICS

# P1 - Debugging RAM issue

I solved it by deleting all runtime (Runtime --> Disconnect and delete runtime) then running up until and including the "Setup" cell with all the needed imports. Then skipping all other cells and running the entire "Cross-Validation Section." This fixed my RAM issue and I've been able to run my last cell without a problem (tried it on a few different runtimes).

(previous solution/discussion from Piazza)

Also, upgrade to Google Colab Pro, select “High-RAM runtime”

# torch.Tensor Practice Resources

torch.Tensor
https://docs.pytorch.org/docs/stable/tensors.html

Broadcasting:
https://docs.pytorch.org/docs/stable/notes/broadcasting.html

https://mmuratarat.github.io/2019-01-27/derivation-of-softmax-function

# P1 Hints

Hand-derivation video: https://youtu.be/qW2awlotMzo

- Deriving Derivatives `dW` for Linear Classifiers

  - We have some dataset of (x, y)
  - We have a **score function:**
  - We have a **loss function:**

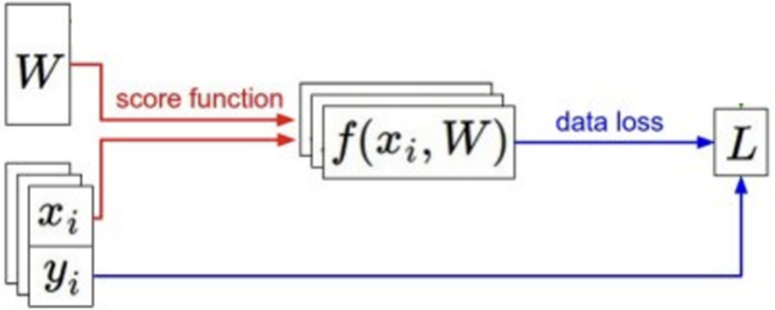
$$s = f(x; W, b) = Wx + b$$

Linear classifier

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda \sum_i w_i^2$$

Softmax: $L_i = -\log\left(\frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}\right)$

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

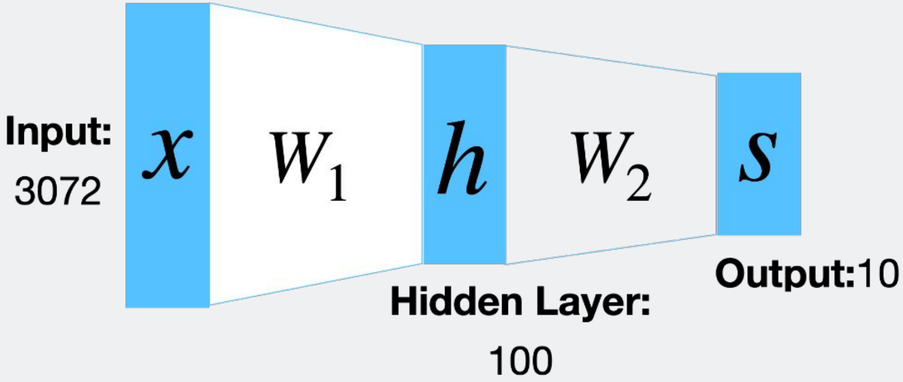


# Recap

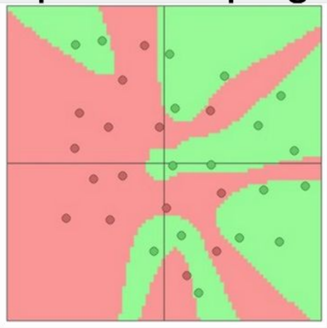
**P1 Deadline: Feb. 1, 2026**

From linear classifiers to fully-connected networks

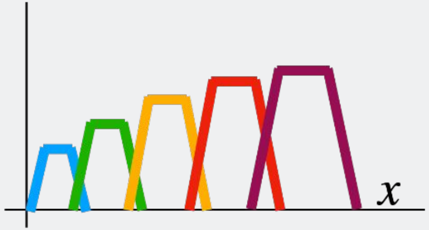
$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



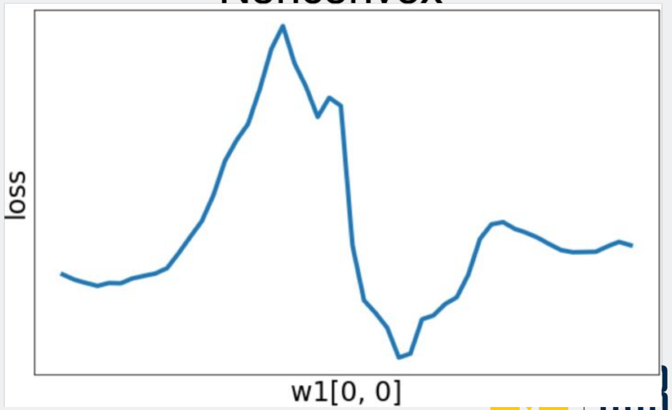

Space Warping



Universal approximation




Nonconvex



# How to Compute Gradients?

ReLU activation

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

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

Hinge loss

$$R(W) = \sum_k W_k^2$$ L2 regularization

Data loss

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

Regularization term

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

# Bad Idea: Derive $\nabla_W L$ on paper

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

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

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

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

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

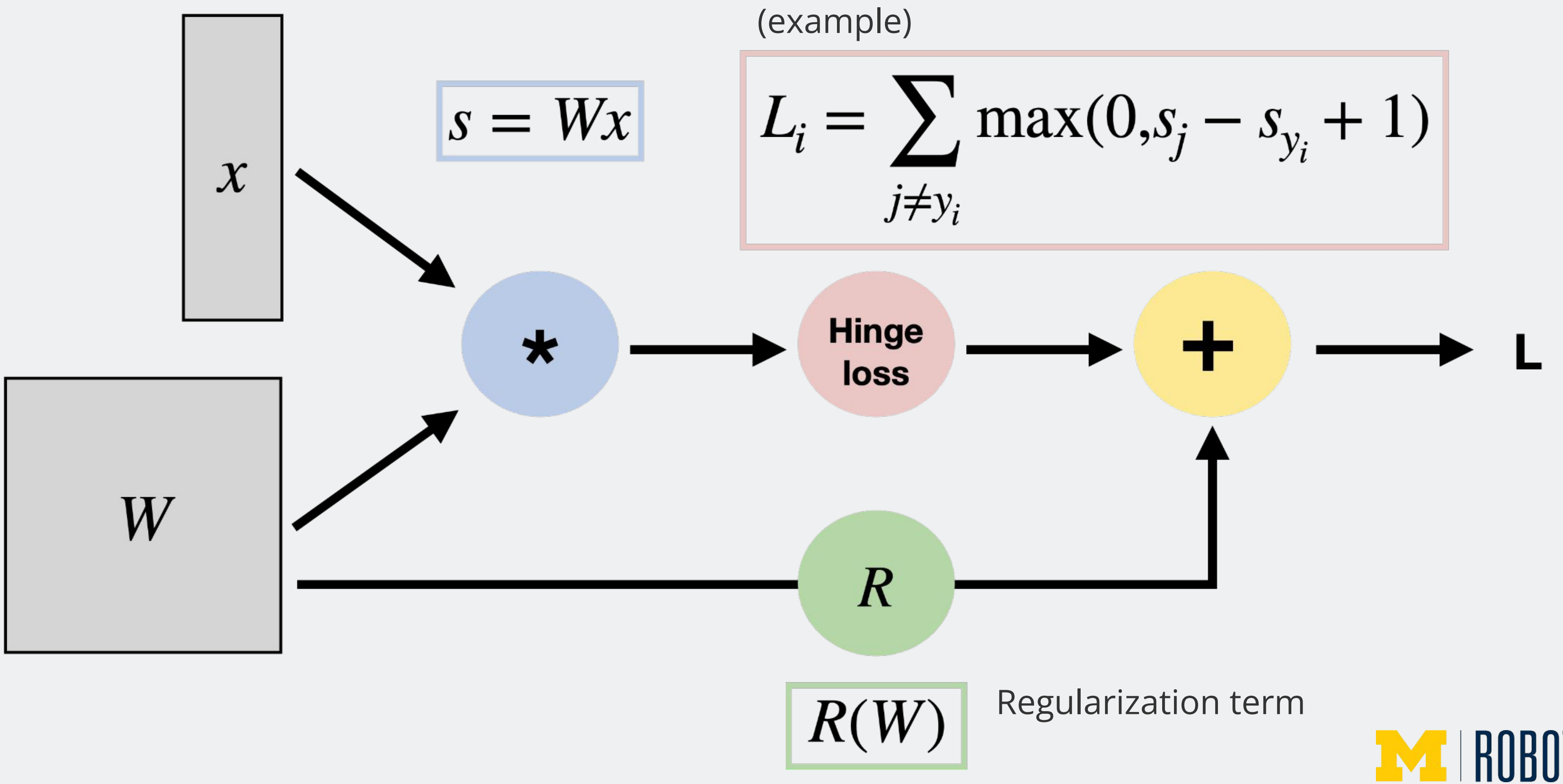
$$\nabla_W L = \nabla_W \left( \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x - W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

**Problem**: Very tedious with lots of matrix calculus

**Problem**: What if we want to change the 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



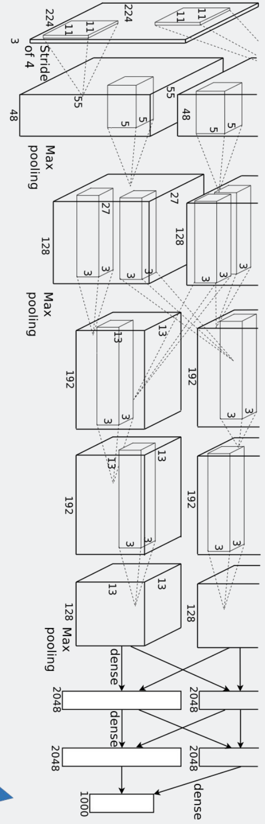

ROBOTICS

# Deep Network (AlexNet)

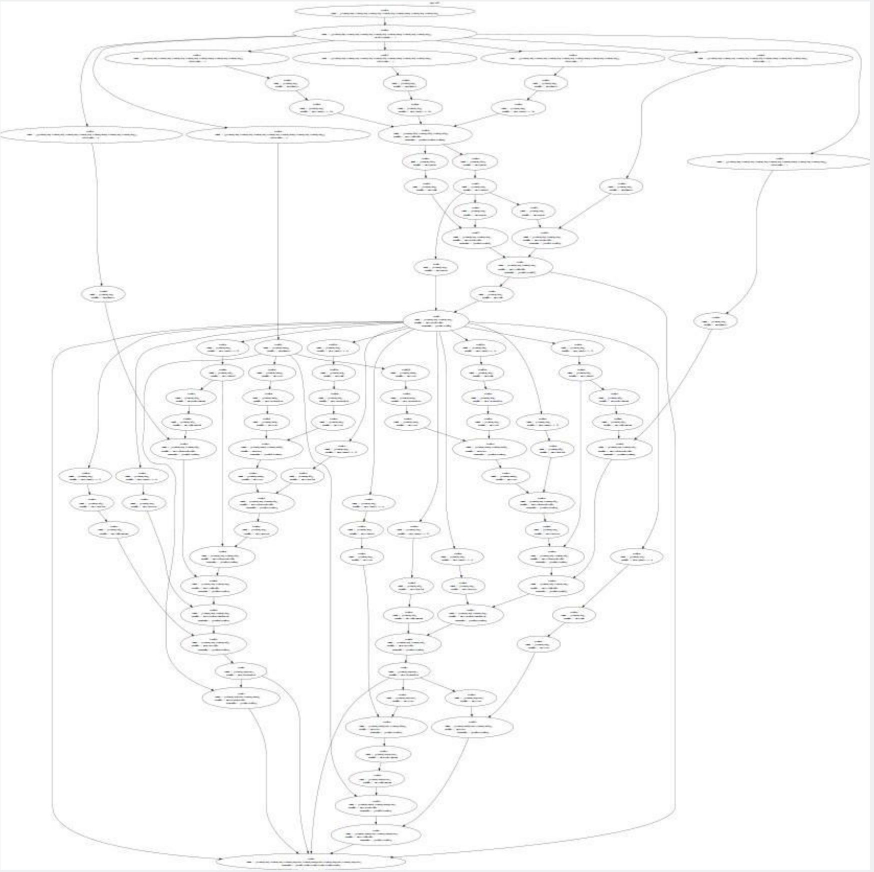
Input image

Weights

Loss



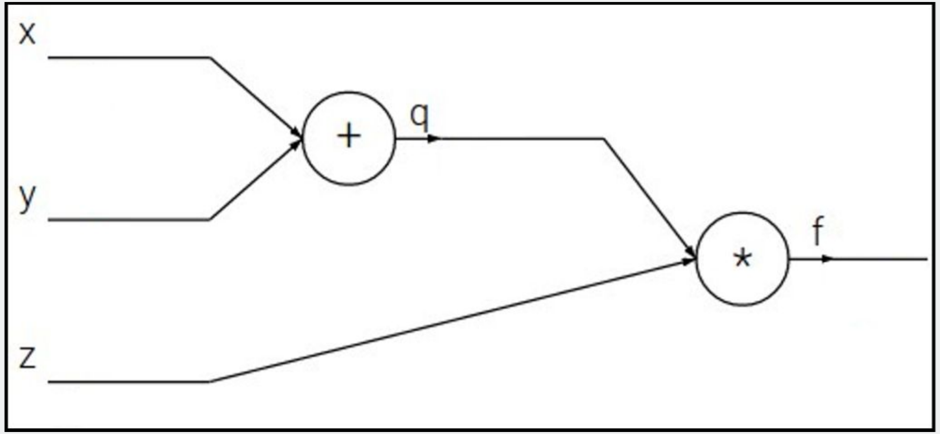
# Deep Network (Neural Turing Machine)



https://arxiv.org/abs/1410.5401
andrej karpathy (graph)

# Backpropagation: A simple example

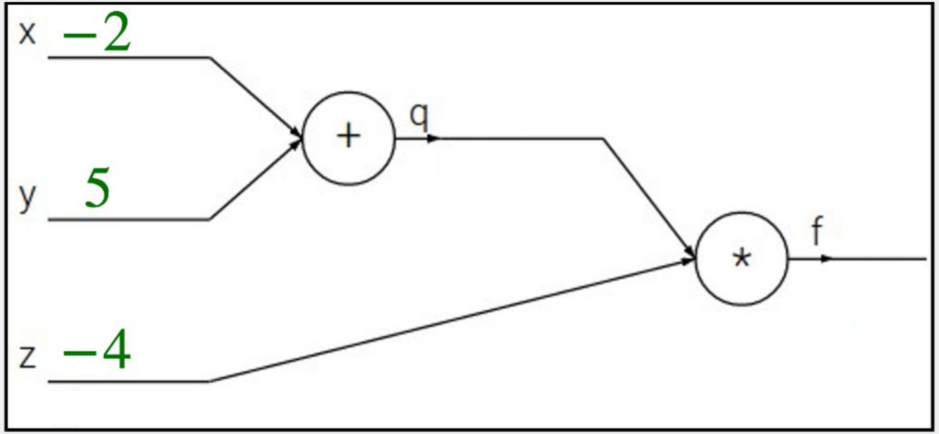
$$f(x, y, z) = (x + y) \cdot z$$



# Backpropagation: A simple example

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

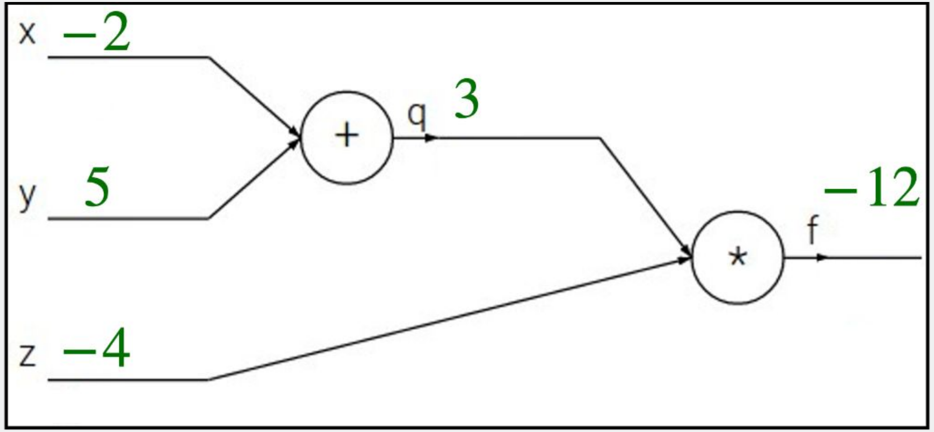
e.g. $x = -2, y = 5, z = -4$



# Backpropagation: A simple example

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

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



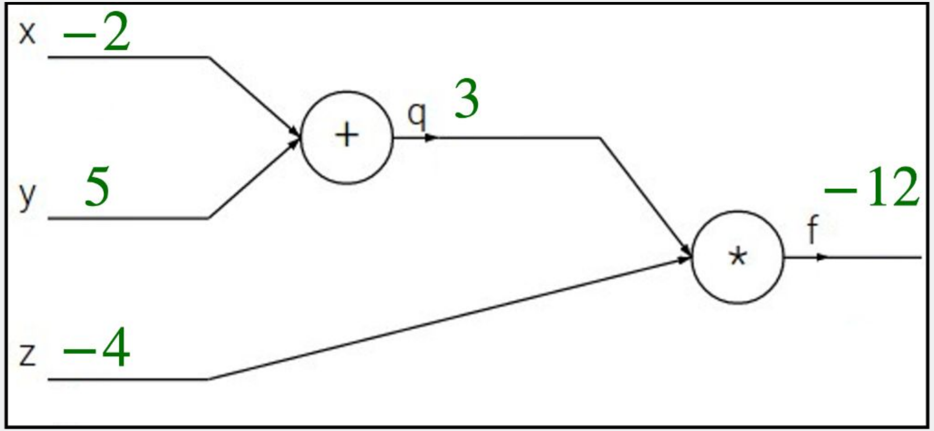

**1. Forward pass**: Compute outputs

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

# Backpropagation: A simple example

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

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




1. **Forward pass**: Compute outputs

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

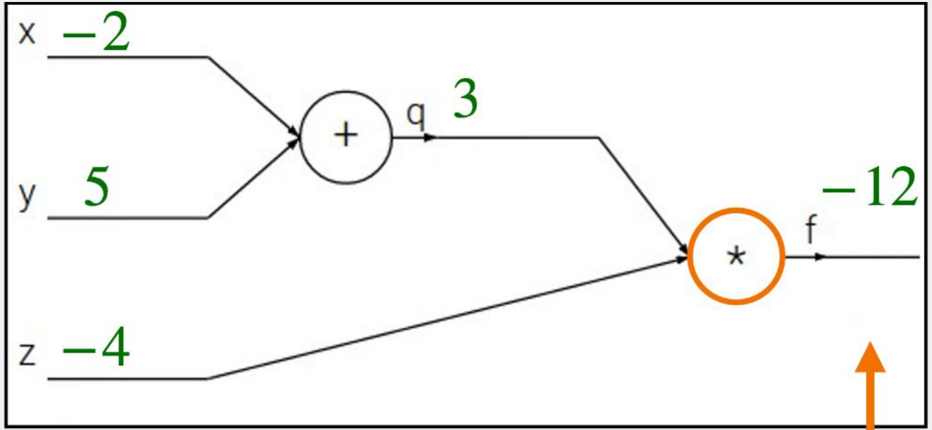
2. **Backward pass**: Compute derivatives

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

# Backpropagation: A simple example

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

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




1. **Forward pass**: Compute outputs

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

2. **Backward pass**: Compute derivatives

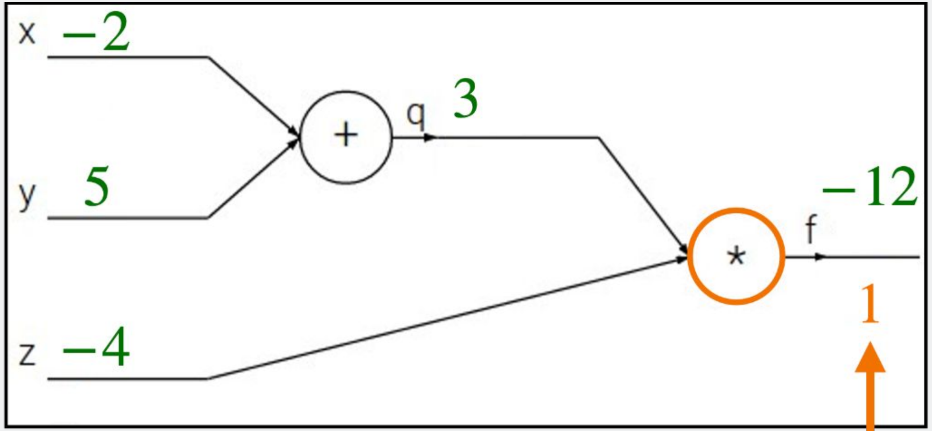
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

