# CS 4644 / 7643-A: LECTURE 5
# DANFEI XU

Topics:

- Backpropagation / Automatic Differentiation
- Neural Networks
- Jacobians

- **PS1/HW1 due Sep 19th**
- Resources:
  - These lectures
  - Matrix calculus for deep learning
  - Gradients notes and MLP/ReLU Jacobian notes.
  - Assignment (@41) and matrix calculus (@46)

- **Project:**
  - Teaming thread on piazza
  - Proposal due Sep $27^{th}$
  - Release project registration form soon

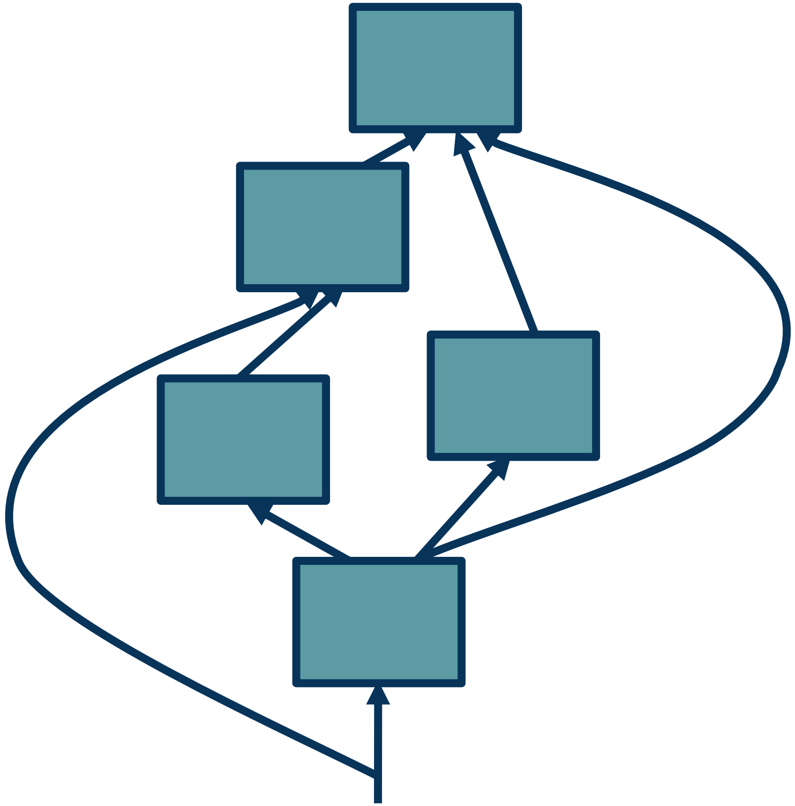
Recap: Computation Graph

To develop a general algorithm for this, we will view the function as a **computation graph**

Graph can be any **directed acyclic graph (DAG)**

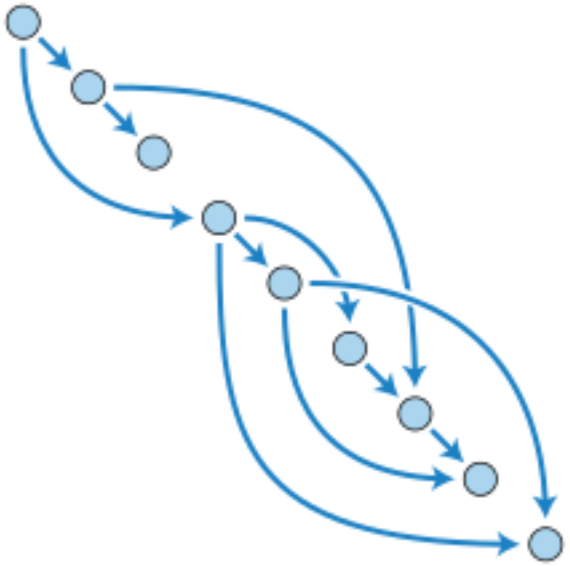
- Modules must be differentiable to support gradient computations for gradient descent