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

# Backpropagation: A simple example

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

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




**1. Forward pass**: Compute outputs

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

**2. Backward pass**: Compute derivatives

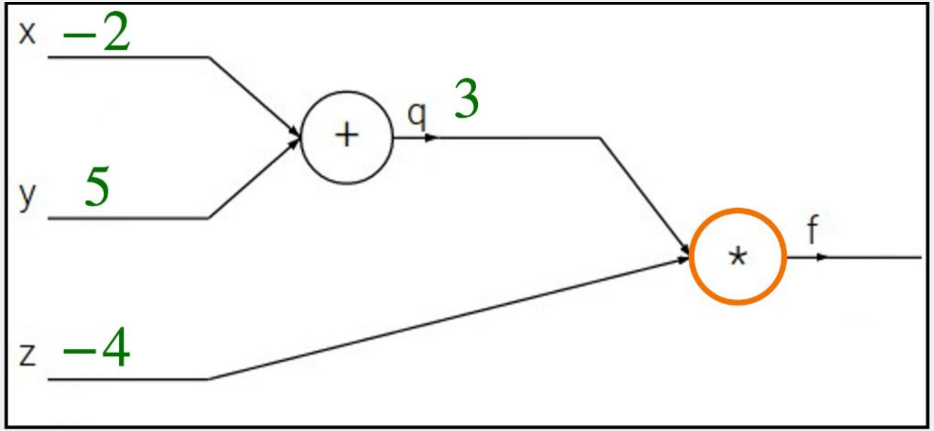
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

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

# Backpropagation: A simple example

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

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




**1. Forward pass**: Compute outputs

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

**2. Backward pass**: Compute derivatives

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

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

ROBOTICS

# Backpropagation: A simple example

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

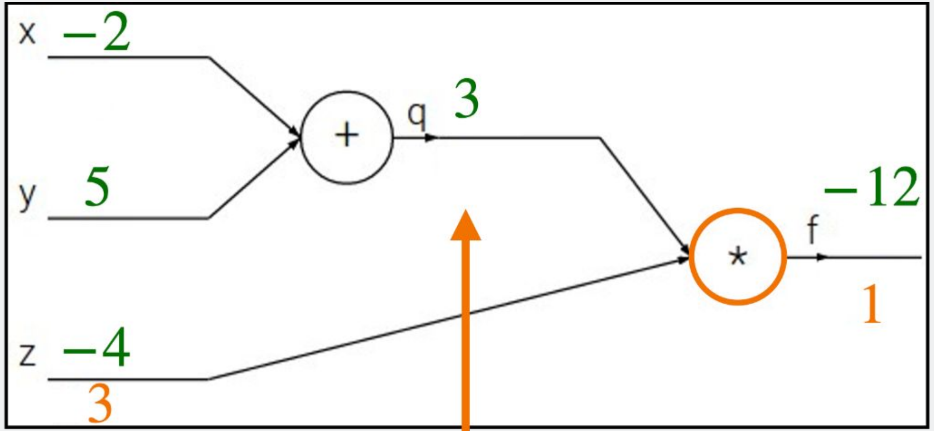
e.g. $x = -2, y = 5, z = -4$

**1. Forward pass**: Compute outputs

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

**2. Backward pass**: Compute derivatives

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




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

# Backpropagation: A simple example

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

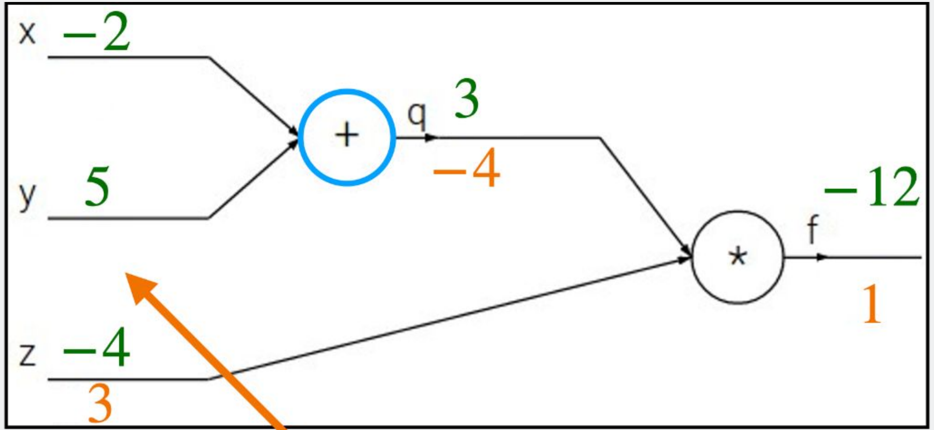
e.g. $x = -2, y = 5, z = -4$

**1. Forward pass**: Compute outputs

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

**2. Backward pass**: Compute derivatives

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



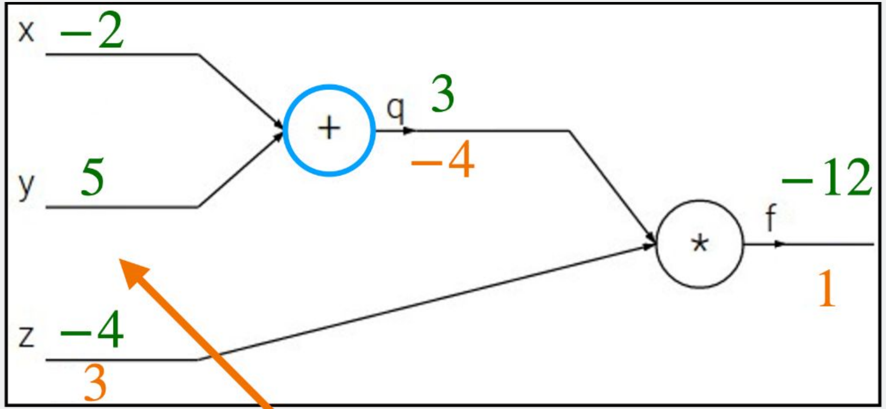

$\dfrac{\partial f}{\partial y}$ =???

ROBOTICS

# Backpropagation: A simple example

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

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




**1. Forward pass**: Compute outputs

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

**2. Backward pass**: Compute derivatives

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

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

Downstream Gradient

Local Gradient

Upstream Gradient

# Backpropagation: A simple example

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

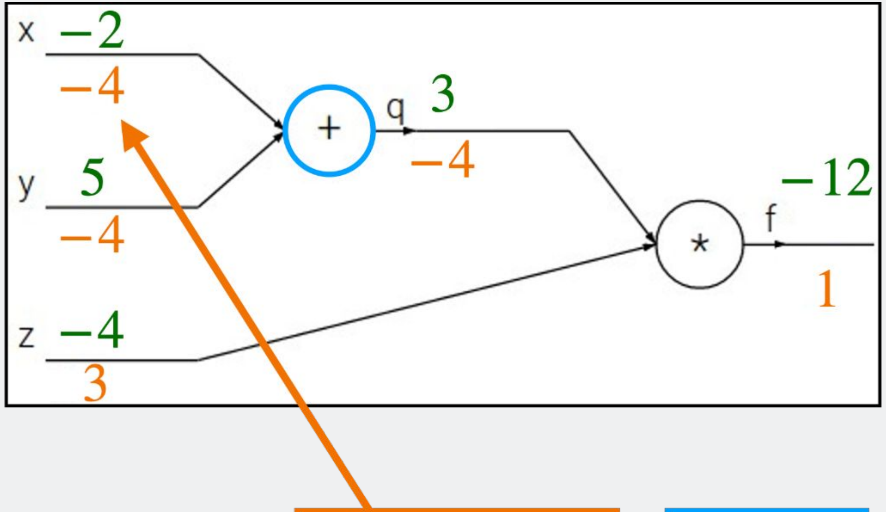
e.g. $x = -2, y = 5, z = -4$

**1. Forward pass**: Compute outputs

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

**2. Backward pass**: Compute derivatives

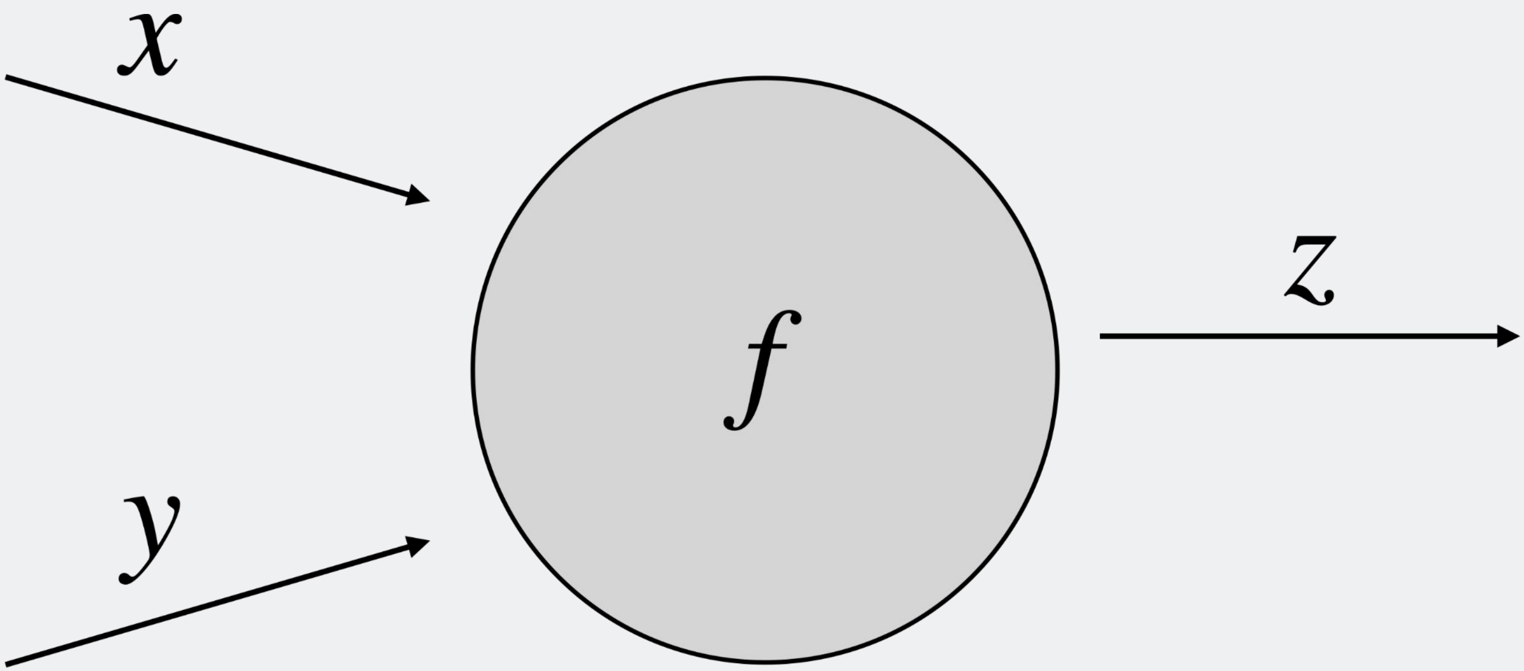
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



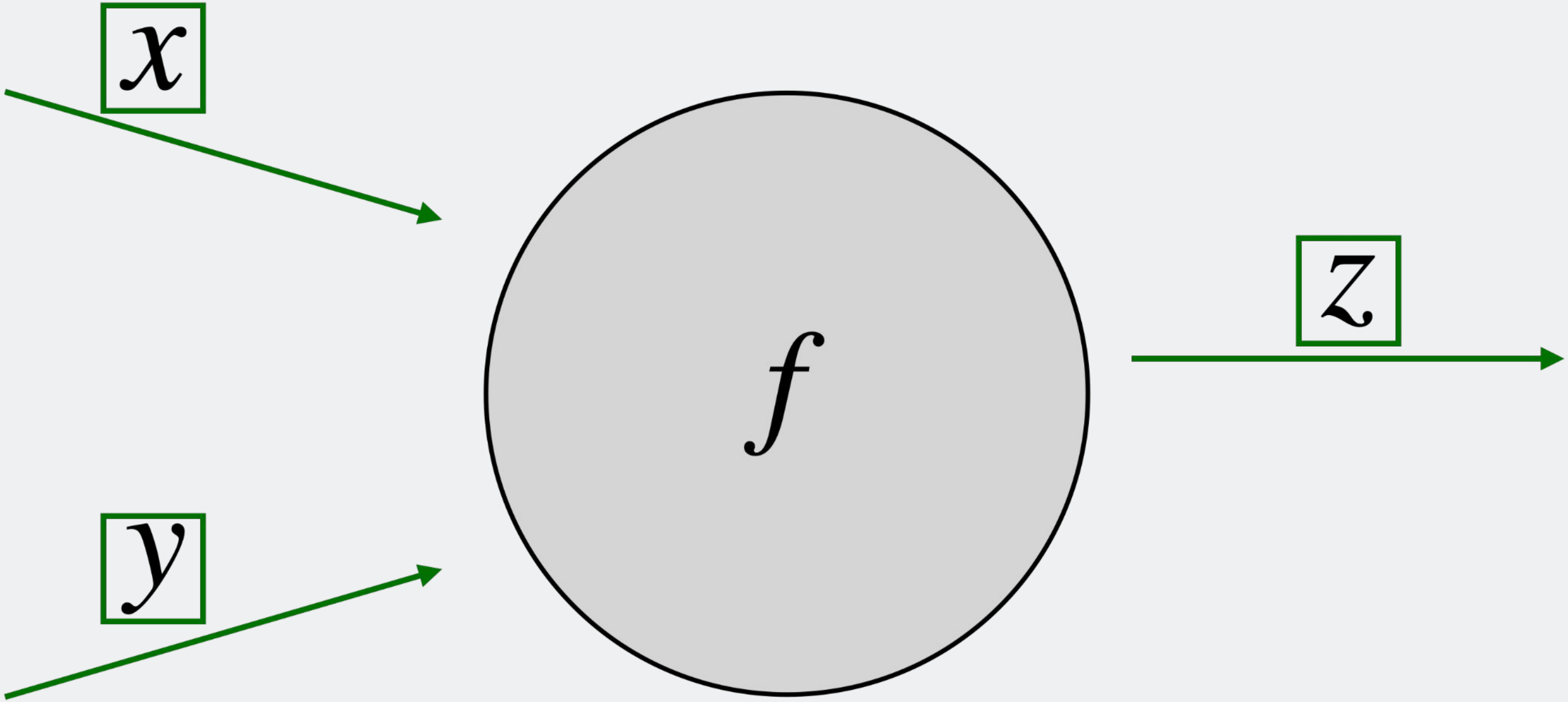

$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x} \frac{\partial f}{\partial q} \qquad \frac{\partial q}{\partial x} = 1$$

Downstream Gradient — Local Gradient — Upstream Gradient

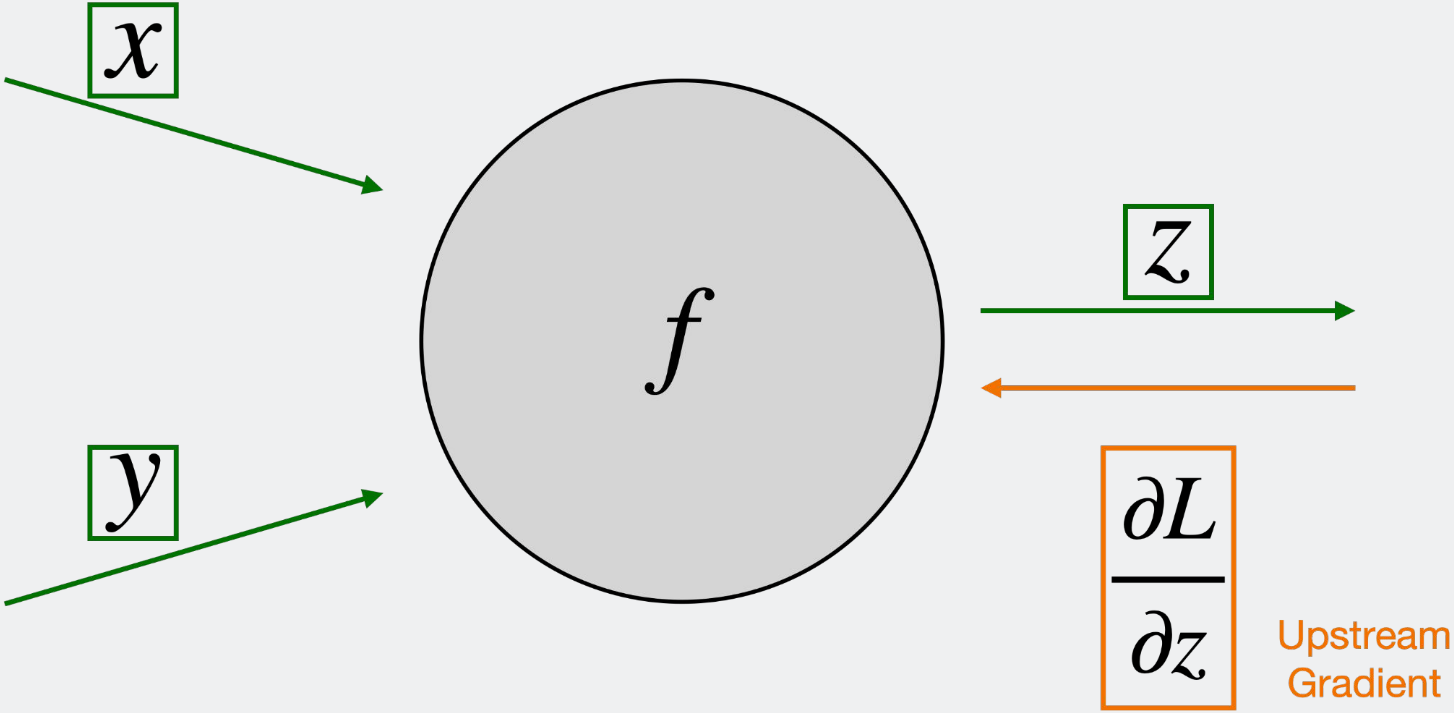
# Local Properties of Backpropagation



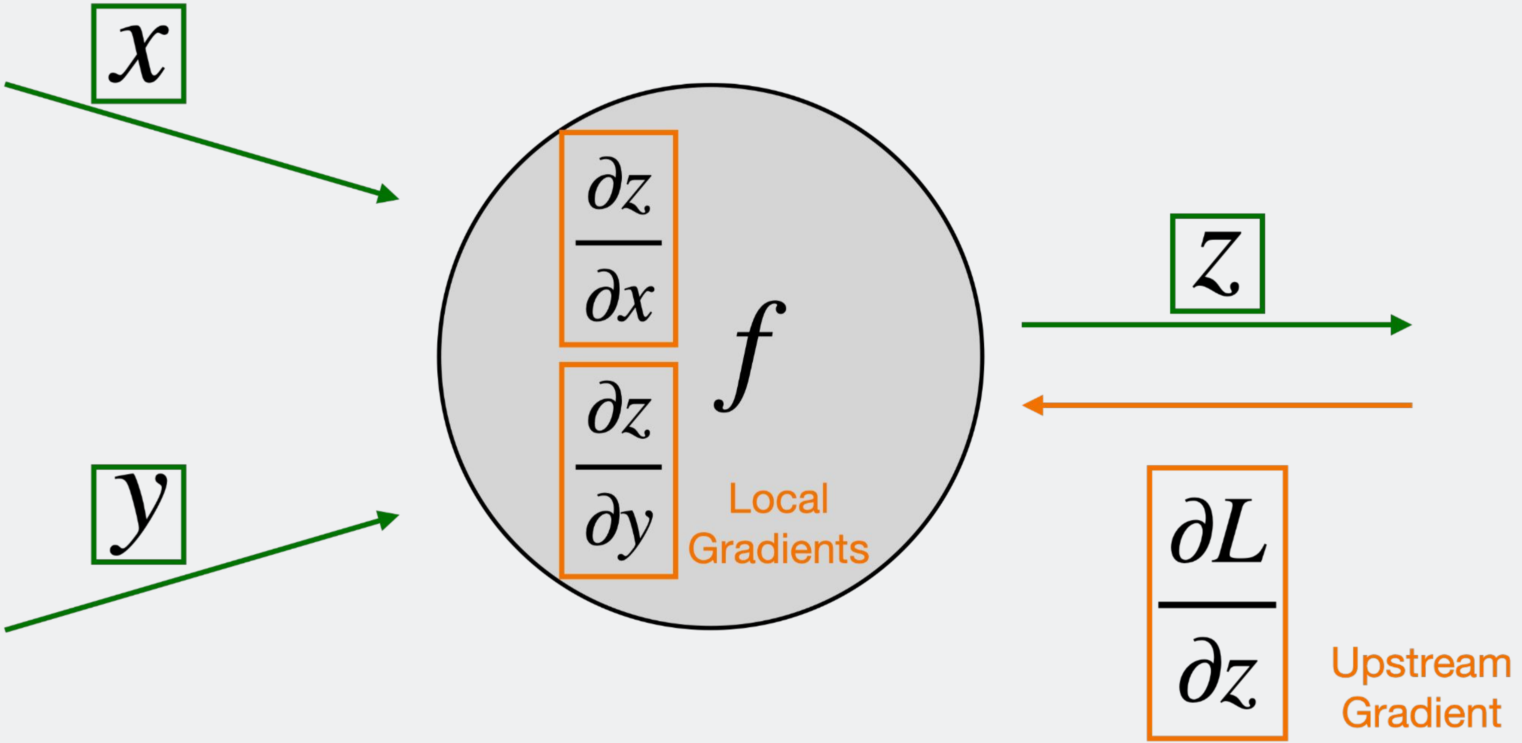
# Local Properties of Backpropagation



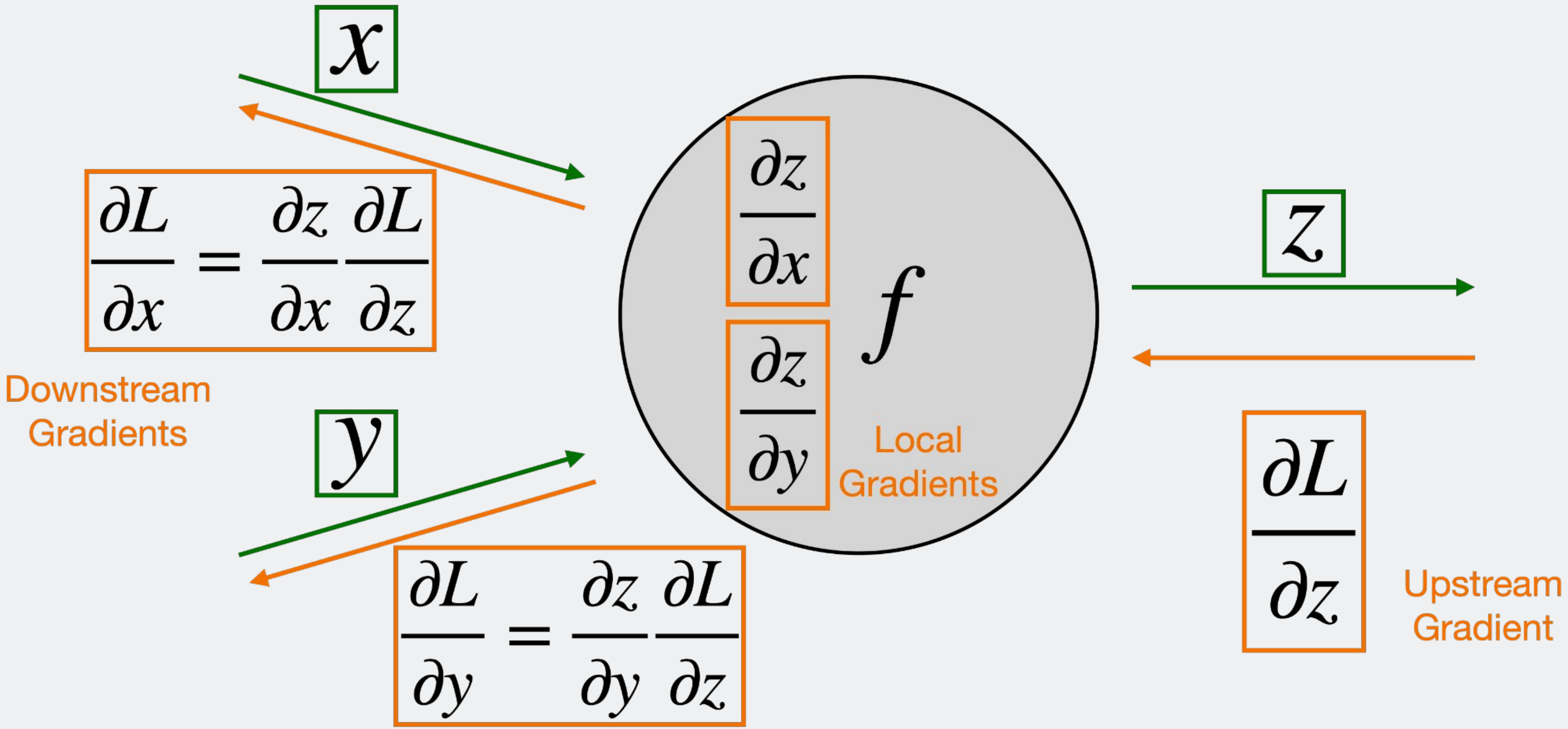
# Local Properties of Backpropagation



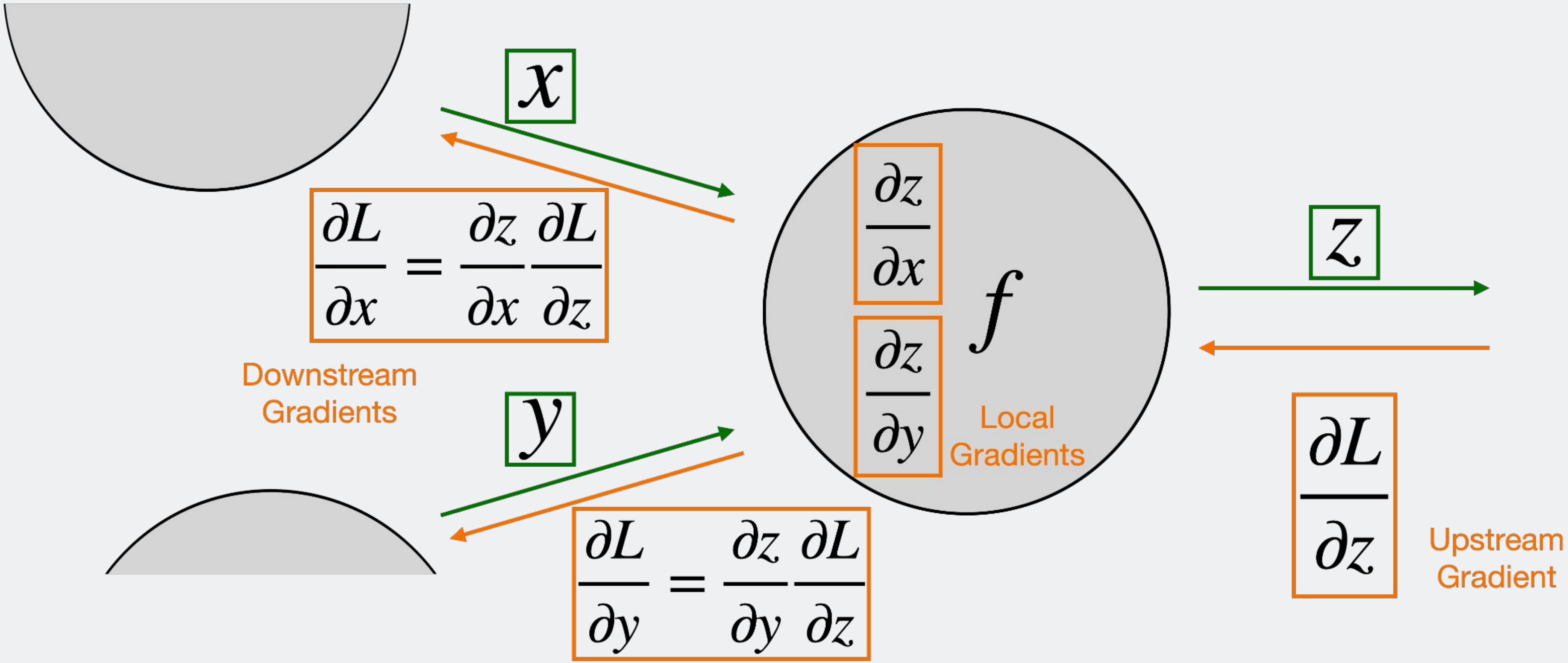
# Local Properties of Backpropagation



# Local Properties of Backpropagation



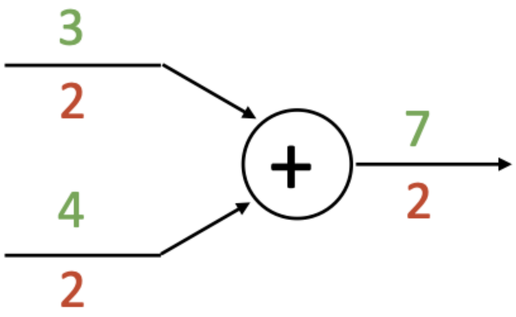
# Local Properties of Backpropagation



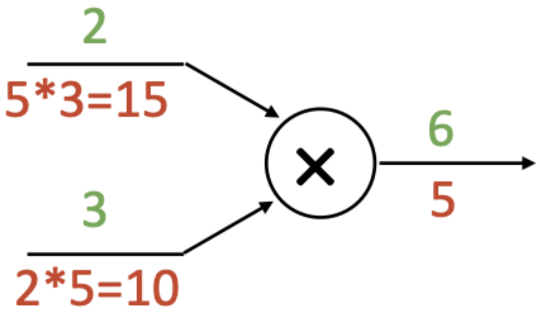
# Patterns in Gradient Flow

important!

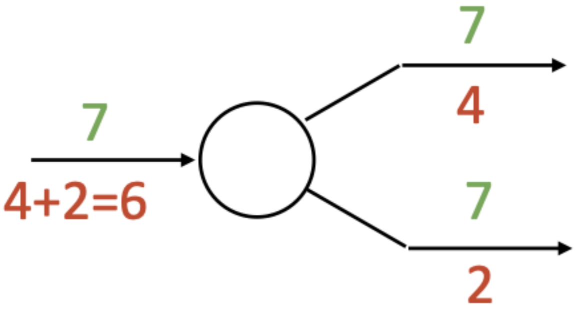
**add** gate: gradient distributor



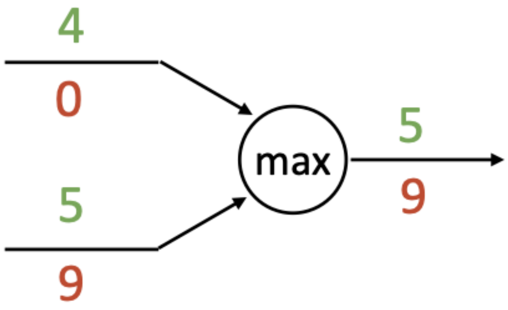

**mul** gate: "swap multiplier"



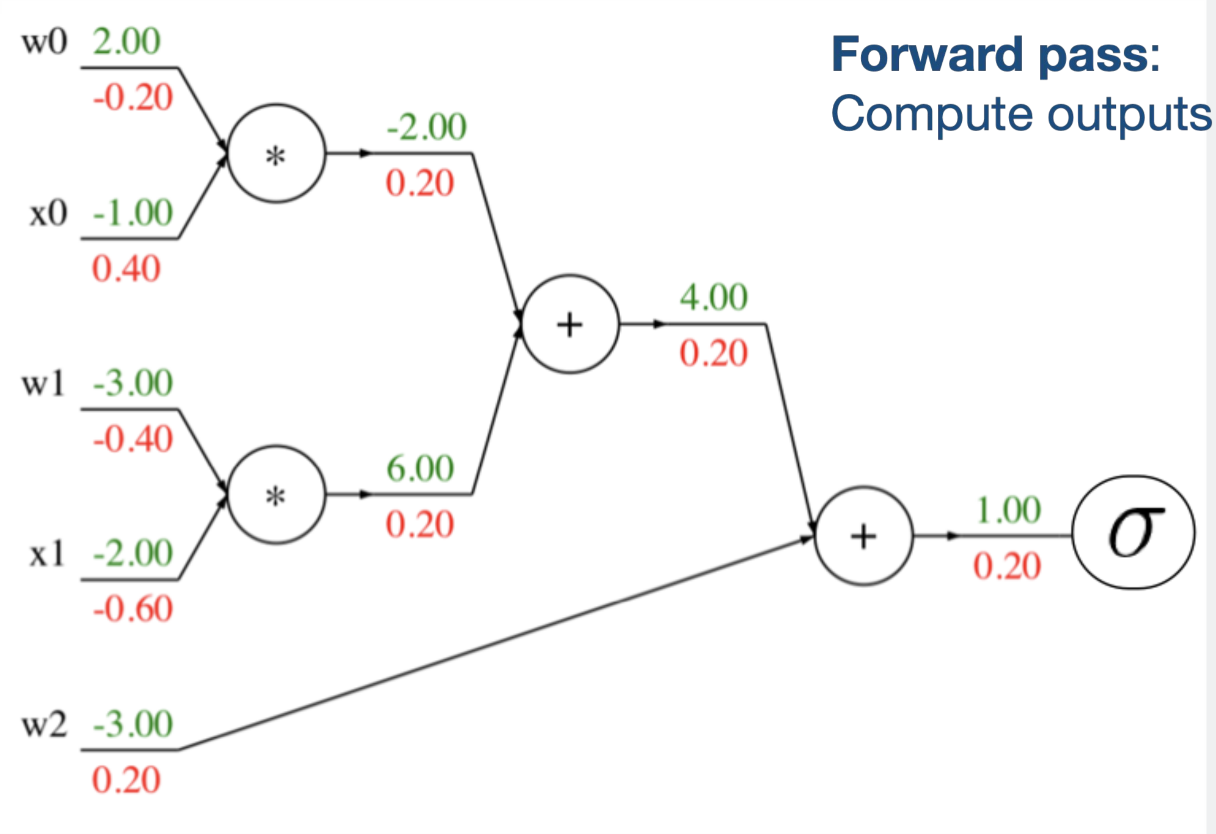

**copy** gate: gradient adder




**max** gate: gradient router



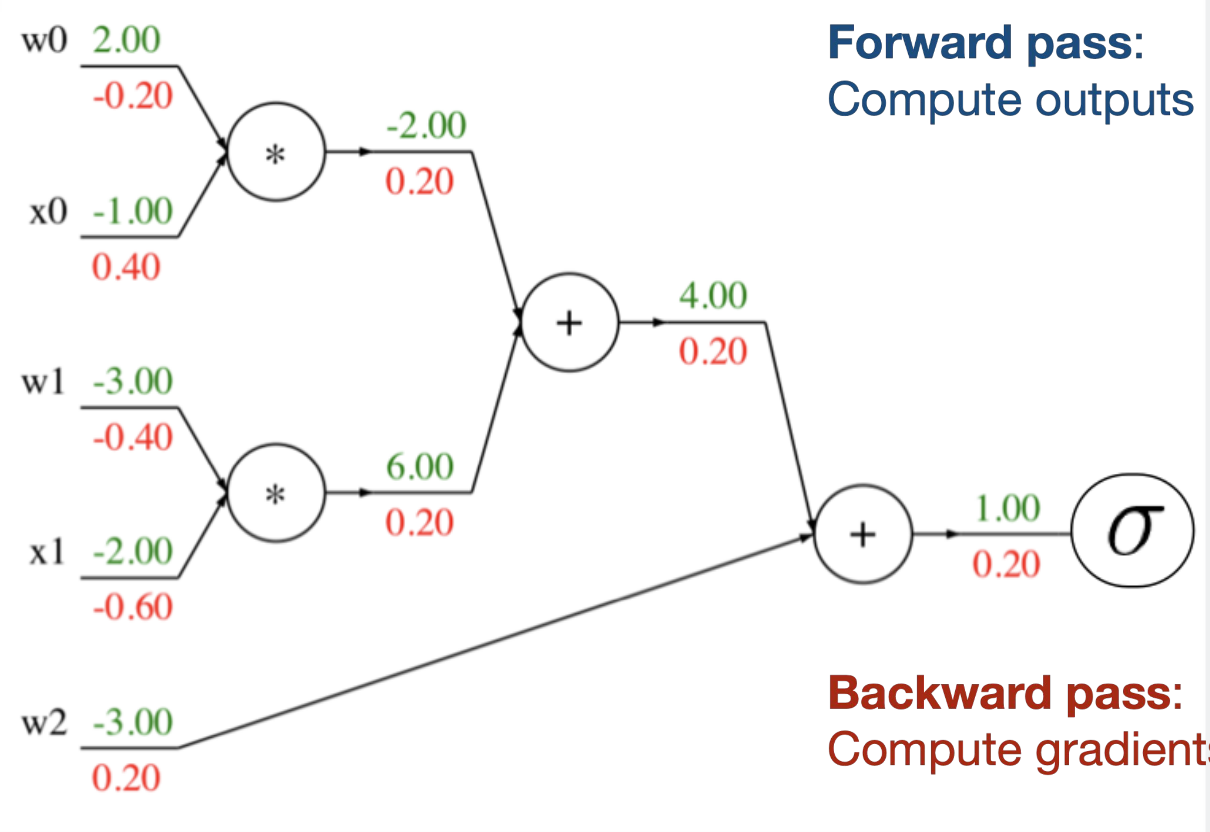
# Backprop Implementation: "Flat" gradient code




**Forward pass**:
Compute outputs

```
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 outputs

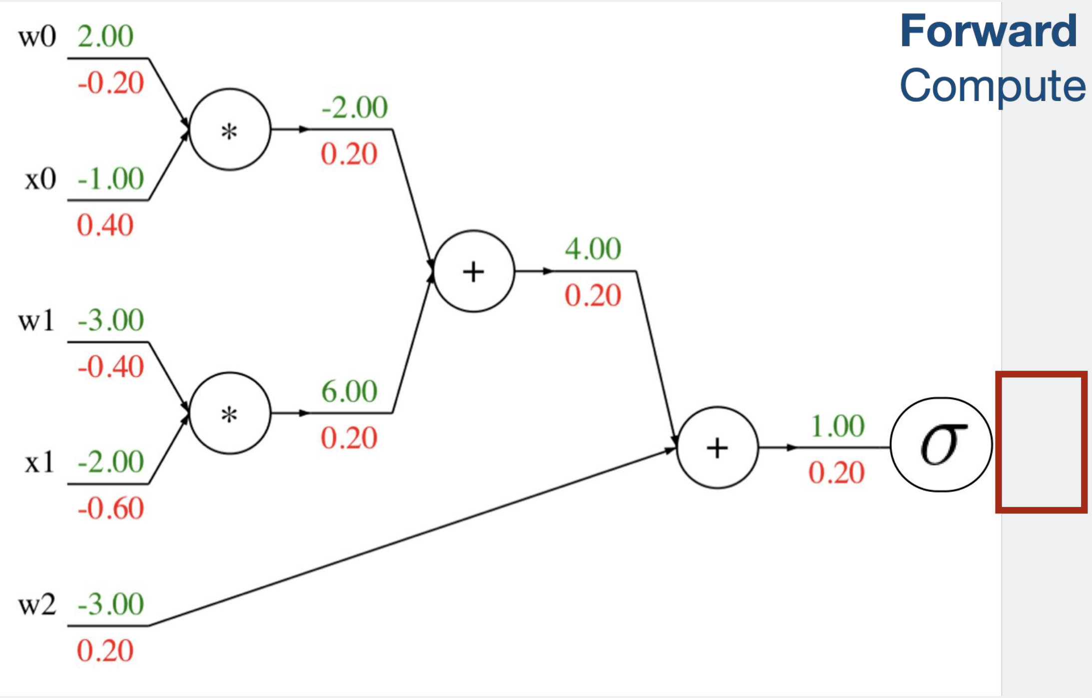
**Backward pass**: Compute gradients

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

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

```python
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 outputs

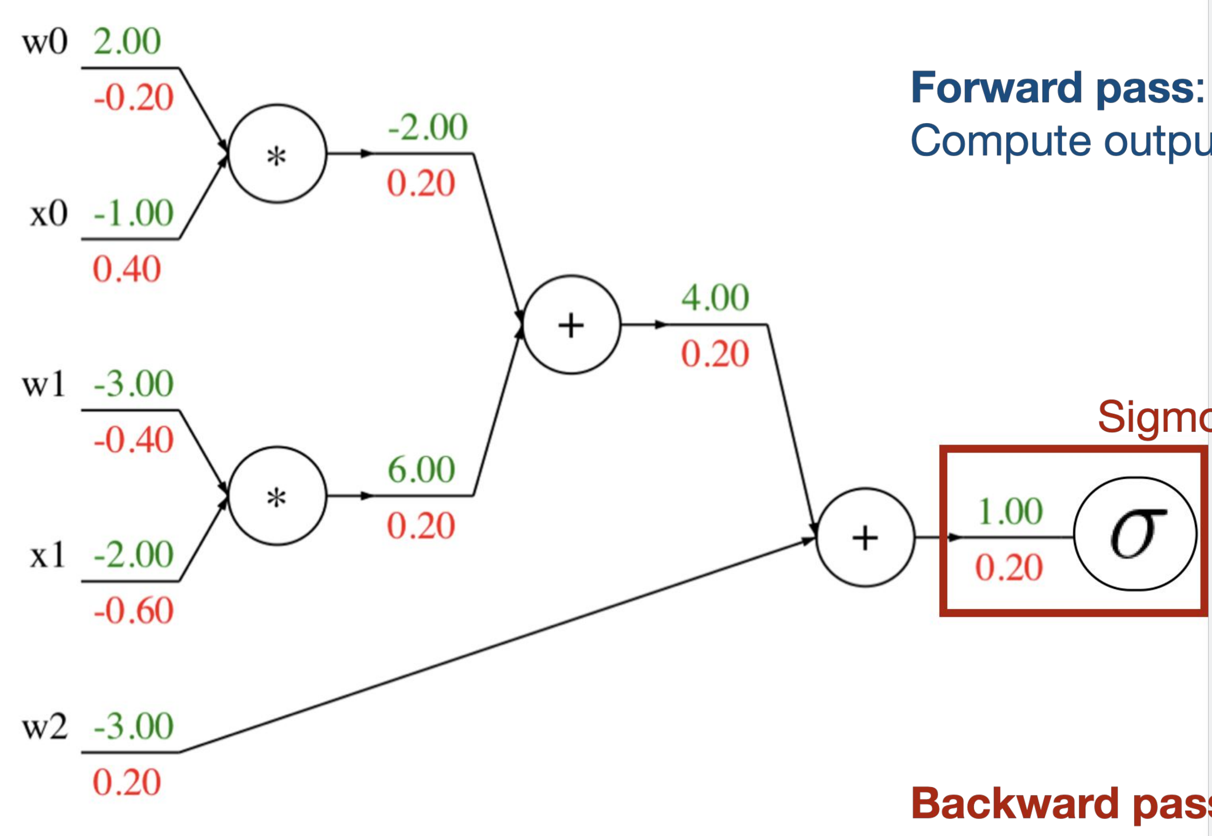
```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

Base case