A **training algorithm** will then process this graph, **one module at a time**

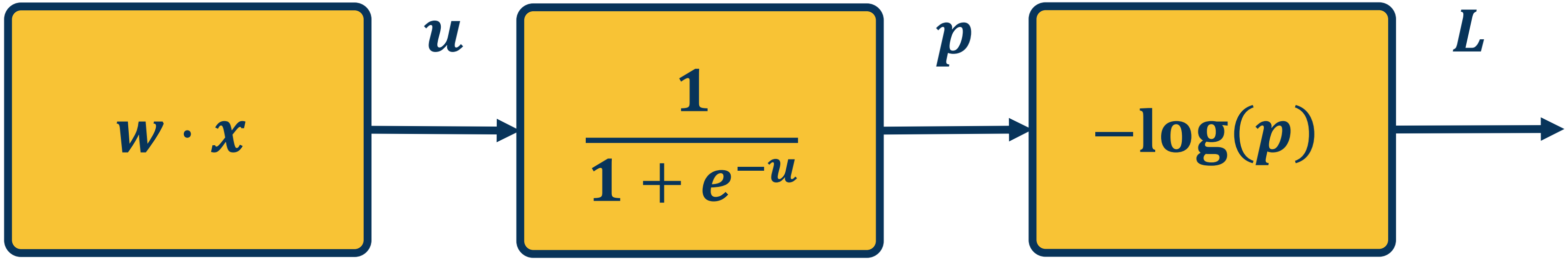


*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

# Directed Acyclic Graphs (DAGs)



$$-\log\left(\frac{1}{1+e^{-w\cdot x}}\right)$$



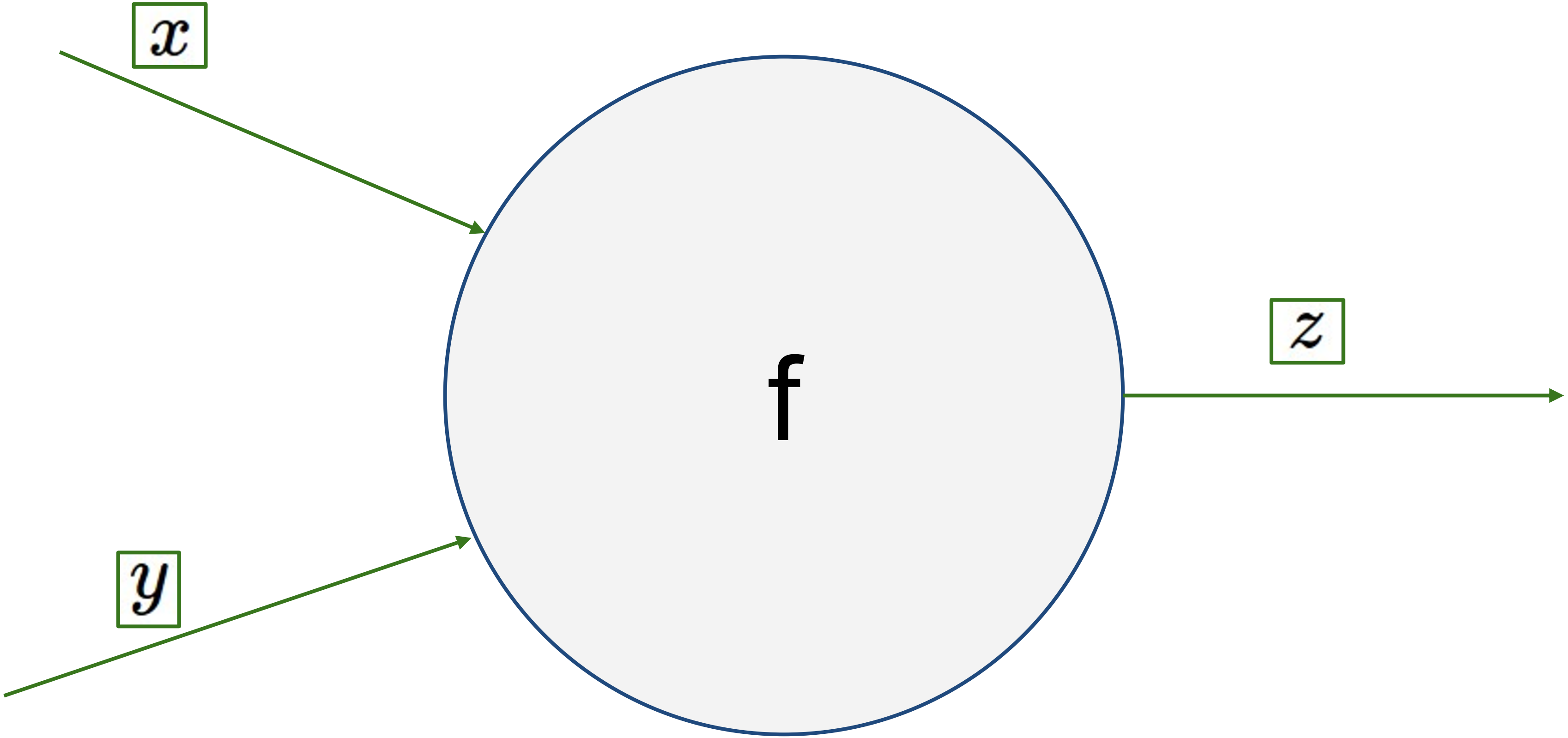

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*


$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w}$$

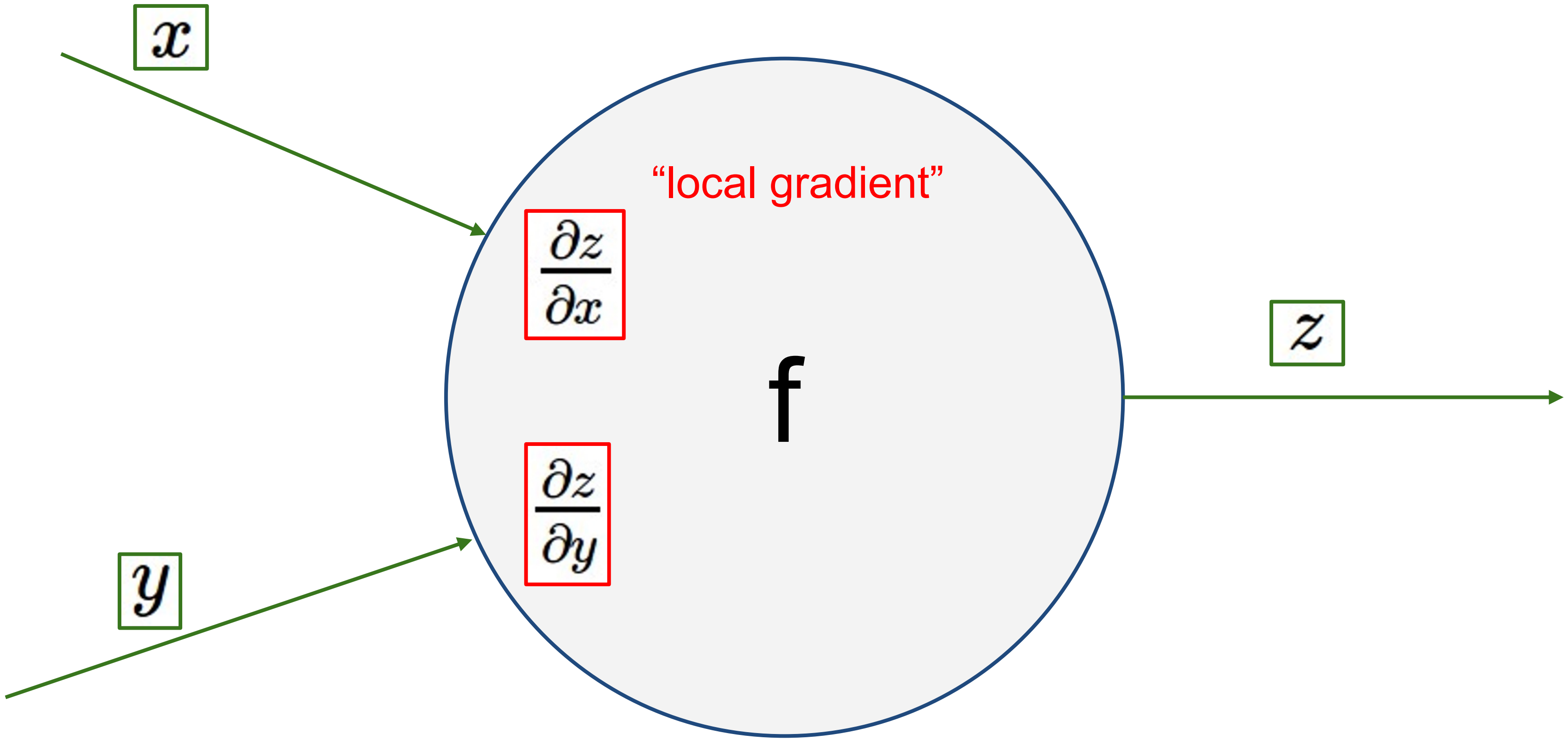
# This time: Chain rule and Backpropagation!