```
  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
```

**Backward pass**:
Compute gradients

# Backprop Implementation: "Flat" gradient code




**Forward pass**:
Compute outputs

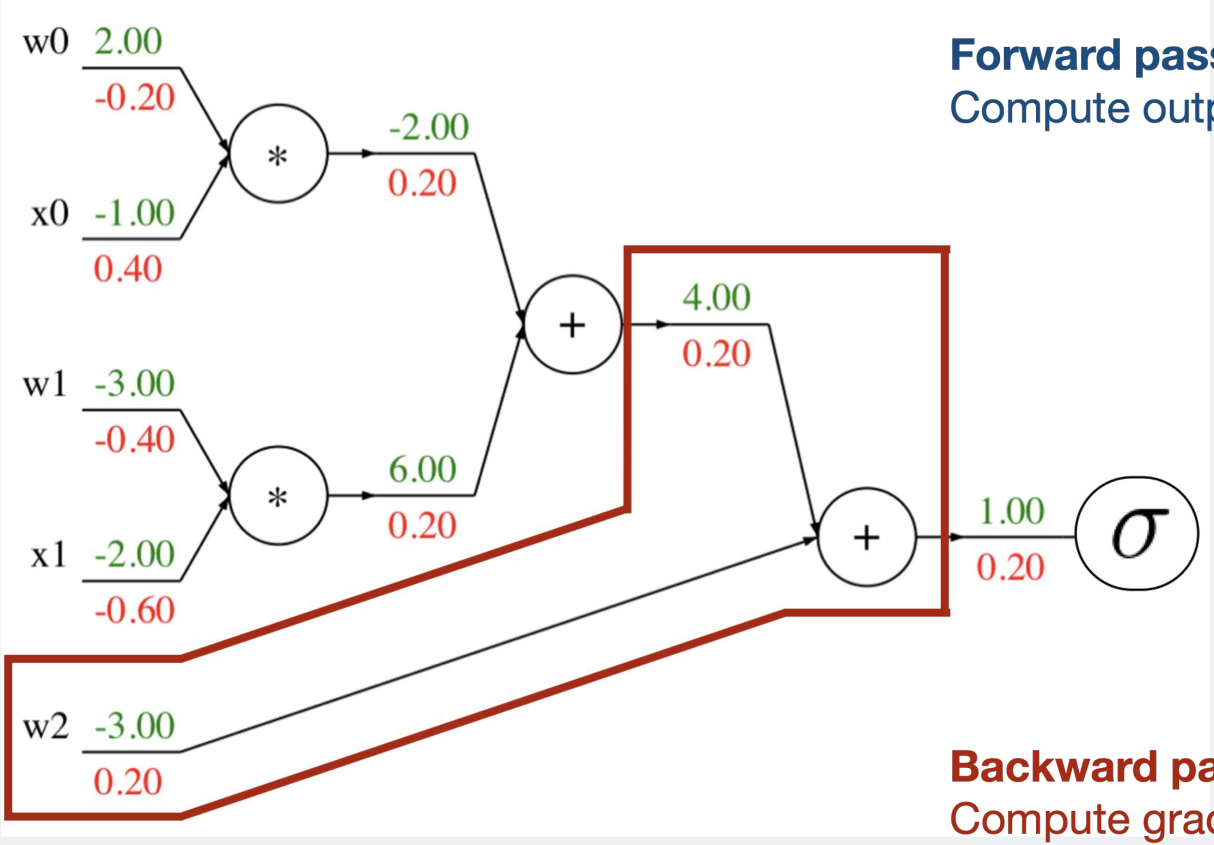
Sigmoid

```
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
```

**Backward pass**:
Compute gradients

# Backprop Implementation: "Flat" gradient code




**Forward pass**:
Compute outputs

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

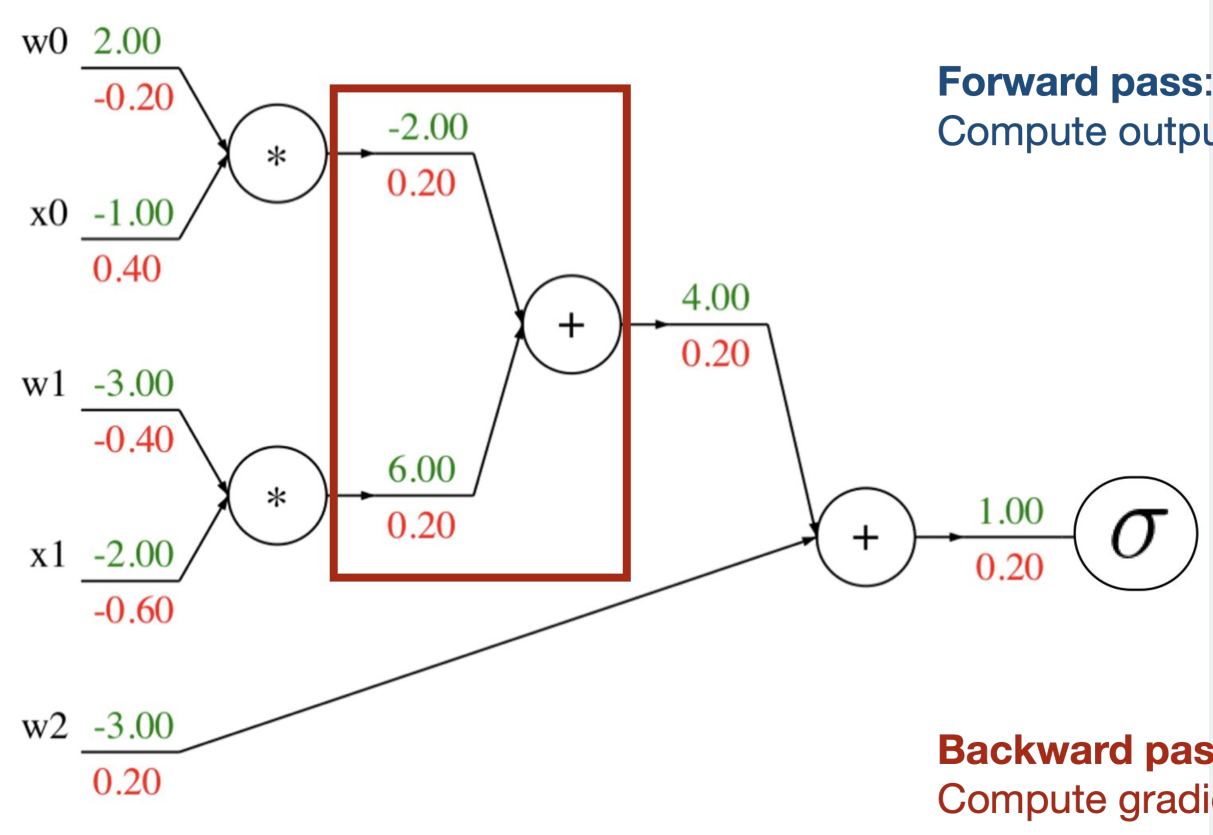
Add

```
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
```

**Backward pass**:
Compute gradients

ROBOTICS

# Backprop Implementation: "Flat" gradient code




**Forward pass:**
Compute outputs

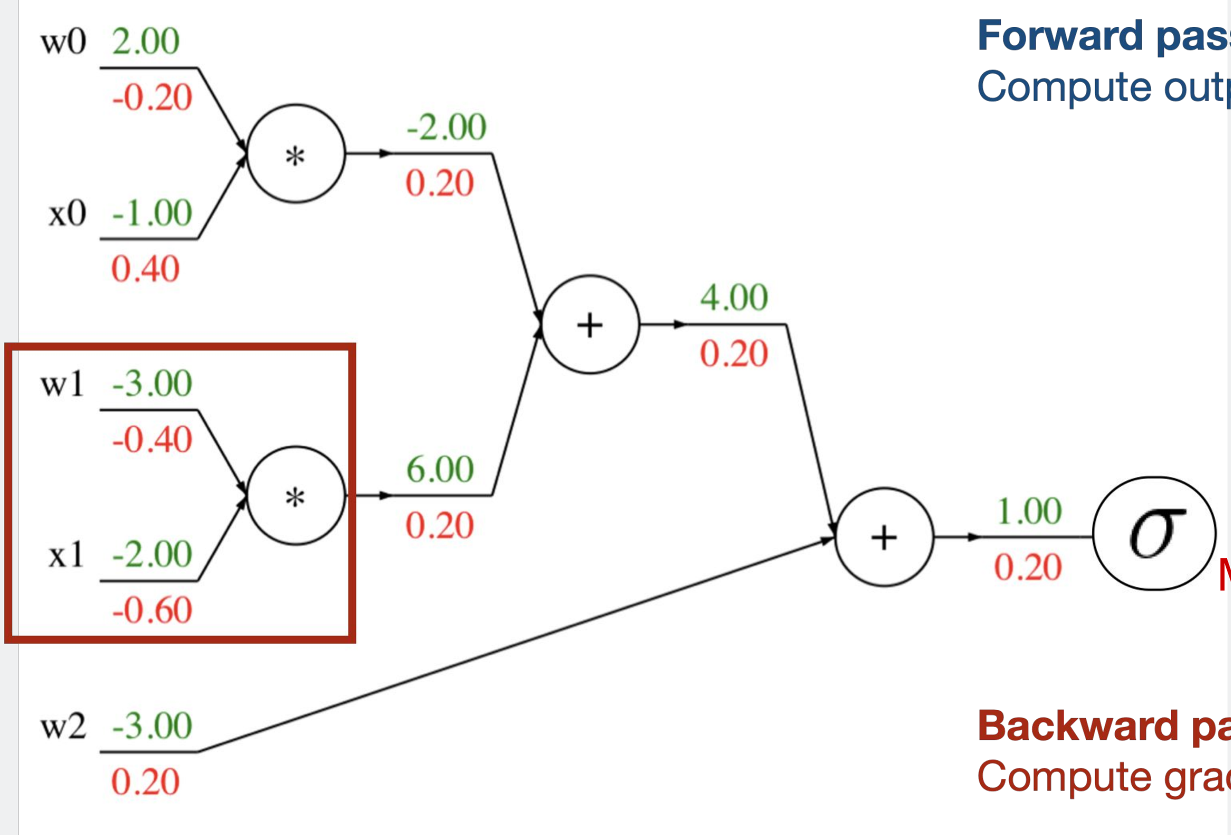
```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

```
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

**Backward pass:**
Compute gradients

# Backprop Implementation: "Flat" gradient code




**Forward pass**: Compute outputs

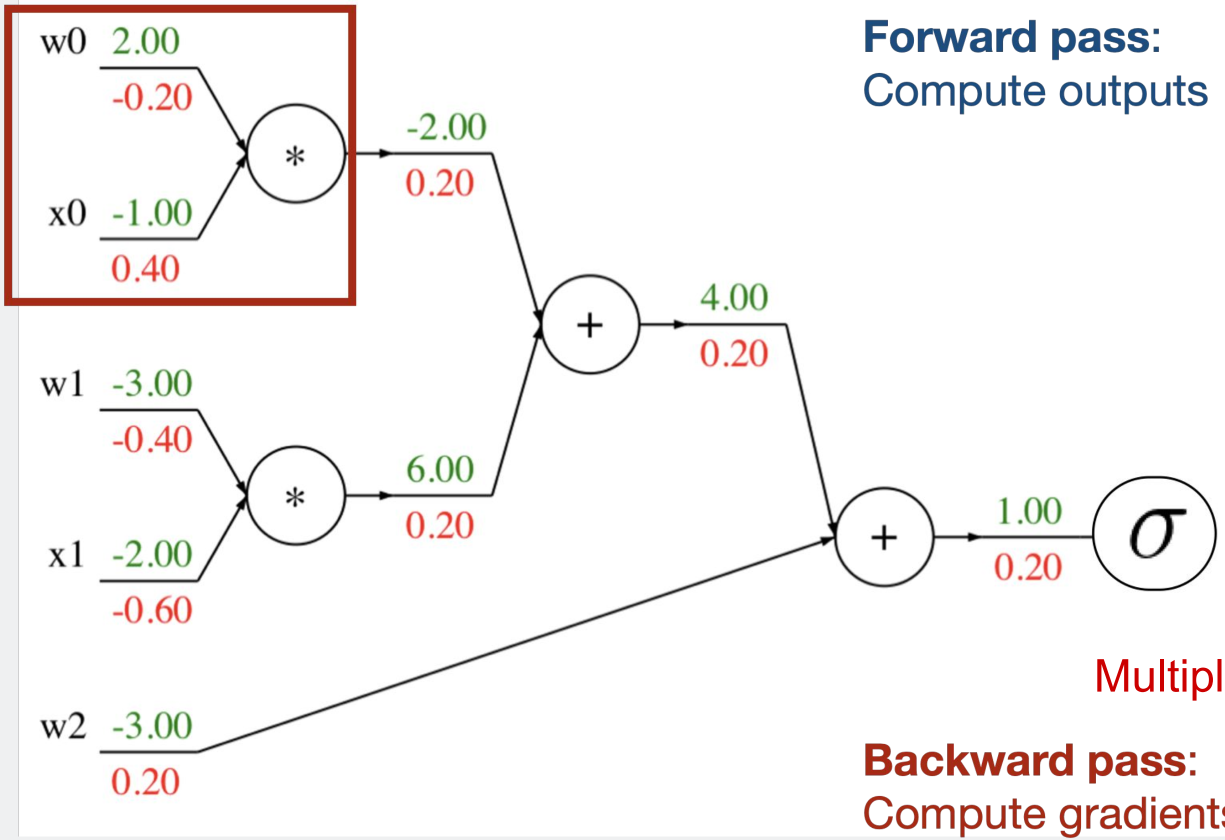
```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

Multiply

```
    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
```

**Backward pass**: Compute gradients

# Backprop Implementation: “Flat” gradient code




**Forward pass**: Compute outputs

```python
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

Multiply

```python
    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
```

**Backward pass**: Compute gradients

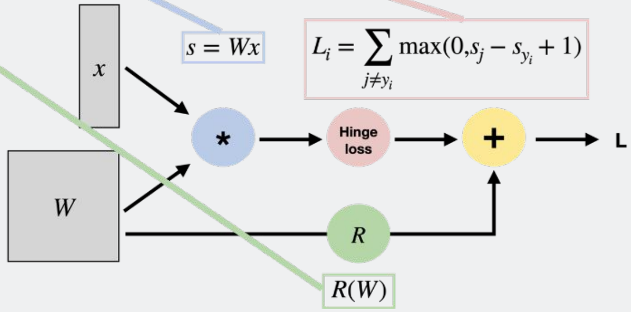
# “Flat” backprop: Do this for Project 2

**Forward pass:**
Compute outputs

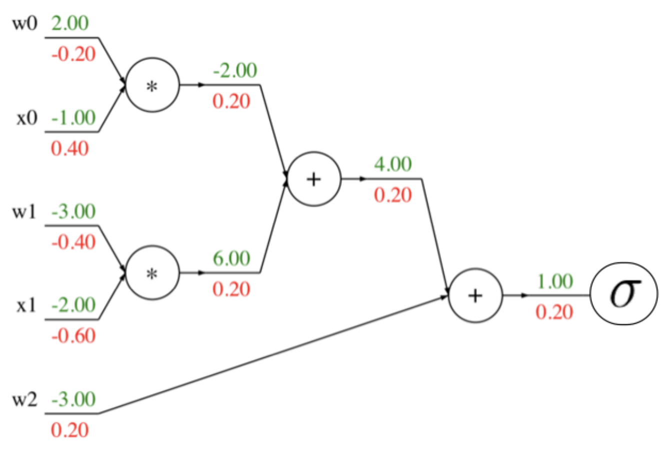
```
#########################################################################
# TODO:                                                                 #
# Implement a vectorized version of the structured SVM loss, storing the #
# result in loss.                                                       #
#########################################################################
# Replace "pass" statement with your code
num_classes = W.shape[1]
num_train = X.shape[0]
score = # ...
correct_class_score = # ...
margin = # ...
data_loss = # ...
reg_loss = # ...
loss += data_loss + reg_loss
#########################################################################
#                            END OF YOUR CODE                           #
#########################################################################
```

**Backward pass:**
Compute gradients

```
#########################################################################
# TODO:                                                                 #
# Implement a vectorized version of the gradient for the structured SVM #
# loss, storing the result in dW.                                       #
#                                                                       #
# Hint: Instead of computing the gradient from scratch, it may be easier #
# to reuse some of the intermediate values that you used to compute the #
# loss.                                                                 #
#########################################################################
# Replace "pass" statement with your code
dmargins = # ...
dscores = # ...
dW = # ...
#########################################################################
#                            END OF YOUR CODE                           #
#########################################################################
```



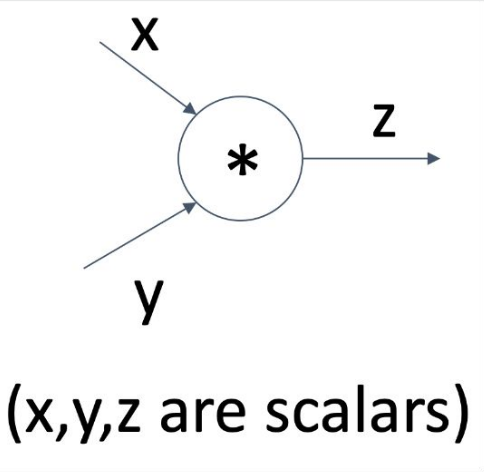
# Backprop Implementation: Modular API




## 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




```python
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

# So far: backprop w/ 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?

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

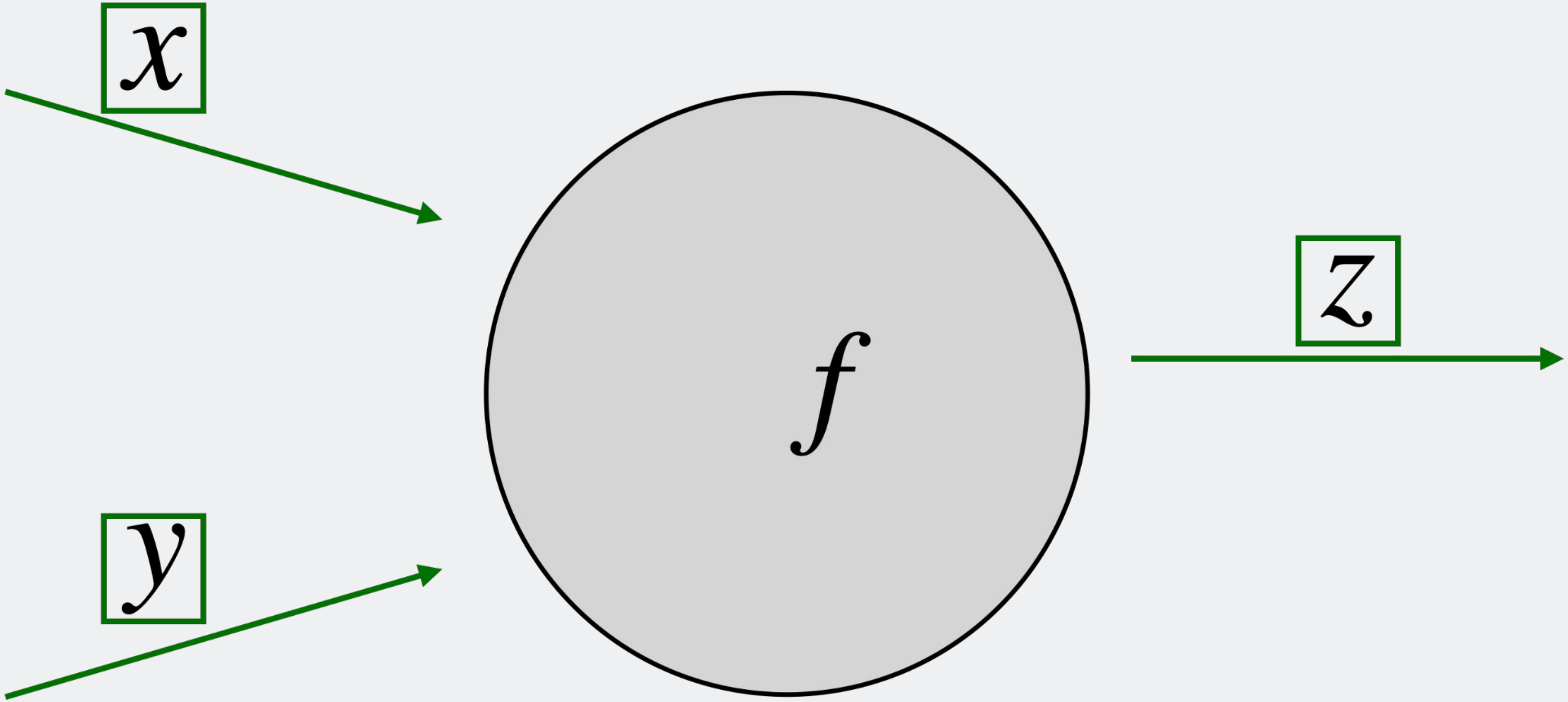
Derivative is **Jacobian**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M}$$

$$\left(\frac{\partial y}{\partial x}\right)_{i,j} = \frac{\partial y_j}{\partial x_i}$$

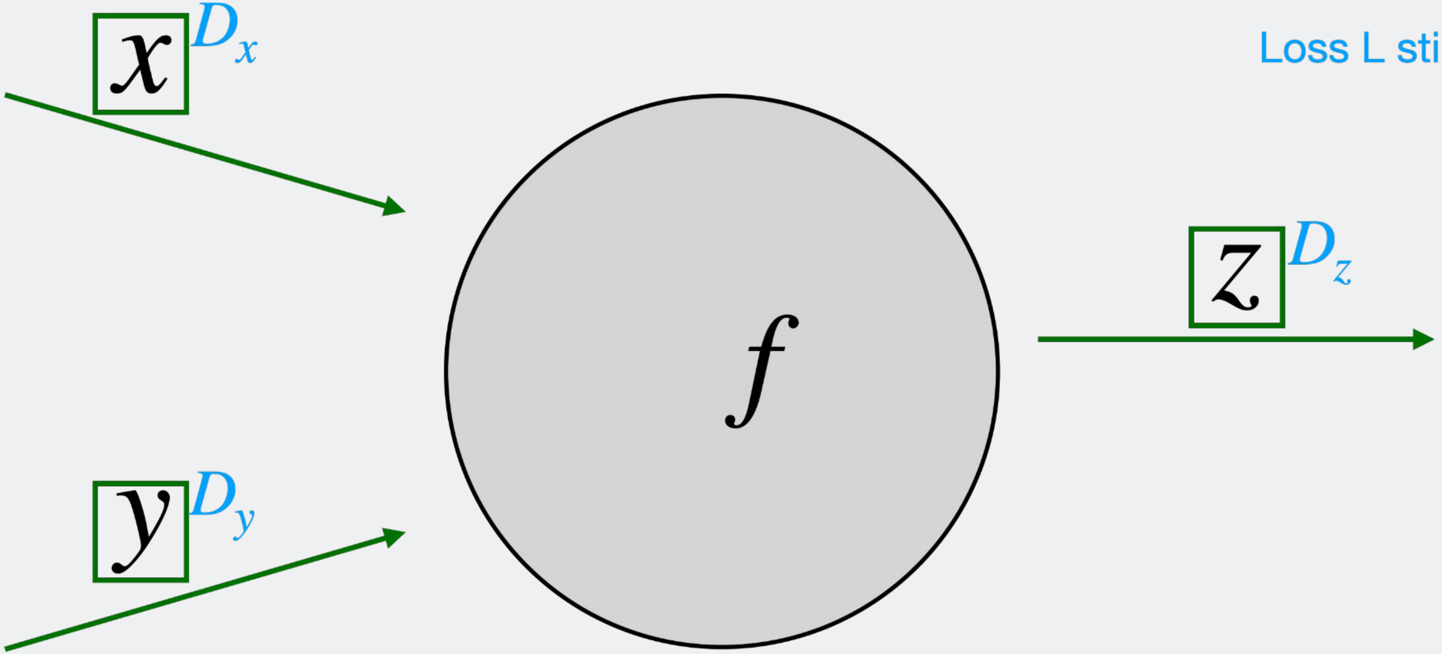
For each element of x, if it changes by a small amount then how much will each element of y change?

# Backprop with Vectors

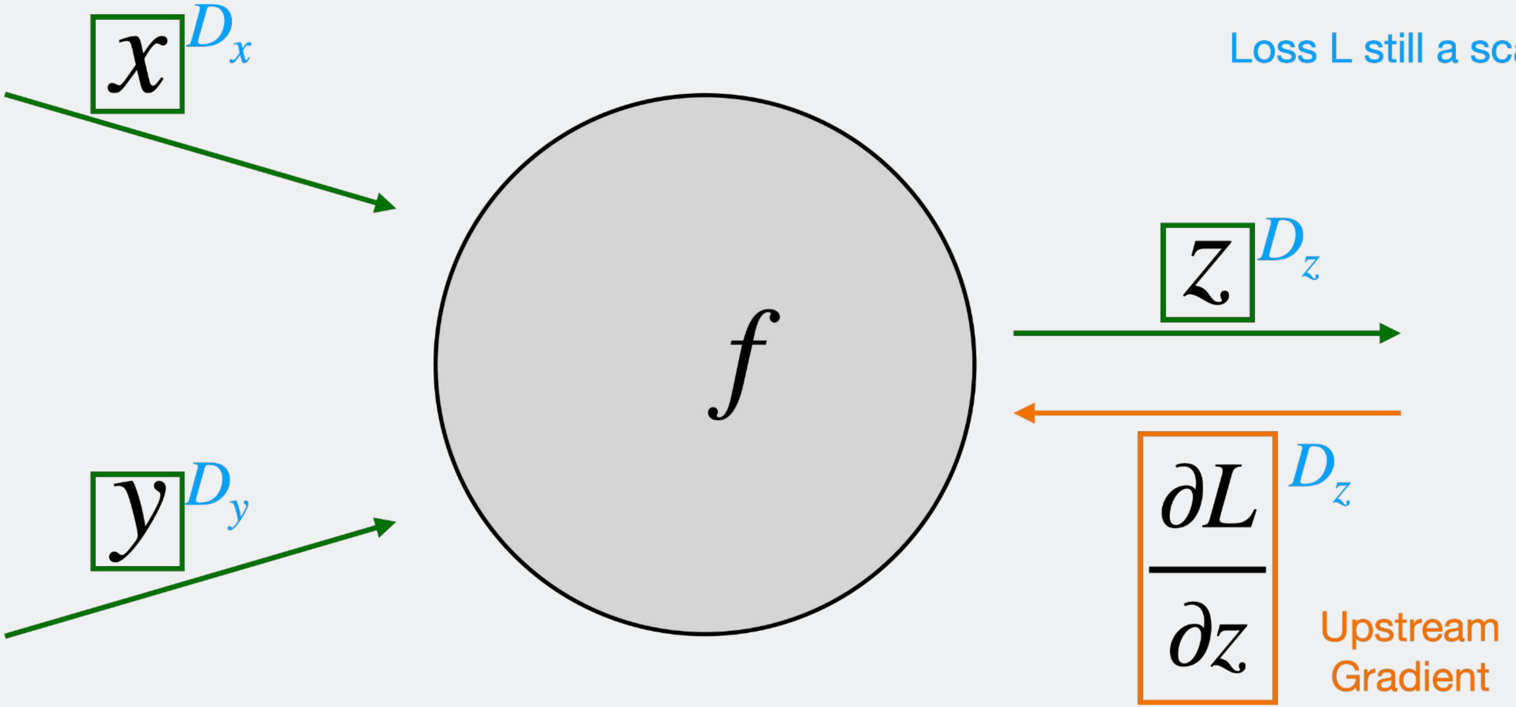


# Backprop with Vectors

Loss L still a scalar!



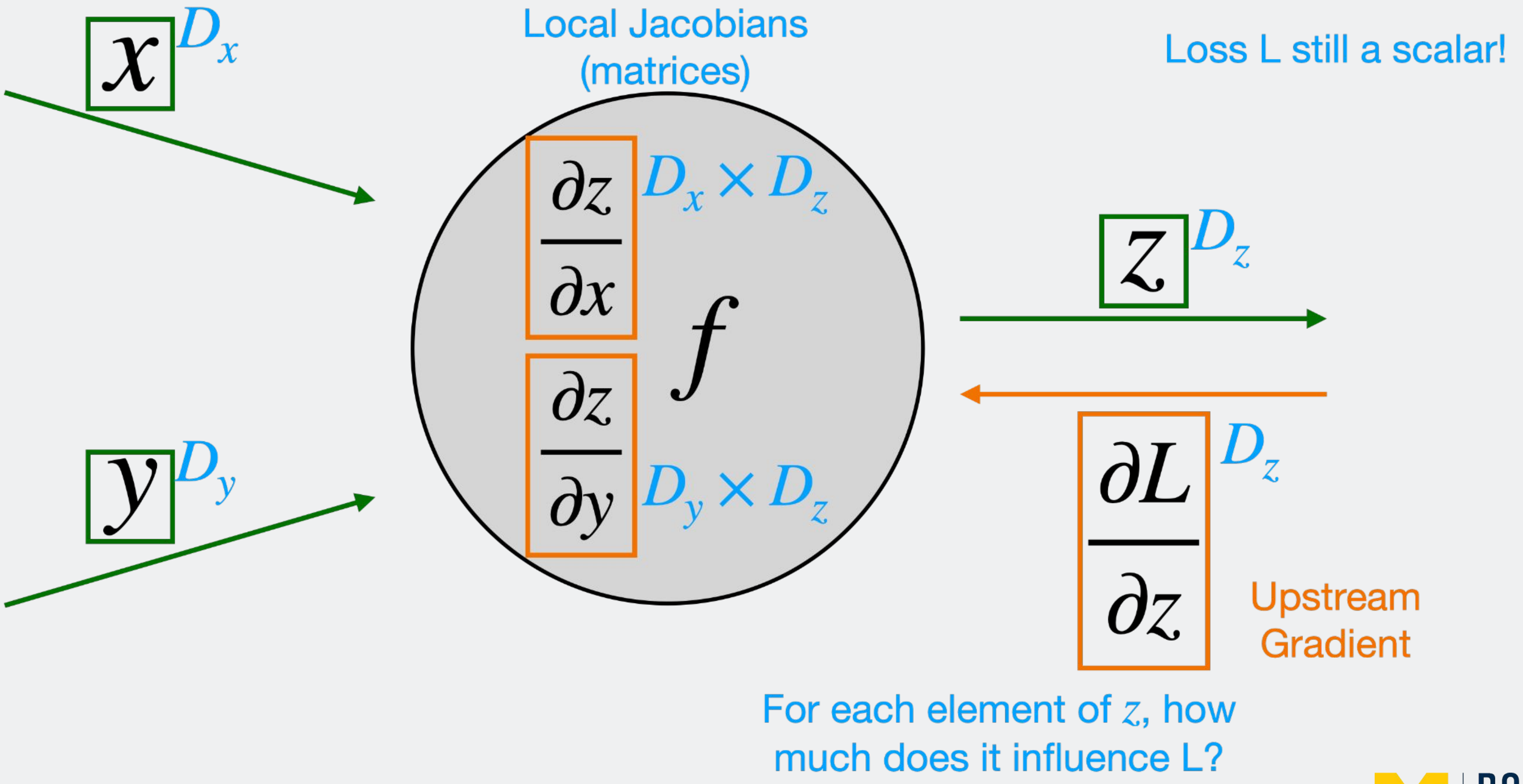
# Backprop with Vectors




Loss L still a scalar!

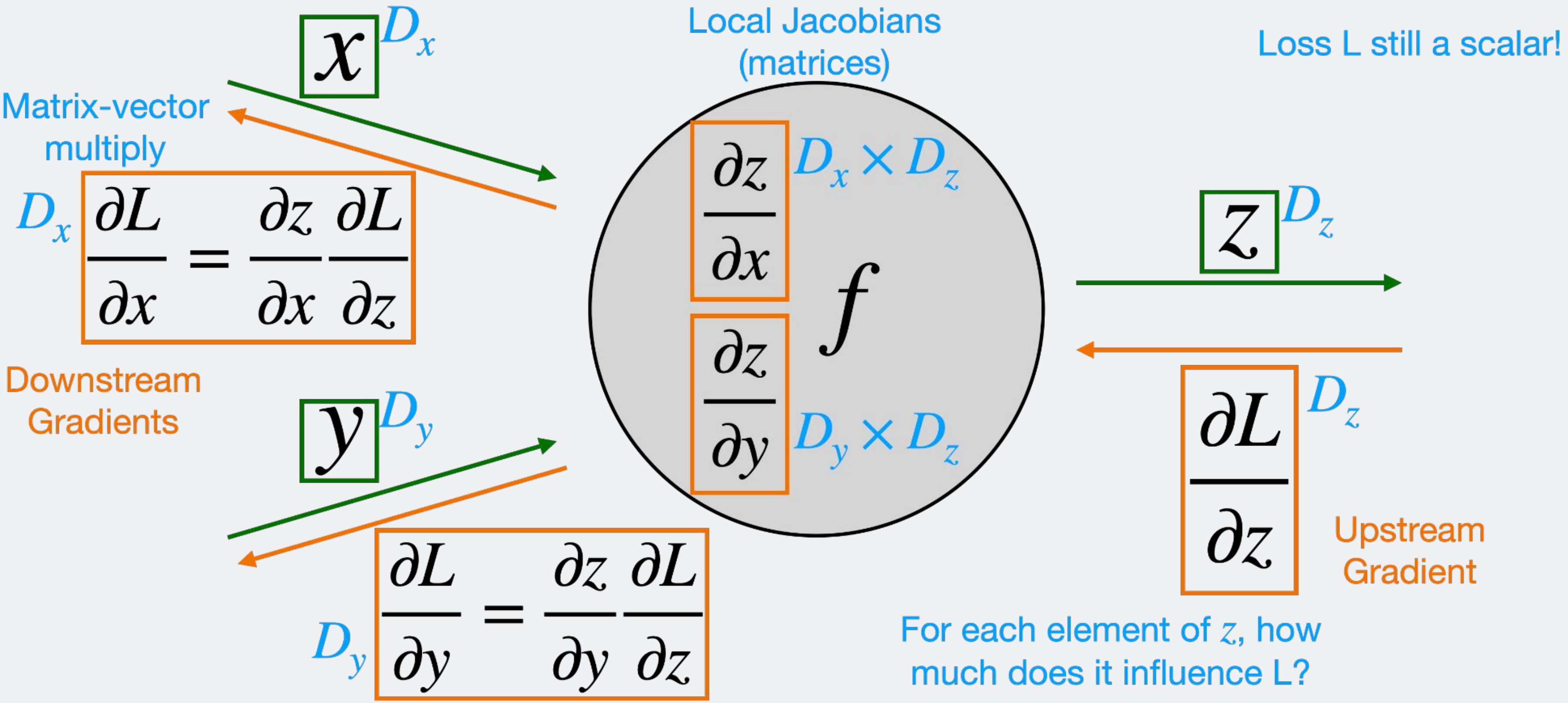
For each element of $z$, how much does it influence L?