*Adapted from slides by: Marc'Aurelio Ranzato, Yann LeCun*

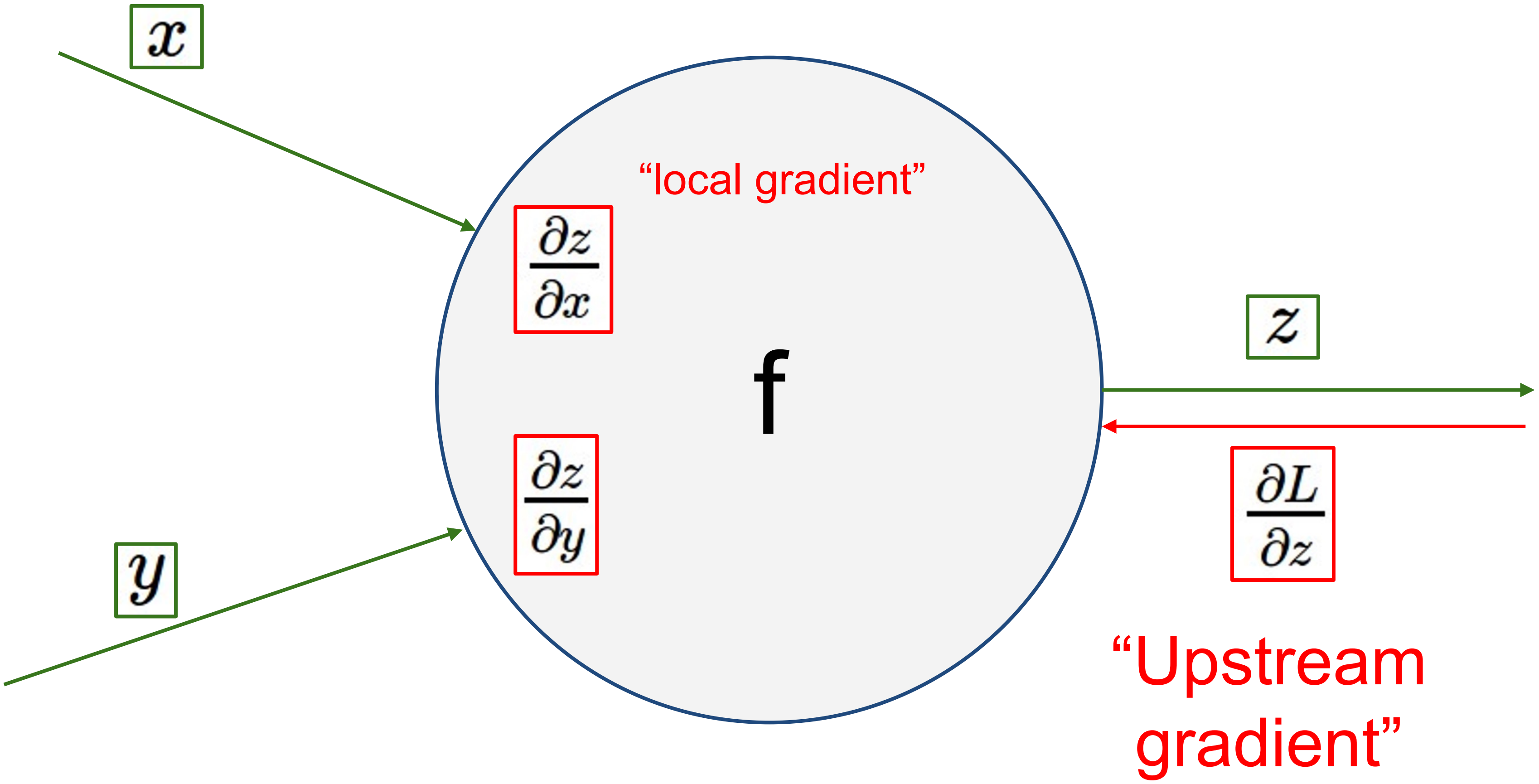
# A computation node



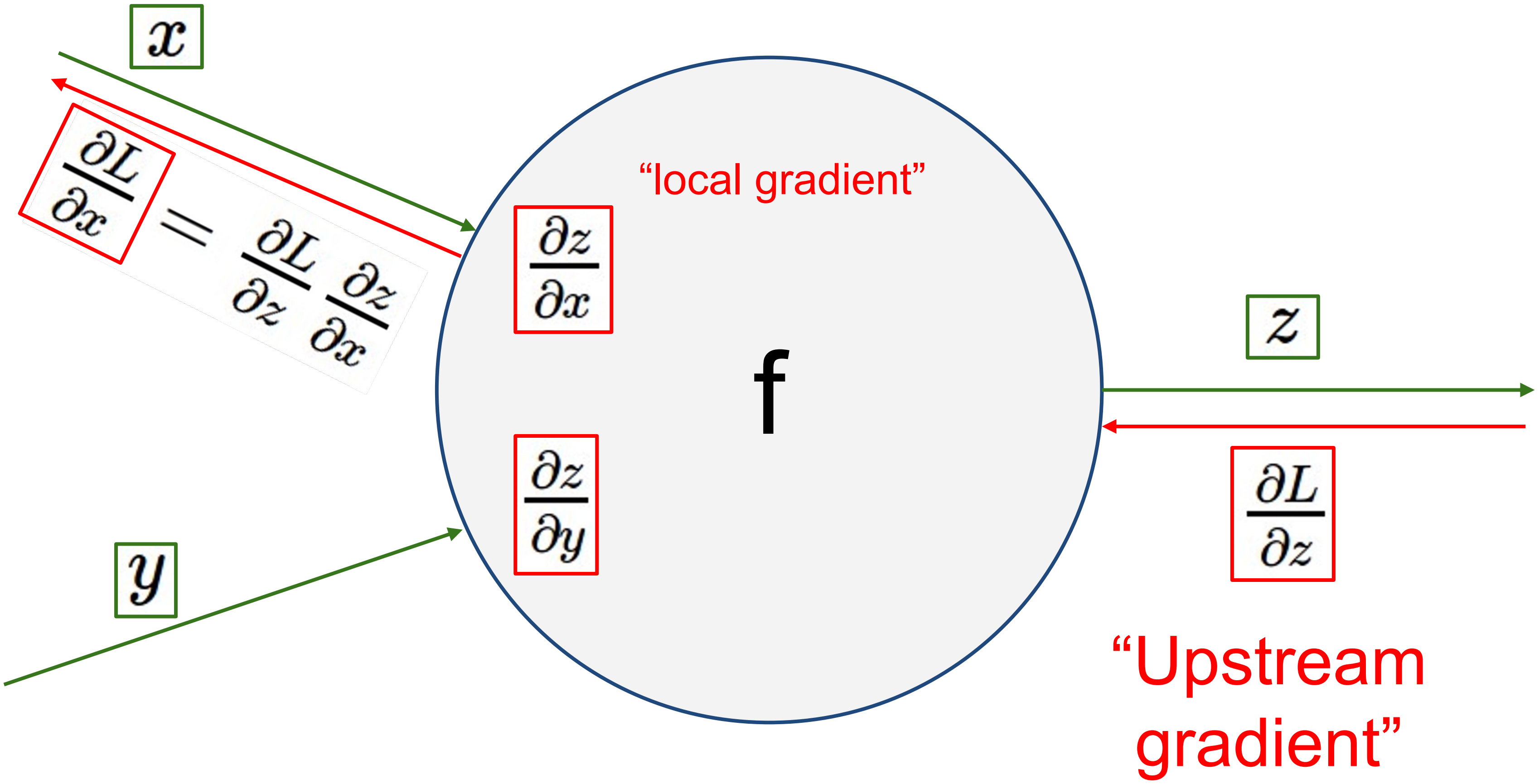

Slide credit: Stanford CS231n Instructors



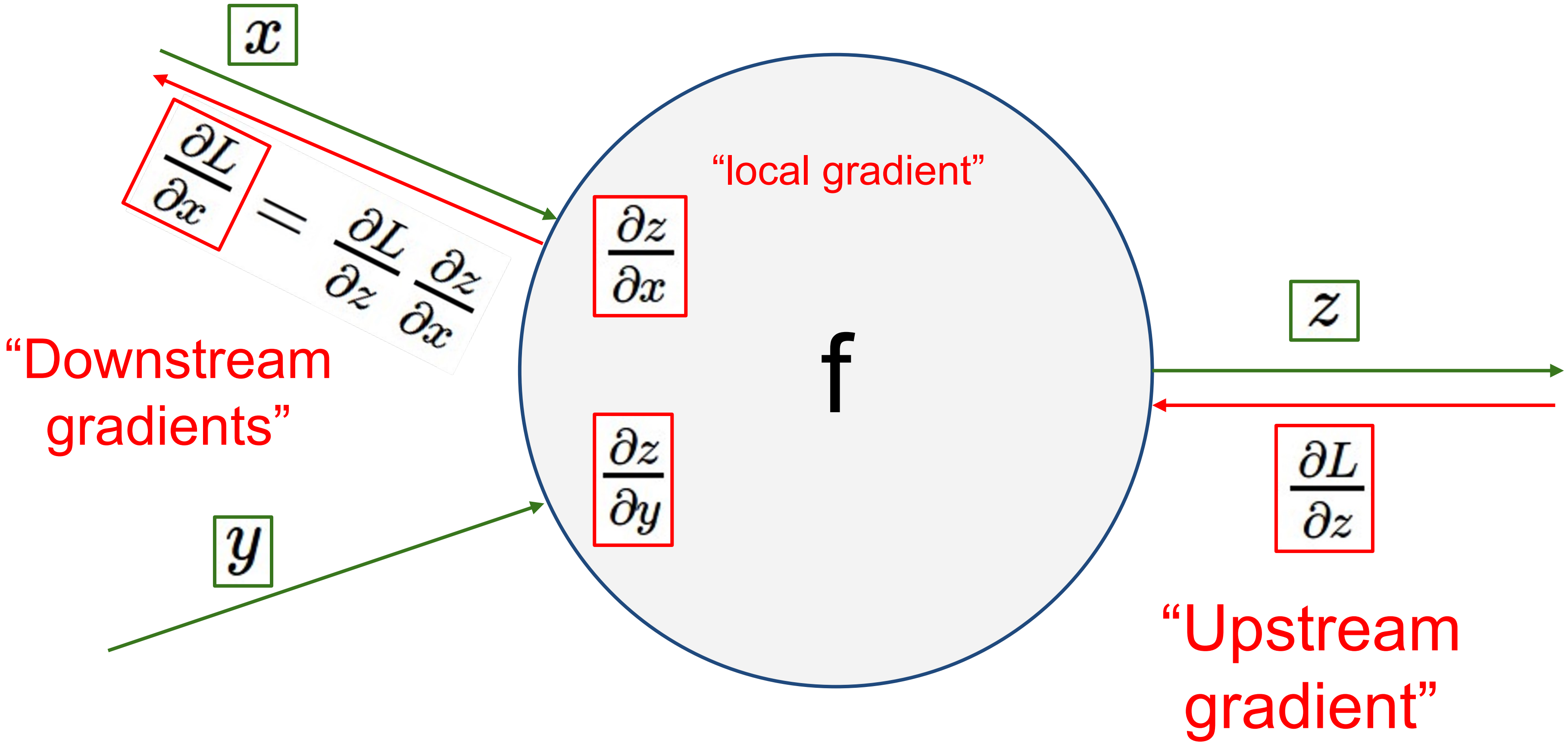

*Slide credit: Stanford CS231n Instructors*