# Backprop with Vectors

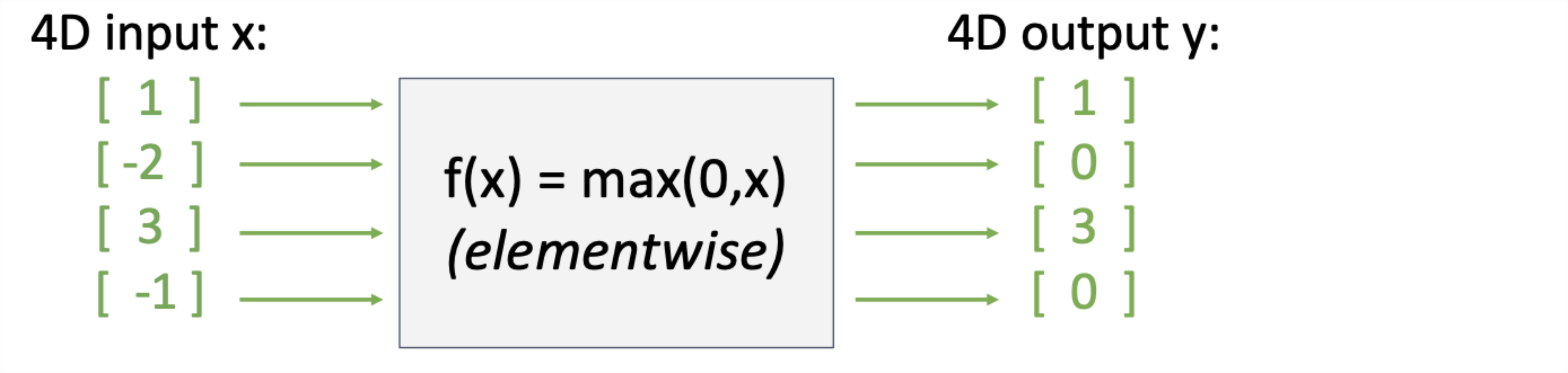



ROBOTICS

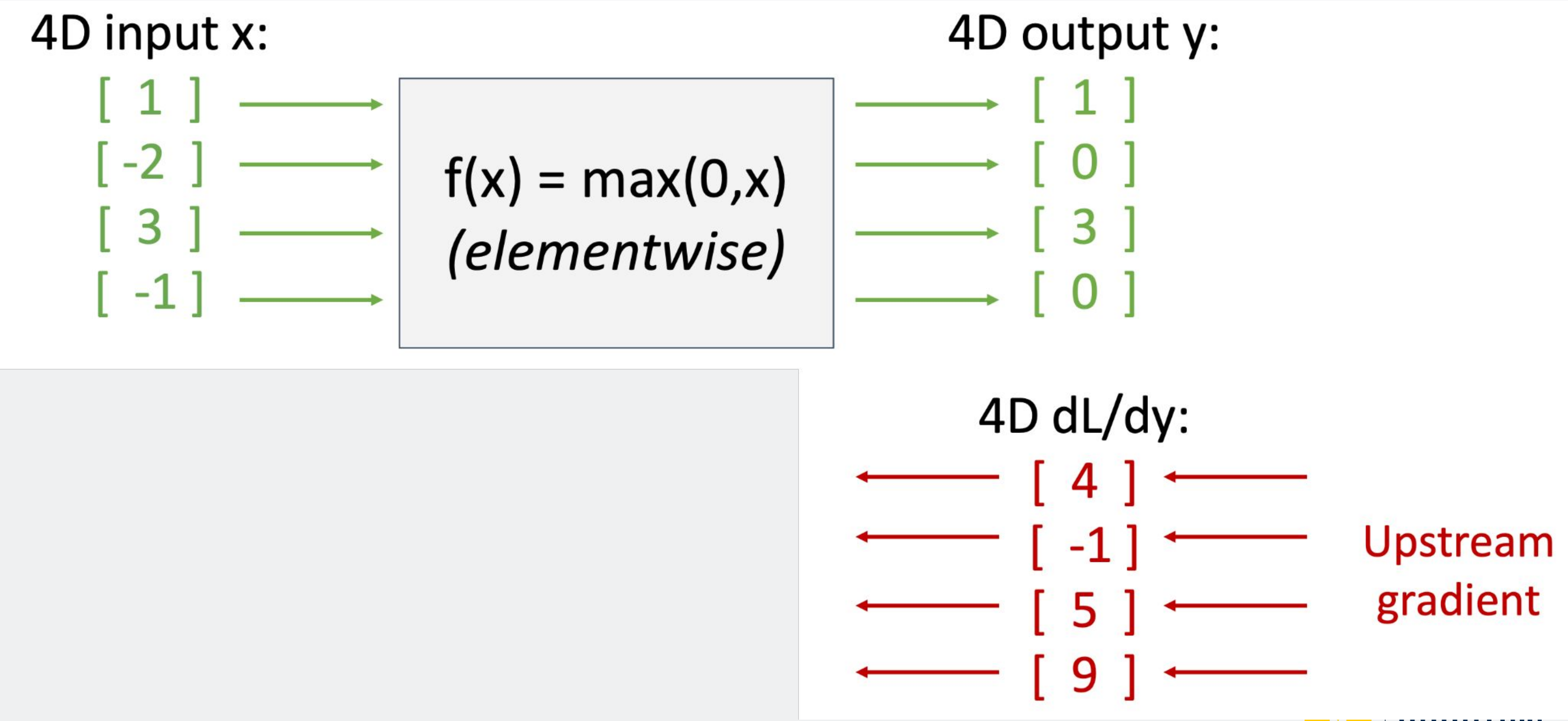
# Backprop with Vectors



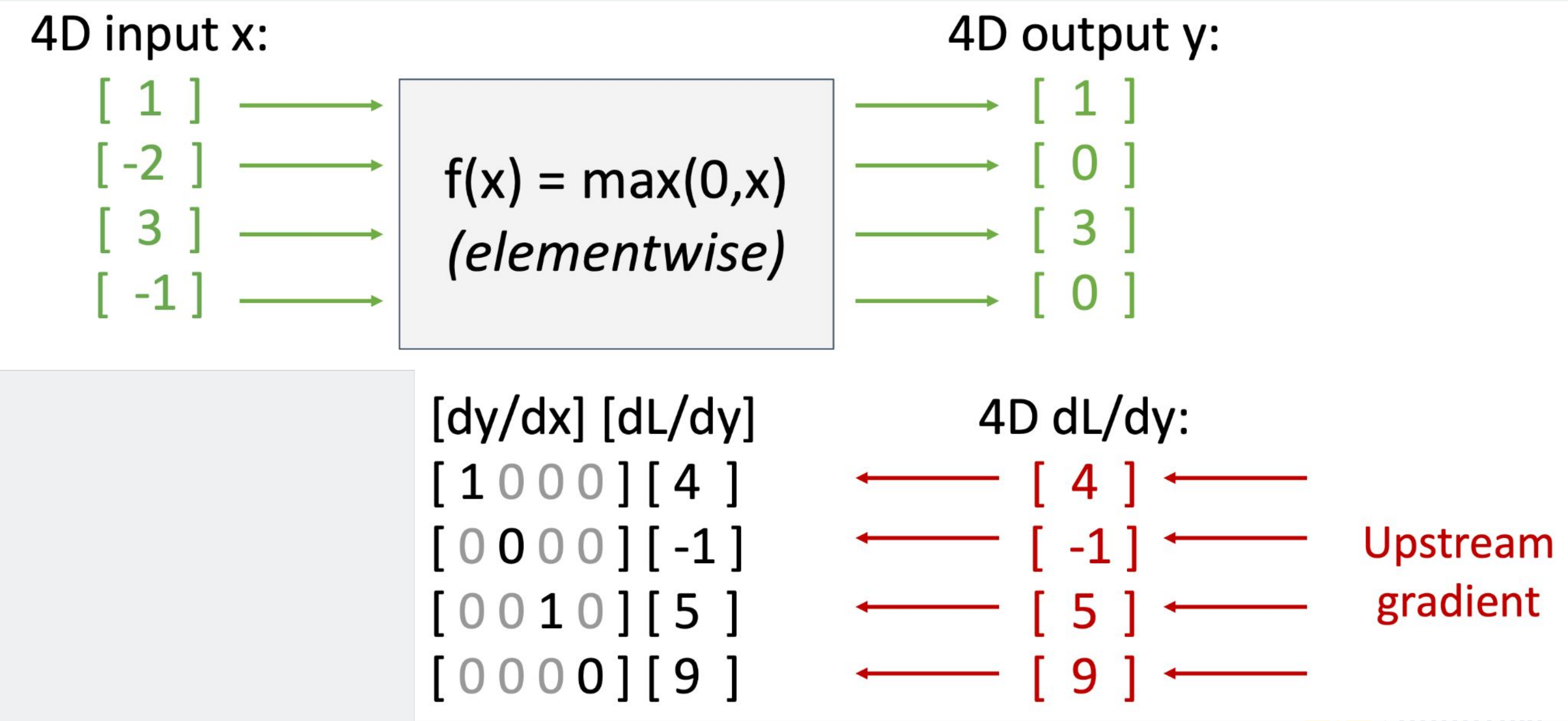
# Backprop with Vectors



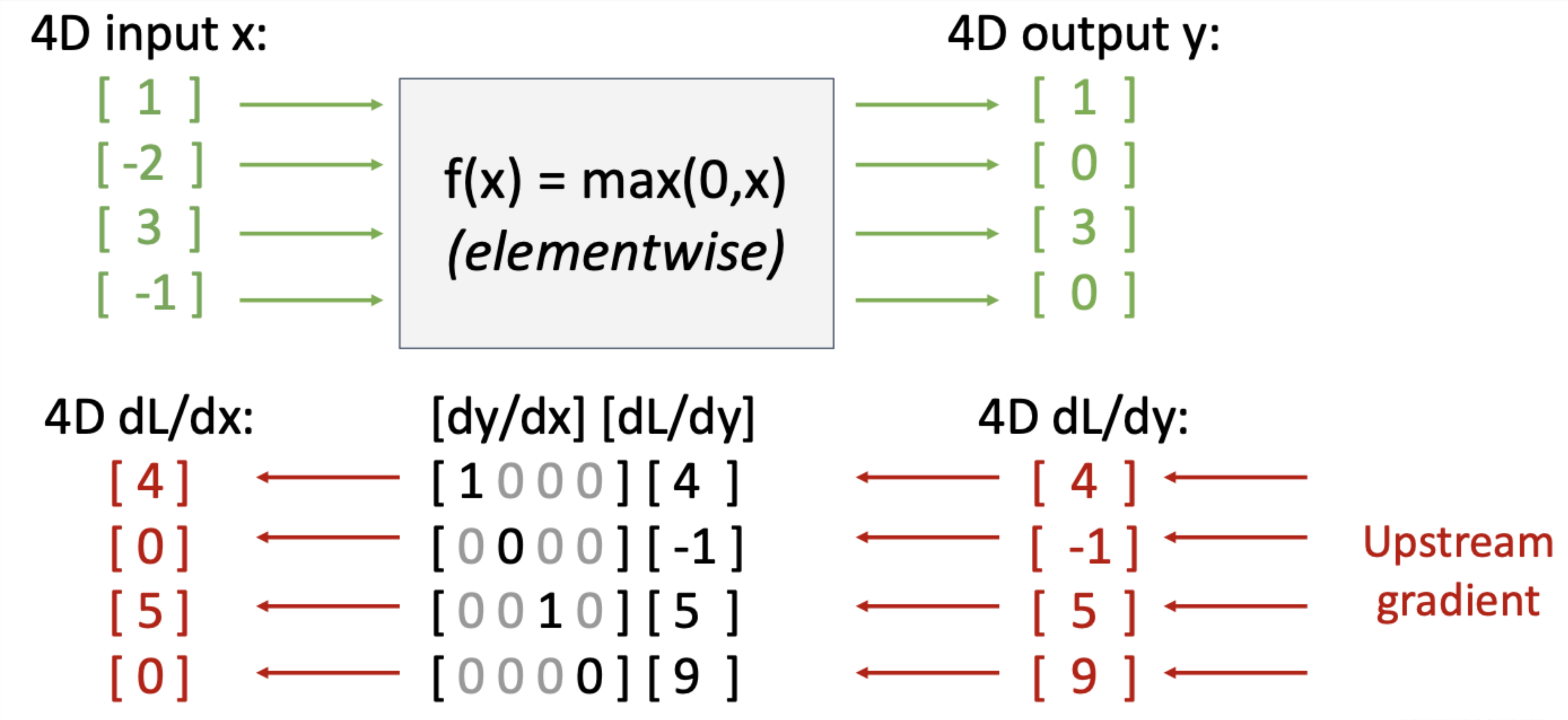
# Backprop with Vectors



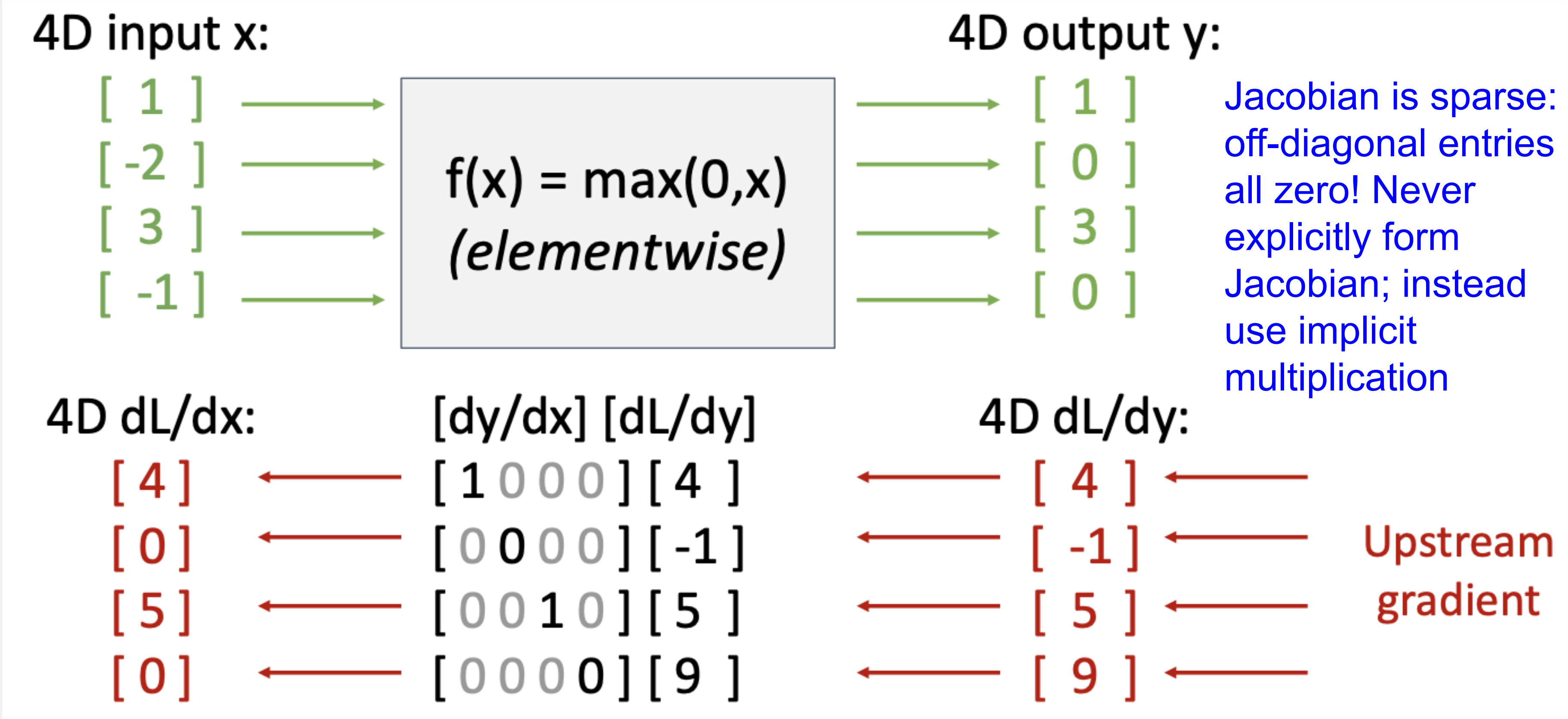
# Backprop with Vectors



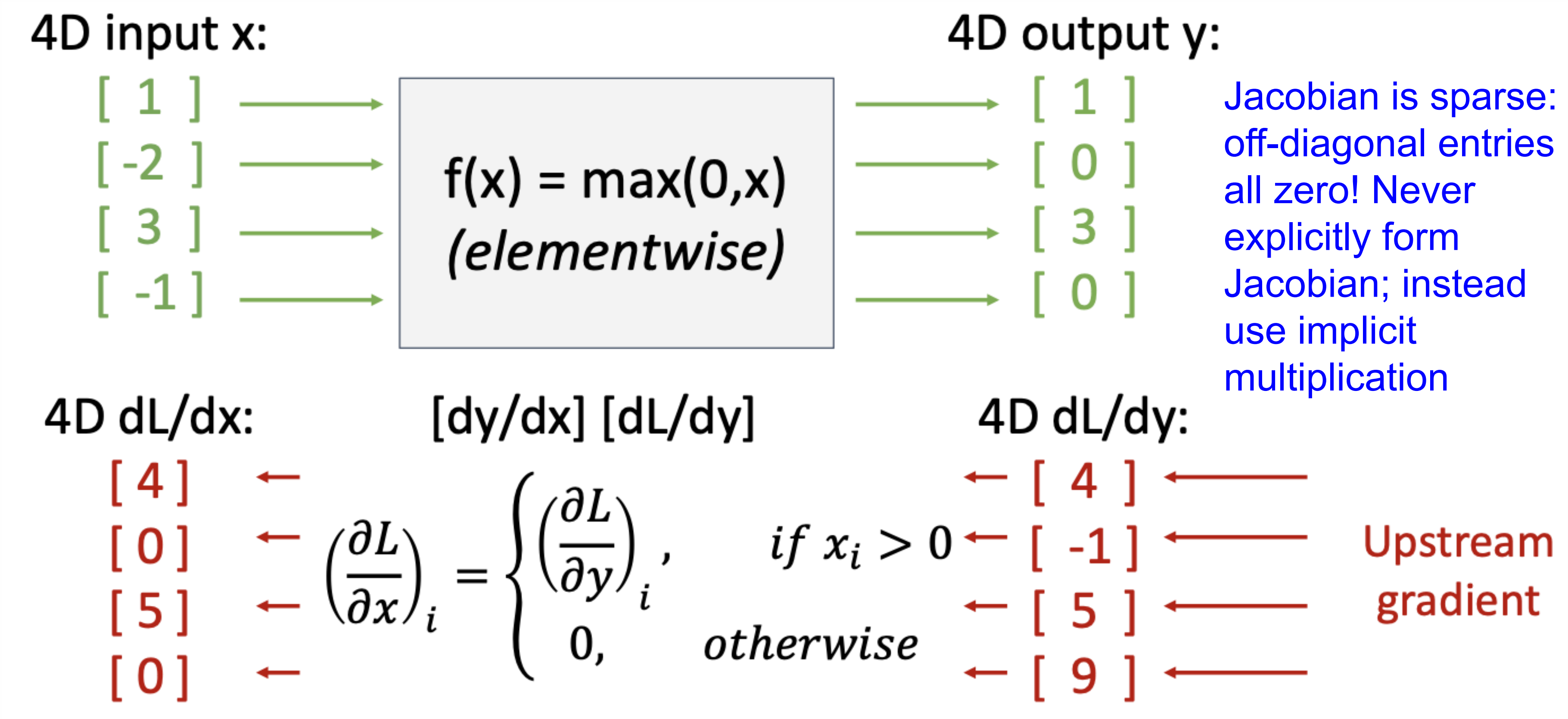
# Backprop with Vectors



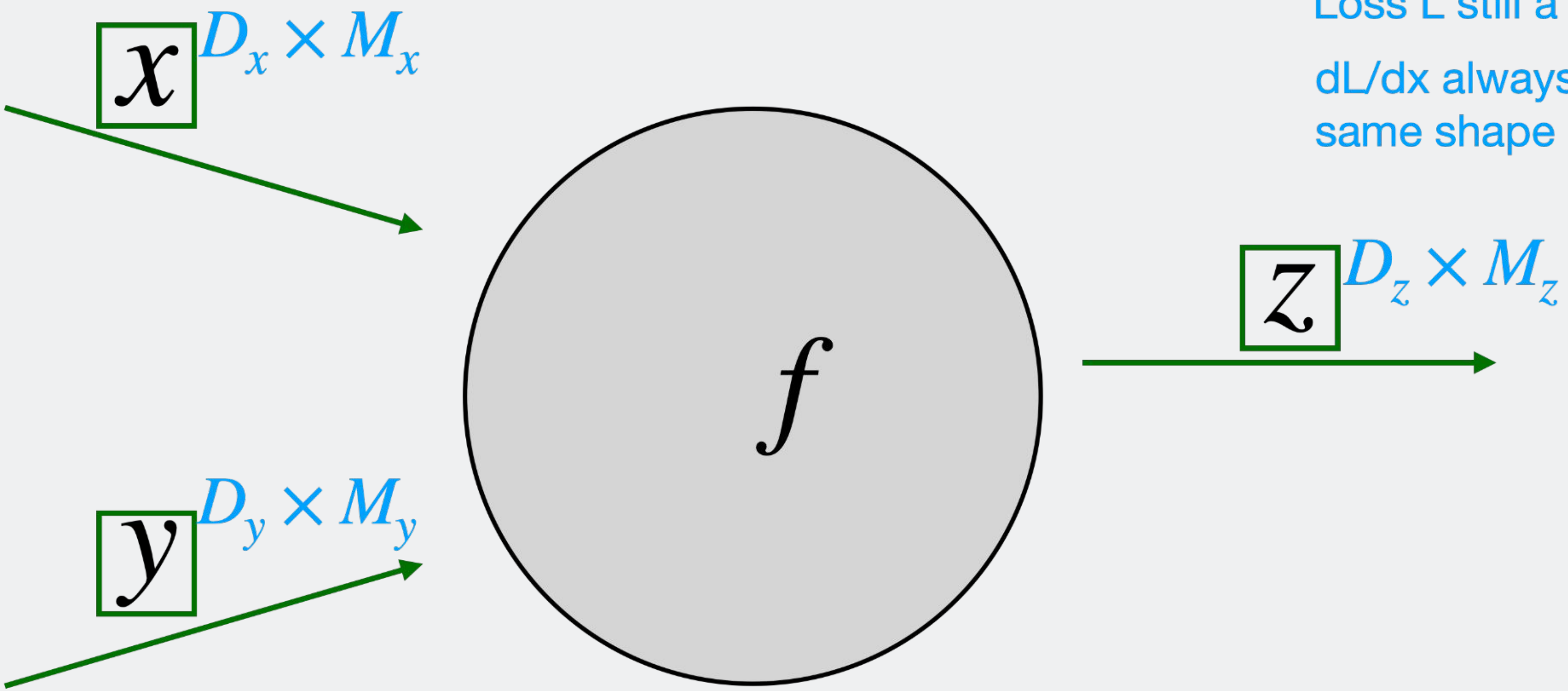
# Backprop with Vectors



# Backprop with Vectors



# Backprop with Matrices (or Tensors)



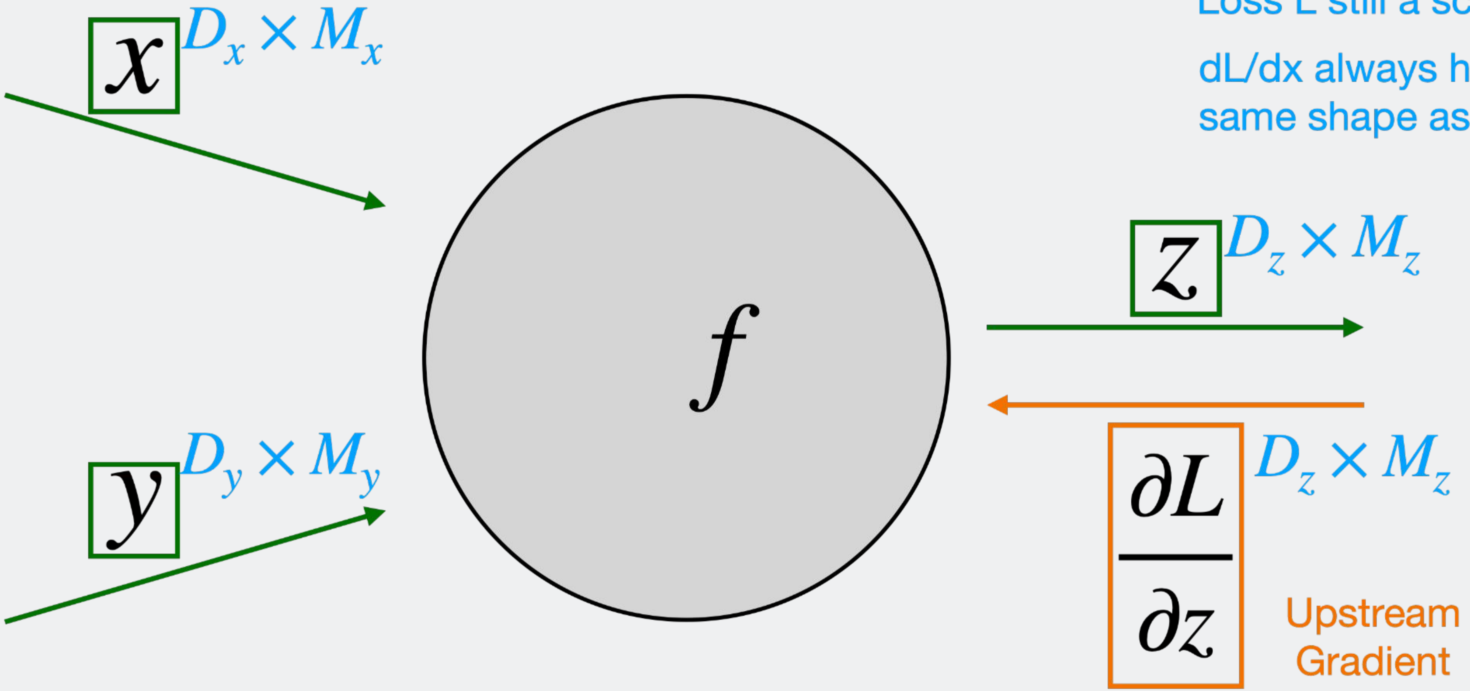

Loss L still a scalar!

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!



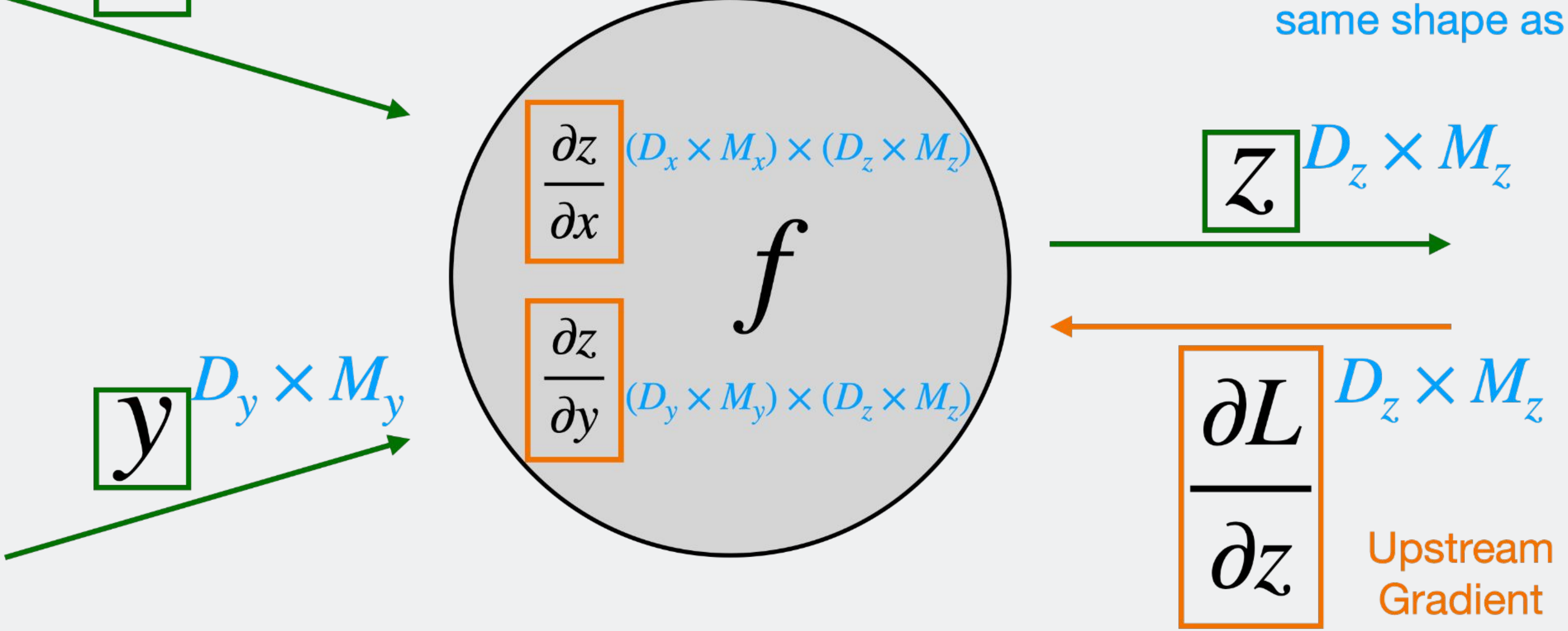

For each element of $z$, how much does it influence L?

# Backprop with Matrices (or Tensors)

Loss L still a scalar!

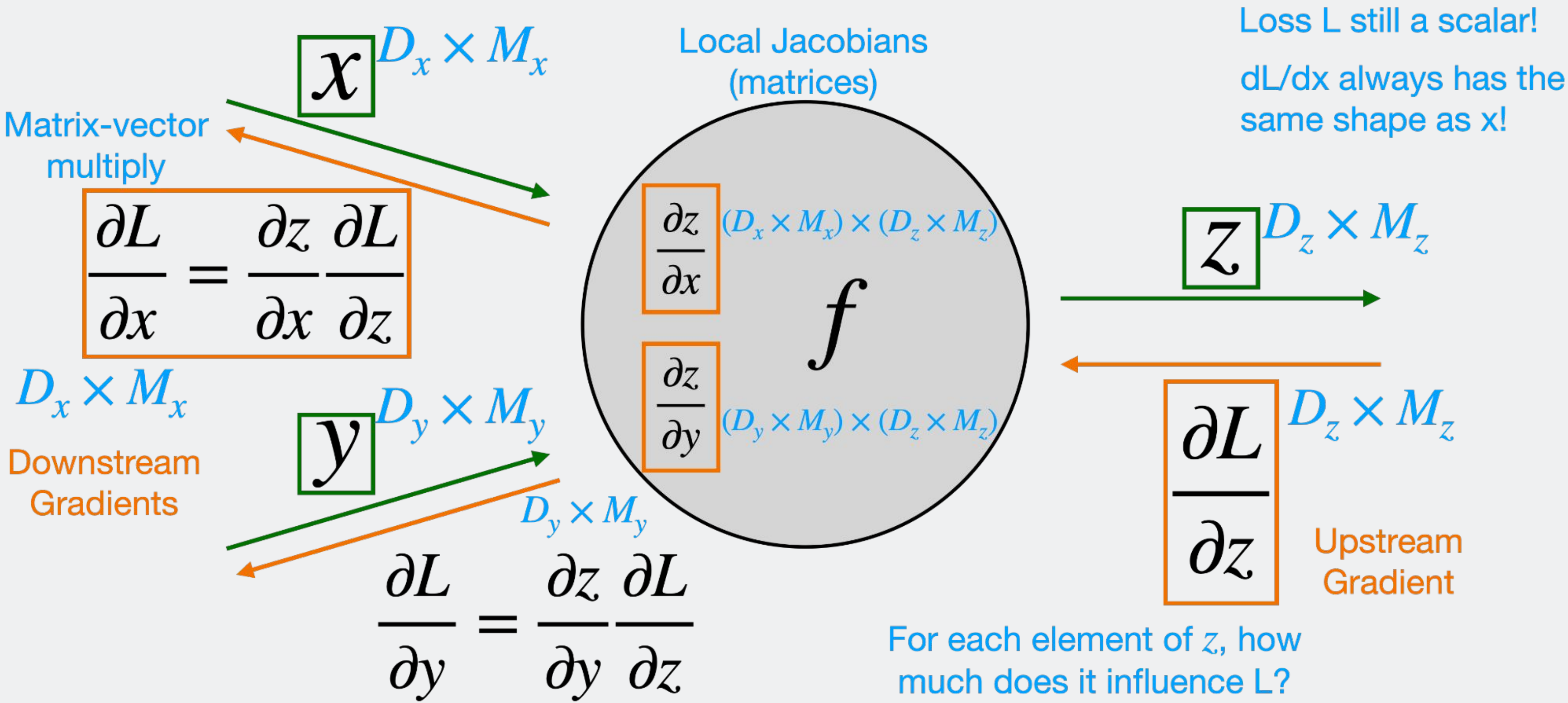
dL/dx always has the same shape as x!

Local Jacobians (matrices)




For each element of $z$, how much does it influence L?

# Backprop with Matrices (or Tensors)



# Example: Matrix Multiplication

x: [N×D]

[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1]
[ 2 1 3 2]
[ 3 2 1 -2]

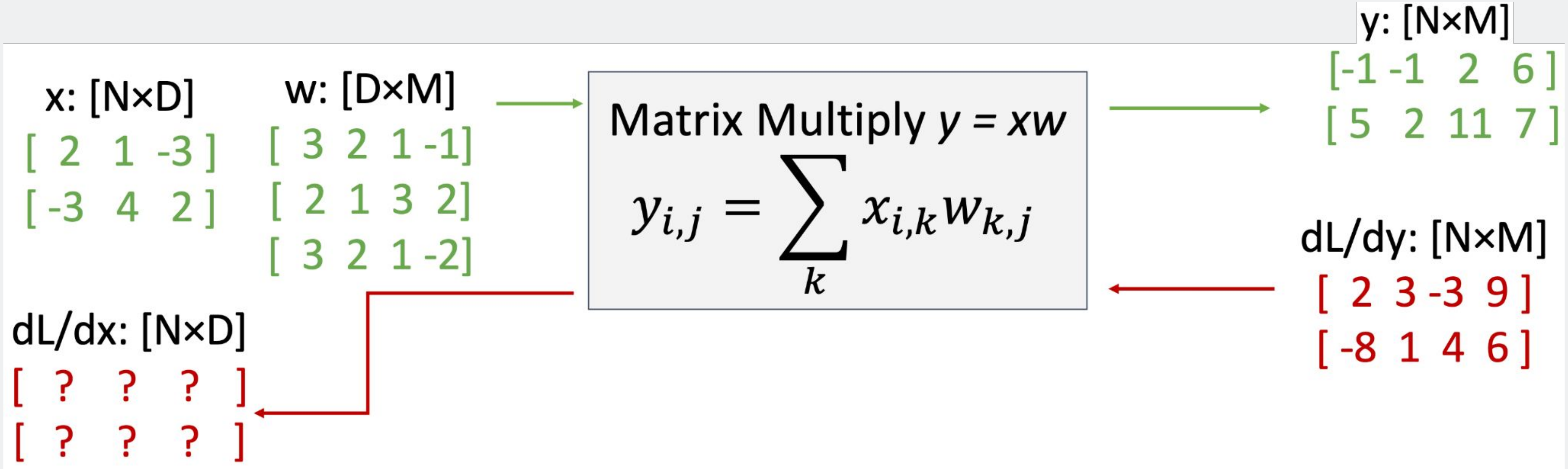
→ **Matrix Multiply** $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

→ y: [N×M]

[-1 -1 2 6 ]
[ 5 2 11 7 ]

# Example: Matrix Multiplication



# Example: Matrix Multiplication

x: [N×D]

[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1]
[ 2 1 3 2]
[ 3 2 1 -2]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

y: [N×M]

[-1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ ? ? ? ]
[ ? ? ? ]

**Jacobians**:
dy/dx: [(N×D)×(N×M)]
dy/dw: [(D×M)×(N×M)]

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

x: [N×D]

$$\begin{bmatrix} \boxed{2} & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

w: [D×M]

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

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

y: [N×M]

$$\begin{bmatrix} -1 & -1 & 2 & 6 \\ 5 & 2 & 11 & 7 \end{bmatrix}$$

dL/dy: [N×M]

$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

dL/dx: [N×D]

$$\begin{bmatrix} \boxed{?} & ? & ? \\ ? & ? & ? \end{bmatrix}$$

Local Gradient Slice:

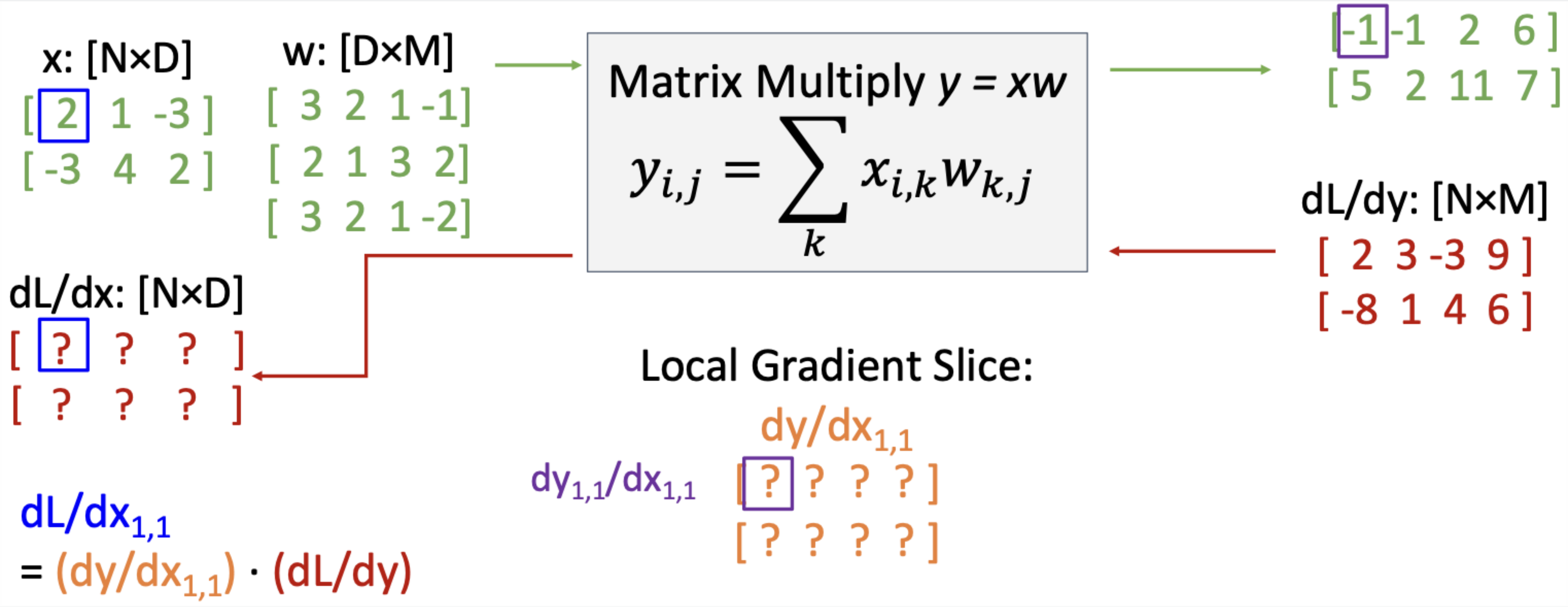
$dy/dx_{1,1}$

$$\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

$dL/dx_{1,1}$

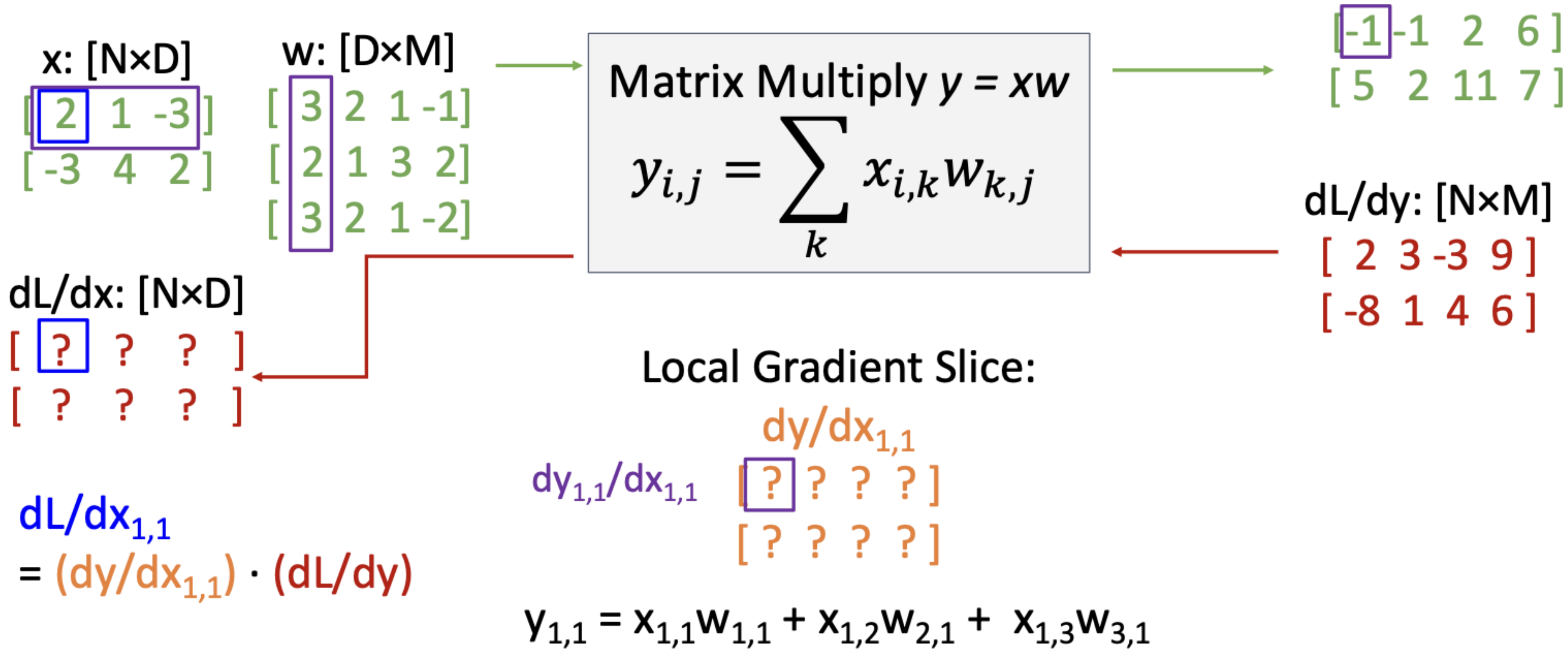
$= (dy/dx_{1,1}) \cdot (dL/dy)$

# Example: Matrix Multiplication

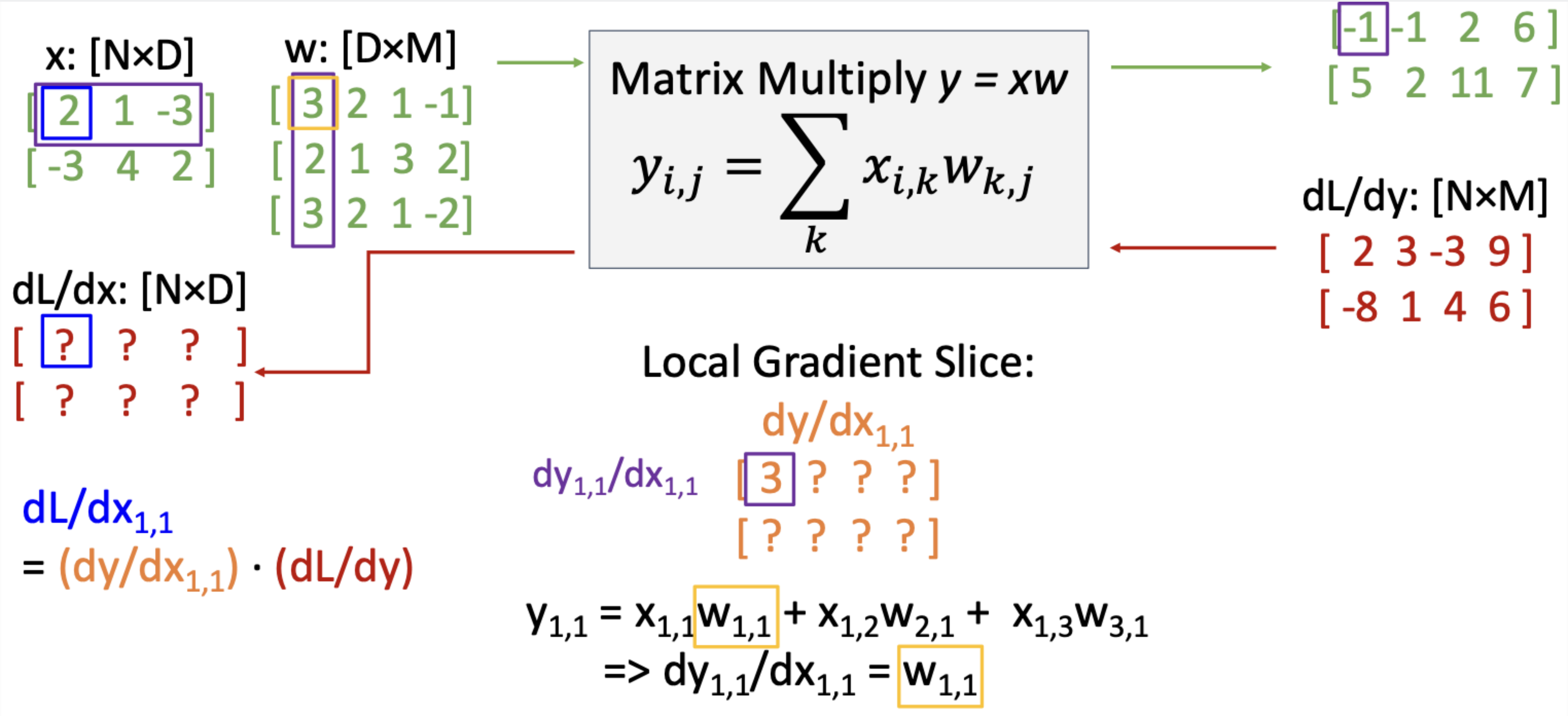



ROBOTICS

# Example: Matrix Multiplication



# Example: Matrix Multiplication



# Example: Matrix Multiplication

x: [N×D]

$$x = \begin{bmatrix} \boxed{2} & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

w: [D×M]

$$w = \begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & 3 & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

$$y = \begin{bmatrix} -1 & \boxed{-1} & 2 & 6 \\ 5 & 2 & 11 & 7 \end{bmatrix}$$

dL/dy: [N×M]

$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

dL/dx: [N×D]

$$\begin{bmatrix} \boxed{?} & ? & ? \\ ? & ? & ? \end{bmatrix}$$

Local Gradient Slice:

$dy/dx_{1,1}$

$dy_{1,2}/dx_{1,1}$

$$\begin{bmatrix} 3 & \boxed{?} & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

$dL/dx_{1,1}$

$= (dy/dx_{1,1}) \cdot (dL/dy)$

# Example: Matrix Multiplication

x: [N×D]

[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1]
[ 2 1 3 2]
[ 3 2 1 -2]

**Matrix Multiply $y = xw$**

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ ? ? ? ]
[ ? ? ? ]

Local Gradient Slice:

$dy/dx_{1,1}$

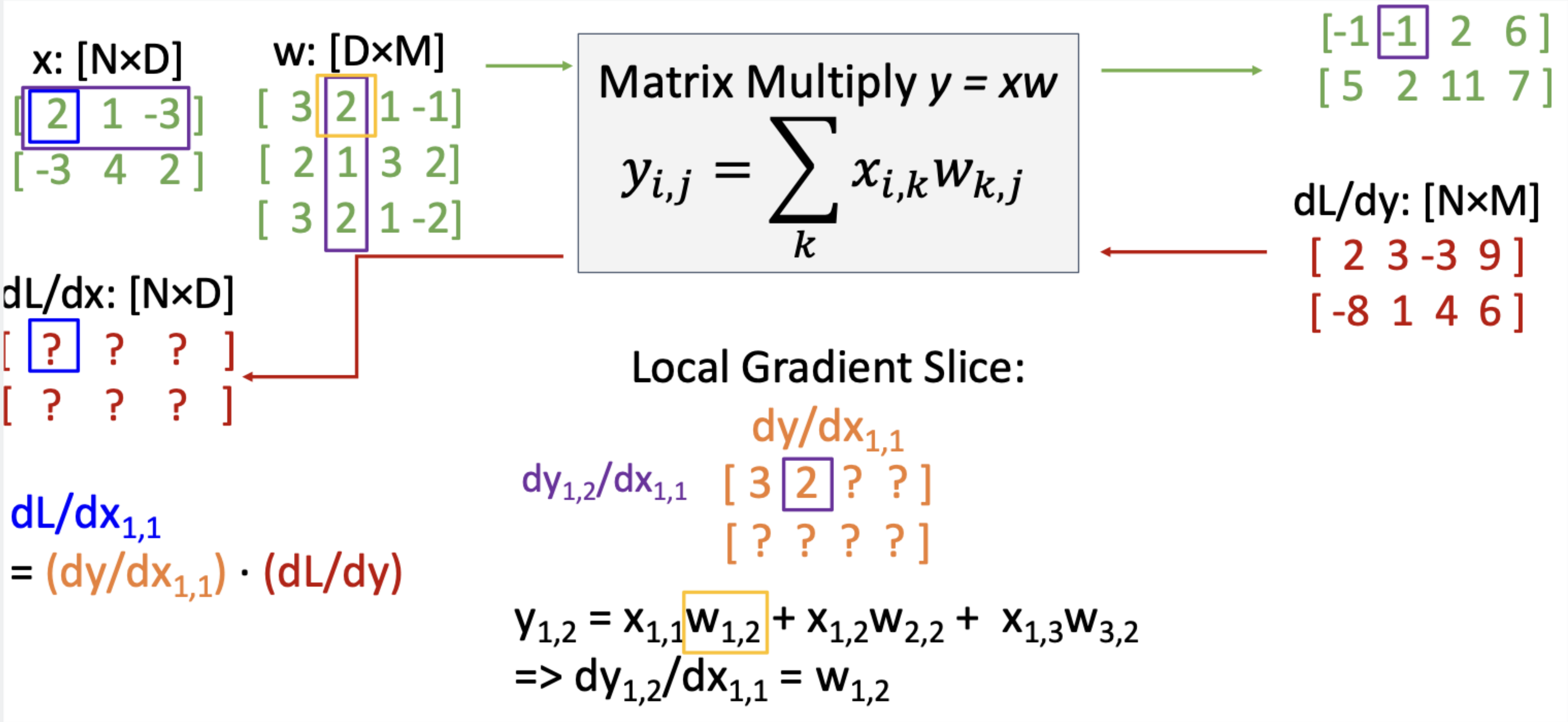
$dy_{1,2}/dx_{1,1}$

[ 3 ? ? ? ]
[ ? ? ? ? ]