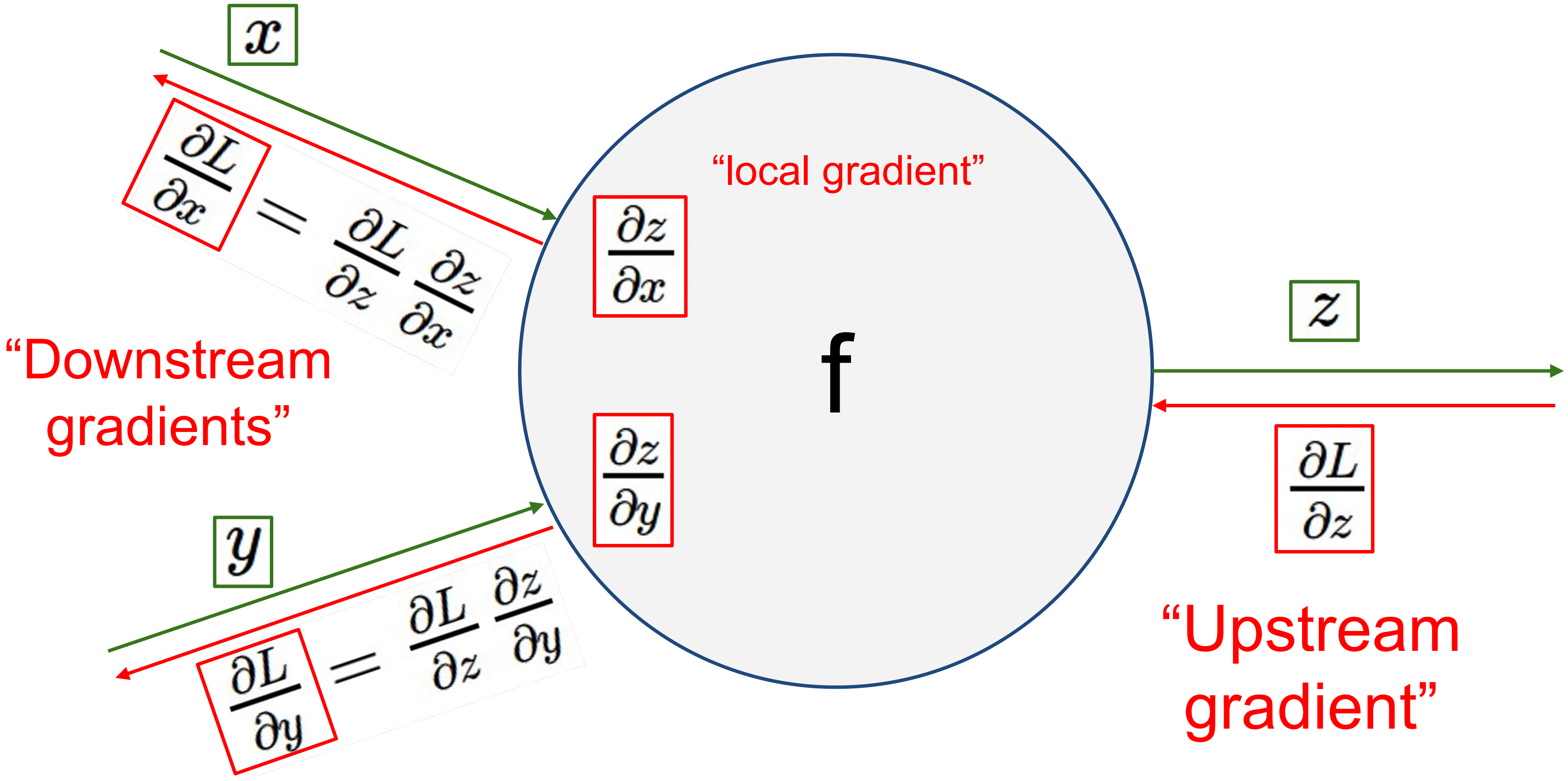

*Slide credit: Stanford CS231n Instructors*



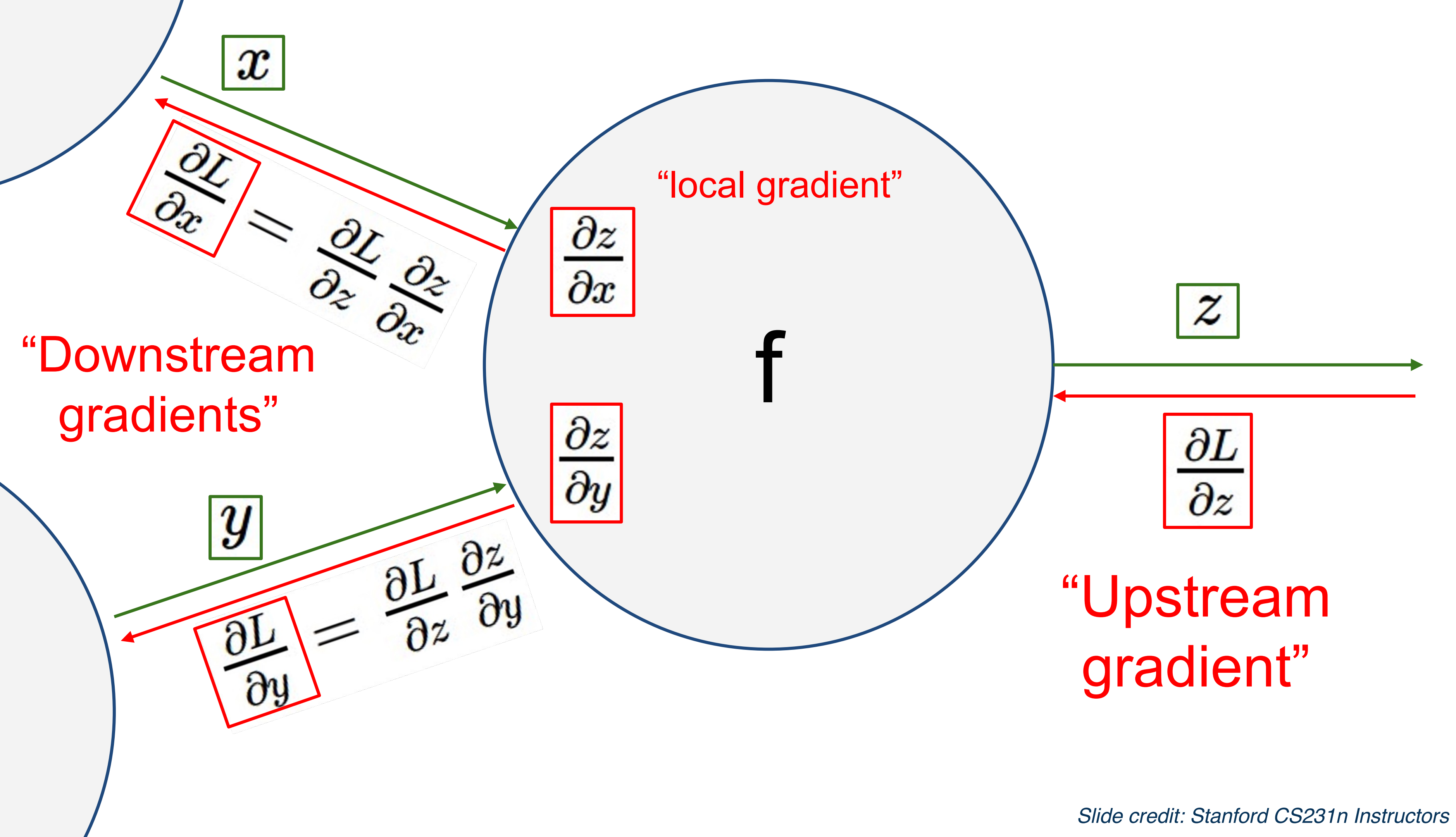

Slide credit: Stanford CS231n Instructors




Slide credit: Stanford CS231n Instructors




*Slide credit: Stanford CS231n Instructors*


$x$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"Downstream gradients"
$y$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
$z$
$\frac{\partial L}{\partial z}$
"Upstream gradient"

# Backpropagation: a simple example

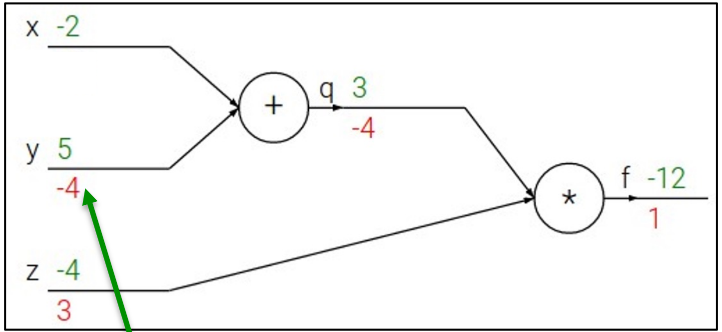
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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




Chain rule:

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

Upstream gradient

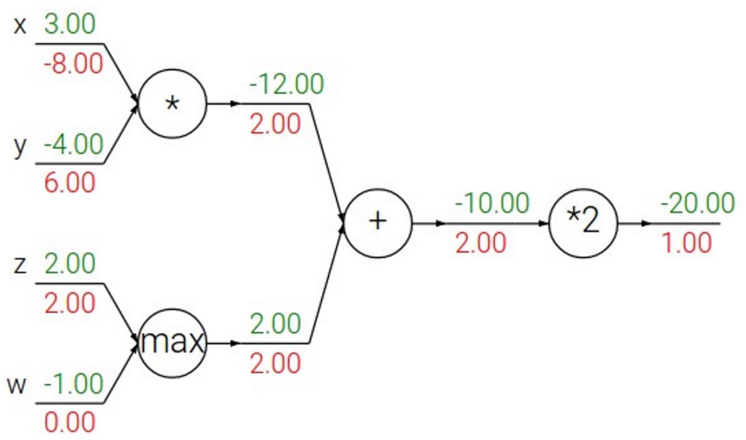
Local gradient

# Patterns in backward flow

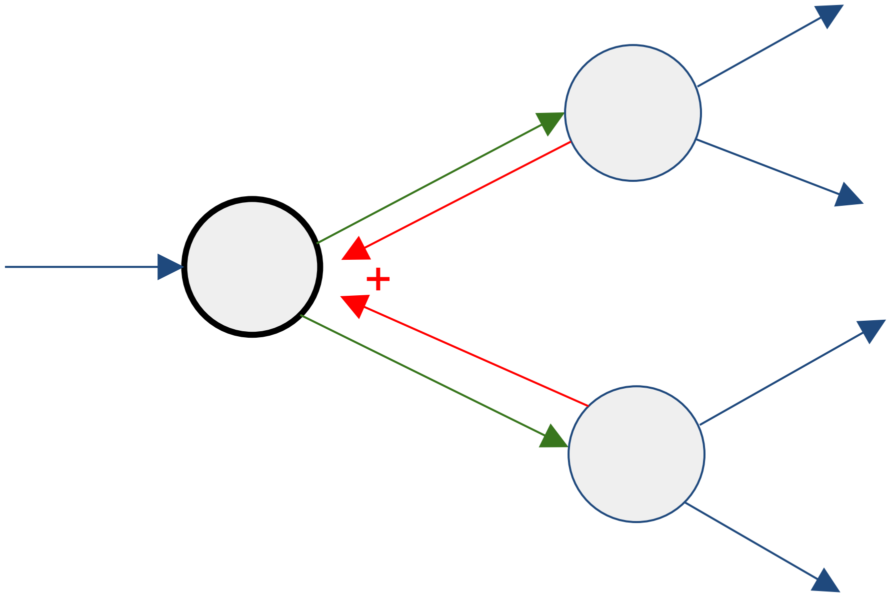
**add** gate: gradient distributor

**max** gate: gradient router

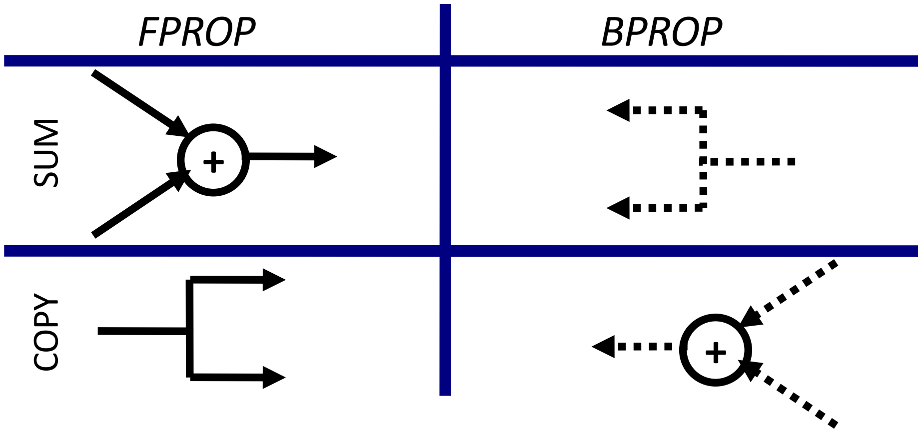
**mul** gate: gradient switcher



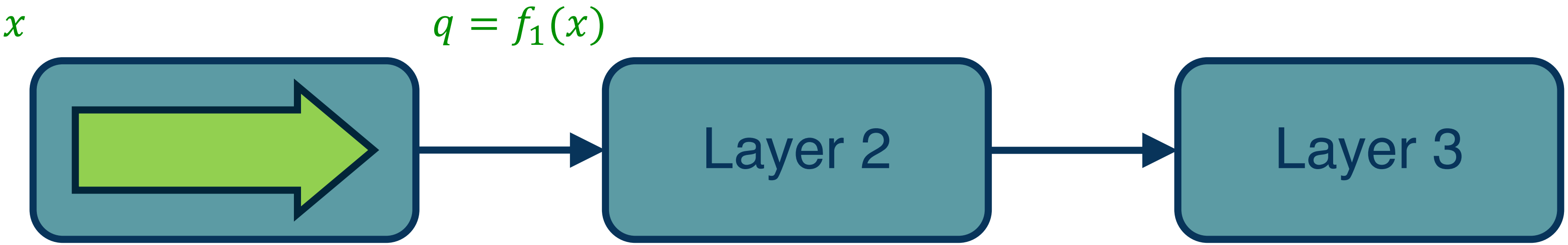
# Gradients add at branches



# Duality in Fprop and Bprop

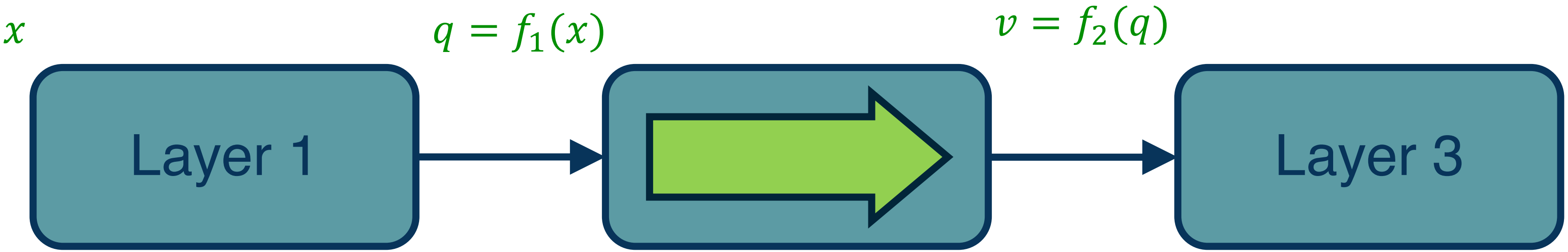


## **Step 1:** Compute Loss on Mini-Batch: **Forward Pass**



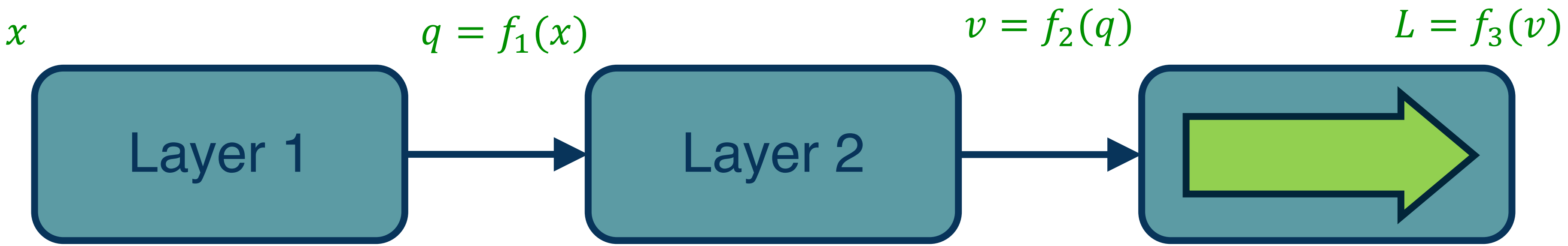

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