$dL/dx_{1,1}$
$= (dy/dx_{1,1}) \cdot (dL/dy)$

$$y_{1,2} = x_{1,1}w_{1,2} + x_{1,2}w_{2,2} + x_{1,3}w_{3,2}$$

# Example: Matrix Multiplication



# Example: Matrix Multiplication

x: [N×D]

[ [2] 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 [1 -1] ]
[ 2 1 3 2 ]
[ 3 2 1 -2 ]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

y:

[ -1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ [?] ? ? ]
[ ? ? ? ]

Local Gradient Slice:

$dy/dx_{1,1}$

[ 3 2 [1 -1] ]
[ ? ? ? ? ]

$dy_{1,2}/dx_{1,1}$

$dL/dx_{1,1}$

$= (dy/dx_{1,1}) \cdot (dL/dy)$

# Example: Matrix Multiplication

x: [N×D]

[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1 ]
[ 2 1 3 2 ]
[ 3 2 1 -2 ]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[ -1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ ? ? ? ]
[ ? ? ? ]

Local Gradient Slice:

$dy/dx_{1,1}$

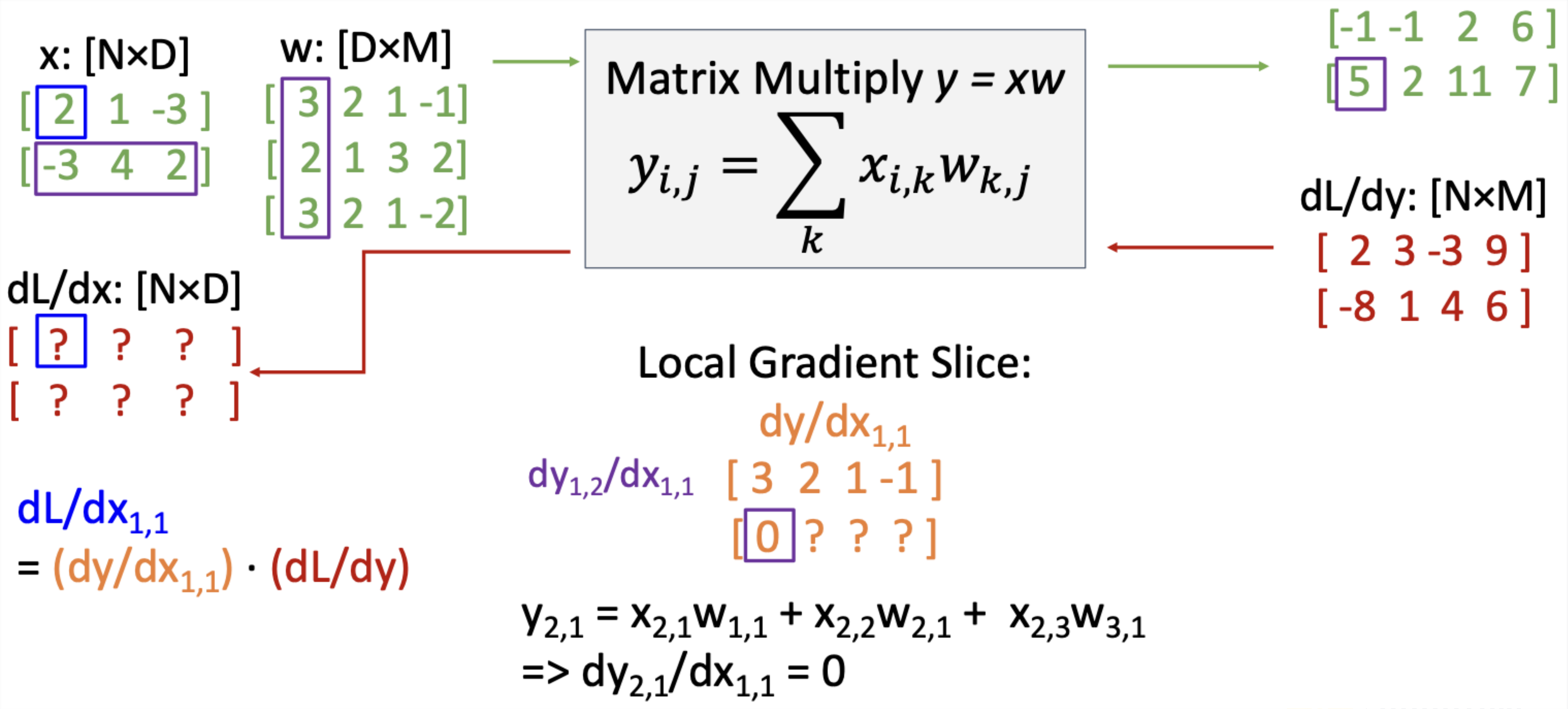
$dy_{1,2}/dx_{1,1}$

[ 3 2 1 -1 ]
[ ? ? ? ? ]

$y_{2,1} = x_{2,1}w_{1,1} + x_{2,2}w_{2,1} + x_{2,3}w_{3,1}$

$dL/dx_{1,1}$
$= (dy/dx_{1,1}) \cdot (dL/dy)$

# Example: Matrix Multiplication



# Example: Matrix Multiplication

x: [N×D]

[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1]
[ 2 1 3 2]
[ 3 2 1 -2]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ ? ? ? ]
[ ? ? ? ]

Local Gradient Slice:

$dy/dx_{1,1}$

$dy_{1,2}/dx_{1,1}$

[ 3 2 1 -1 ]
[ 0 0 0 0 ]

$dL/dx_{1,1}$

$= (dy/dx_{1,1}) \cdot (dL/dy)$

# Example: Matrix Multiplication

x: [N×D]

[ [2] 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1]
[ 2 1 3 2]
[ 3 2 1 -2]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ [?] ? ? ]
[ ? ? ? ]

Local Gradient Slice:

$dy/dx_{1,1}$

[ 3 2 1 -1 ]
[ 0 0 0 0 ]

$dL/dx_{1,1}$

$= (dy/dx_{1,1}) \cdot (dL/dy)$

# Example: Matrix Multiplication

x: [N×D]

[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1 ]
[ 2 1 3 2 ]
[ 3 2 1 -2 ]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[ -1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ 0 ? ? ]
[ ? ? ? ]

Local Gradient Slice:

$dy/dx_{1,1}$

[ 3 2 1 -1 ]
[ 0 0 0 0 ]

$dL/dx_{1,1}$

$= (dy/dx_{1,1}) \cdot (dL/dy)$

$= (w_{1,:}) \cdot (dL/dy_{1,:})$

$= 3*2 + 2*3 + 1*(-3) + (-1)*9 = 0$

# Example: Matrix Multiplication

x: [N×D]

[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1]
[ 2 1 3 2]
[ 3 2 1 -2]

**Matrix Multiply $y = xw$**

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ 0 ? ? ]
[ ? ? -30 ]

**Local Gradient Slice:**

$dy/dx_{2,3}$

[ 0 0 0 0 ]
[ 3 2 1 -2 ]

$dL/dx_{2,3}$

$= (dy/dx_{2,3}) \cdot (dL/dy)$

# Example: Matrix Multiplication

x: [N×D]
[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]
[ 3 2 1 -1 ]
[ 2 1 3 2 ]
[ 3 2 1 -2 ]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[ -1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]
[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]
[ 0 ? ? ]
[ ? ? -30 ]

Local Gradient Slice:

$dy/dx_{2,3}$
[ 0 0 0 0 ]
[ 3 2 1 -2 ]

$dL/dx_{2,3}$
= $(dy/dx_{2,3}) \cdot (dL/dy)$
= $(w_{3,:}) \cdot (dL/dy_{2,:})$
= 3*(-8) + 2*1 + 1*4 + (-2)*6 = -30

# Example: Matrix Multiplication

x: [N×D]

[ 2 1 -3 ]
[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -1]
[ 2 1 3 2]
[ 3 2 1 -2]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6 ]
[ 5 2 11 7 ]

dL/dy: [N×M]

[ 2 3 -3 9 ]
[ -8 1 4 6 ]

dL/dx: [N×D]

[ 0 16 -9 ]
[-24 9 -30 ]

$dL/dx_{i,j}$
$= (dy/dx_{i,j}) \cdot (dL/dy)$
$= (w_{j,:}) \cdot (dL/dy_{i,:})$

# Example: Matrix Multiplication

x: [N×D]

$$\begin{bmatrix} 2 & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

w: [D×M]

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

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

$$\begin{bmatrix} -1 & -1 & 2 & 6 \\ 5 & 2 & 11 & 7 \end{bmatrix}$$

dL/dy: [N×M]

$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

dL/dx: [N×D]

$$\begin{bmatrix} 0 & 16 & -9 \\ -24 & 9 & -30 \end{bmatrix}$$

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

[N x D] [N x M] [M x D]

$$dL/dx_{i,j} = (dy/dx_{i,j}) \cdot (dL/dy) = (w_{j,:}) \cdot (dL/dy_{i,:})$$

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

# Example: Matrix Multiplication

x: [N×D]

$$\begin{bmatrix} 2 & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

w: [D×M]

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

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

$$\begin{bmatrix} -1 & -1 & 2 & 6 \\ 5 & 2 & 11 & 7 \end{bmatrix}$$

dL/dy: [N×M]

$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

dL/dx: [N×D]

$$\begin{bmatrix} 0 & 16 & -9 \\ -24 & 9 & -30 \end{bmatrix}$$

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

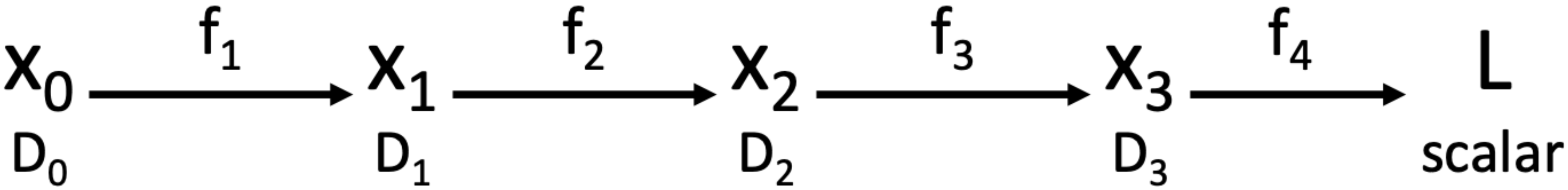
[N x D] [N x M] [M x D]

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

[D x M] [D x N] [N x M]

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

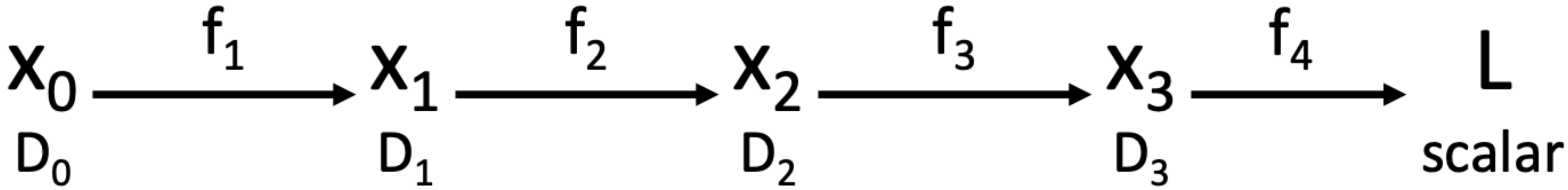
# Backpropagation: Another View




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

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

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

# Reverse-Mode Automatic Differentiation

$$x_0 \xrightarrow{f_1} x_1 \xrightarrow{f_2} x_2 \xrightarrow{f_3} x_3 \xrightarrow{f_4} L$$

$D_0$   $D_1$   $D_2$   $D_3$   scalar

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

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

[$D_0$ x $D_1$] [$D_1$ x $D_2$] [$D_2$ x $D_3$] [$D_3$]

# Reverse-Mode Automatic Differentiation

$$x_0 \xrightarrow{f_1} x_1 \xrightarrow{f_2} x_2 \xrightarrow{f_3} x_3 \xrightarrow{f_4} L$$

$D_0$ | $D_1$ | $D_2$ | $D_3$ | scalar

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

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

$[D_0 \text{ x } D_1]$ $[D_1 \text{ x } D_2]$ $[D_2 \text{ x } D_3]$ $[D_3]$

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

# Reverse-Mode Automatic Differentiation

$$x_0 \xrightarrow{f_1} x_1 \xrightarrow{f_2} x_2 \xrightarrow{f_3} x_3 \xrightarrow{f_4} L$$

$D_0$ $D_1$ $D_2$ $D_3$ scalar

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

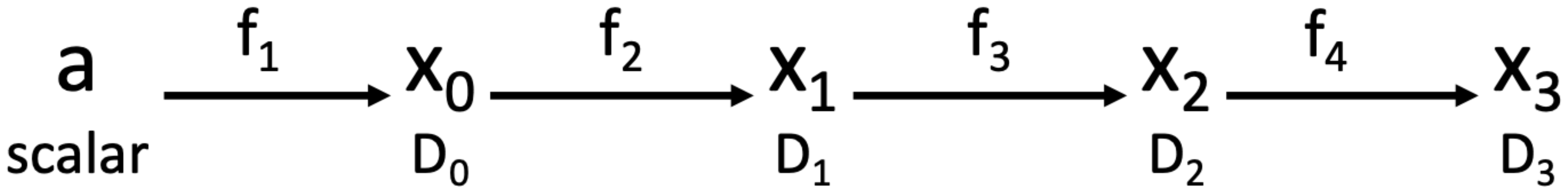
$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0}\right)\left(\frac{\partial x_2}{\partial x_1}\right)\left(\frac{\partial x_3}{\partial x_2}\right)\left(\frac{\partial L}{\partial x_3}\right)$$

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

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

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

# Forward-Mode Automatic Differentiation




Chain rule

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

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

# Forward-Mode Automatic Differentiation

$$\underset{\text{scalar}}{a} \xrightarrow{f_1} \underset{D_0}{x_0} \xrightarrow{f_2} \underset{D_1}{x_1} \xrightarrow{f_3} \underset{D_2}{x_2} \xrightarrow{f_4} \underset{D_3}{x_3}$$

Computing products <u>left-to-right</u> avoids matrix-matrix products; only needs matrix-vector

Chain rule

$$\frac{\partial x_3}{\partial a} = \overrightarrow{\underbrace{\left(\frac{\partial x_0}{\partial a}\right)}_{[D_0]} \underbrace{\left(\frac{\partial x_1}{\partial x_0}\right)}_{[D_0 \times D_1]} \underbrace{\left(\frac{\partial x_2}{\partial x_1}\right)}_{[D_1 \times D_2]} \underbrace{\left(\frac{\partial x_3}{\partial x_2}\right)}_{[D_2 \times D_3]}}$$

# Forward-Mode Automatic Differentiation

$$a \xrightarrow{f_1} x_0 \xrightarrow{f_2} x_1 \xrightarrow{f_3} x_2 \xrightarrow{f_4} x_3$$

scalar $\quad D_0 \quad D_1 \quad D_2 \quad D_3$

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

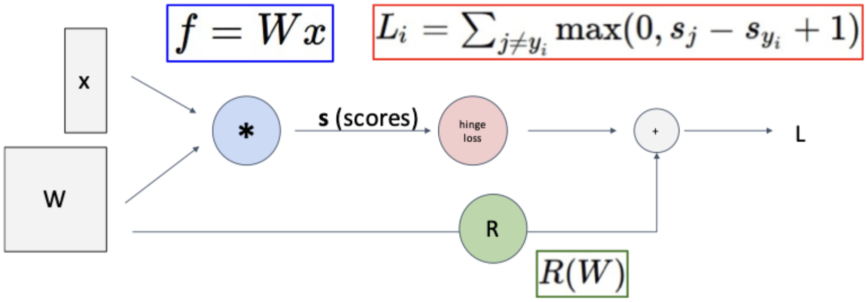
$$\frac{\partial x_3}{\partial a} = \left(\frac{\partial x_0}{\partial a}\right)\left(\frac{\partial x_1}{\partial x_0}\right)\left(\frac{\partial x_2}{\partial x_1}\right)\left(\frac{\partial x_3}{\partial x_2}\right)$$

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

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

# Summary

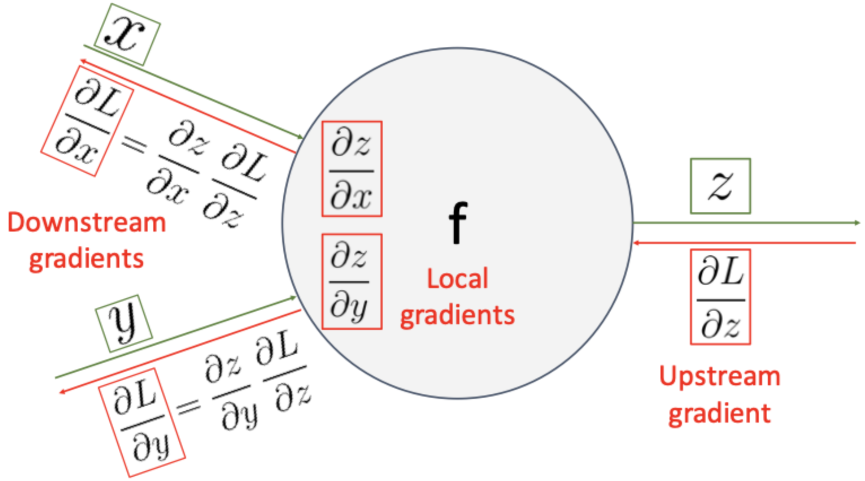
Represent complex expressions as **computational graphs**




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

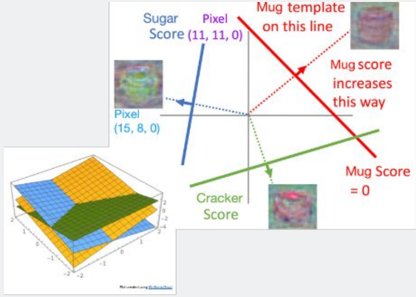
```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)

  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

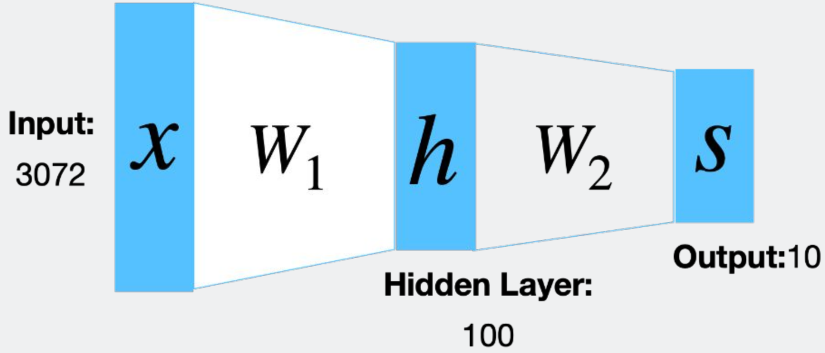
```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz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

# Summary

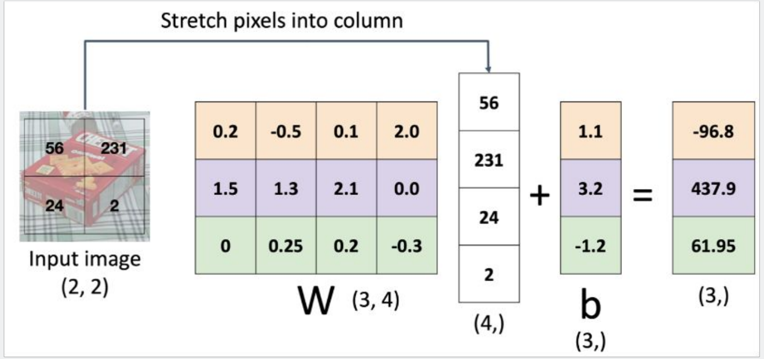
Reminder:
20260127 NN + 20260128 BackProp
**Quizzes released on Canvas**
**(Due next Tuesday Feb. 3, 2026)**




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




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




Next up: Convolutional Neural Networks

ROBOTICS

# Announcement

P1 - Due Feb.1, 2026

P2 will be released soon (upcoming)

Next Monday Feb.2nd: Lecture 7 CNN

Next Tuesday Feb.3rd: in Vis Studio (1401 Duderstadt)