## **Step 1:** Compute Loss on Mini-Batch: **Forward Pass**




*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

# **Step 1:** Compute Loss on Mini-Batch: **Forward Pass**



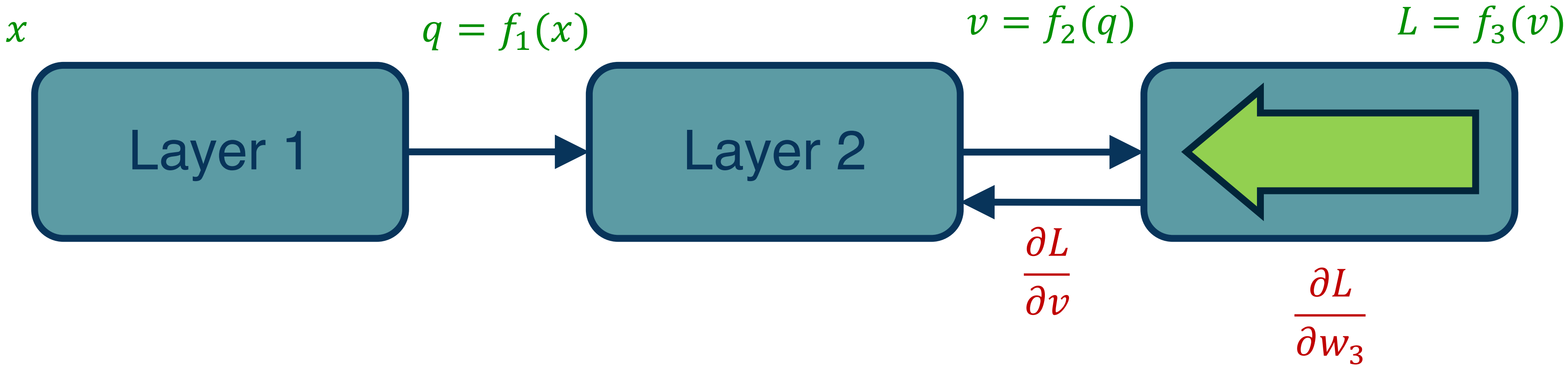

Note that we must store the **intermediate outputs of all layers**!

- This is because we will need them to **compute the gradients** (the gradient equations will have terms with the output values in them)

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

**Step 1:** Compute Loss on Mini-Batch: **Forward Pass**

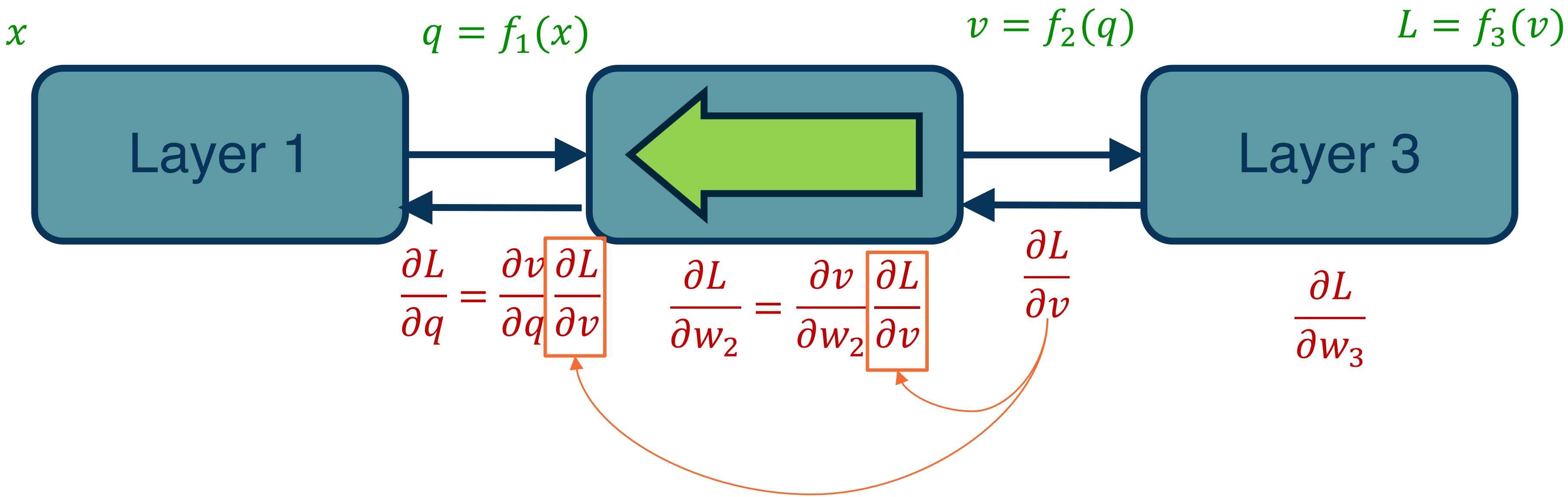
**Step 2:** Compute Gradients wrt parameters: **Backward Pass**




*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

**Step 1:** Compute Loss on Mini-Batch: **Forward Pass**

**Step 2:** Compute Gradients wrt parameters: **Backward Pass**

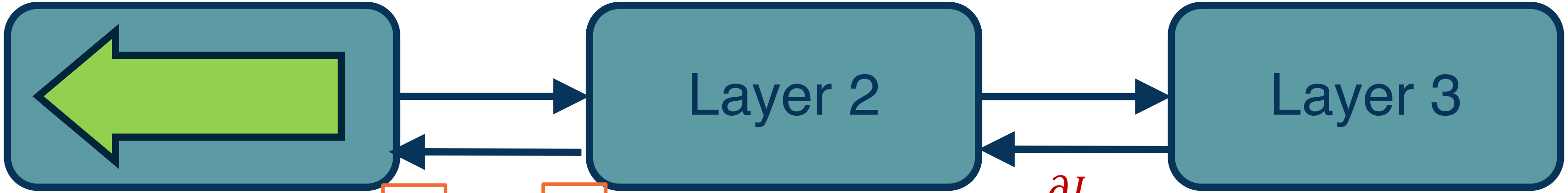



*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

**Step 1:** Compute Loss on Mini-Batch: **Forward Pass**

**Step 2:** Compute Gradients wrt parameters: **Backward Pass**

$x$ $\qquad q = f_1(x) \qquad v = f_2(q) \qquad L = f_3(v)$



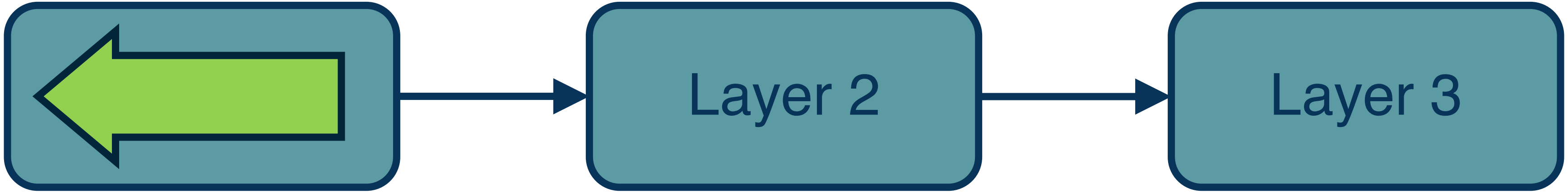

$$\frac{\partial L}{\partial w_3} = \frac{\partial v}{\partial w_3}\boxed{\frac{\partial L}{\partial q}} \qquad \boxed{\frac{\partial L}{\partial q}} = \frac{\partial v}{\partial q}\boxed{\frac{\partial L}{\partial v}} \qquad \frac{\partial L}{\partial w_2} = \frac{\partial v}{\partial w_2}\boxed{\frac{\partial L}{\partial v}} \qquad \frac{\partial L}{\partial v} \qquad \frac{\partial L}{\partial w_3}$$

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

**Step 1:** Compute Loss on Mini-Batch: **Forward Pass**

**Step 2:** Compute Gradients wrt parameters: **Backward Pass**

**Step 3:** Use **gradient** to update **all parameters** at the end




$\frac{\partial L}{\partial w_1}$ $\frac{\partial L}{\partial w_2}$ $\frac{\partial L}{\partial w_3}$

$$\boldsymbol{w_i = w_i - \alpha \frac{\partial L}{\partial w_i}}$$

Gradient Descent!

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

**So far:**

- **Linear classifiers**: a basic model
- **Loss functions**: measures performance of a model
- **Backpropagation**: an algorithm to calculate gradients of loss w.r.t. arbitrary differentiable function
- **Gradient Descent**: an iterative algorithm to perform gradient-based optimization

**Next:**

- What are neural networks?
- How do we run backpropagation on neural nets?

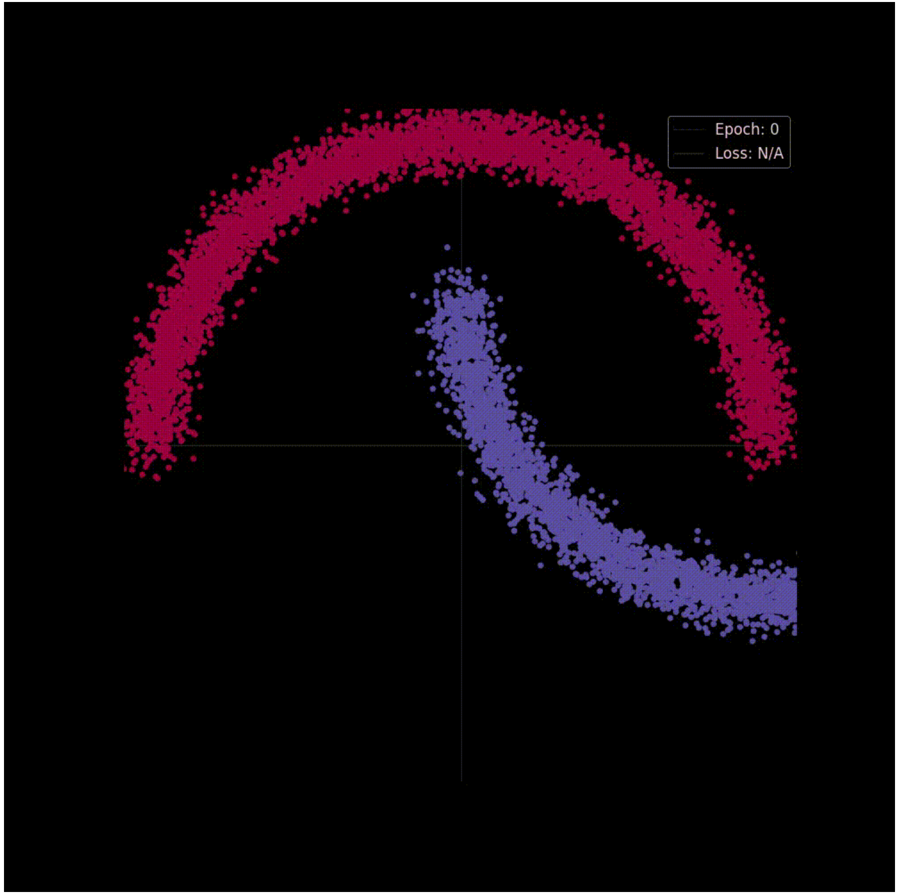
Neural Network

Linear classifiers

# Deep Representation Learning

Want: a function that transforms complex raw data space into a linearly-separable space.

The function needs to be non-linear!


https://khalidsaifullaah.github.io/neural-networks-from-linear-algebraic-perspective

# Neural networks: the original linear classifier

(**Before**) Linear score function: $f = Wx$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

*Slide credit: Stanford CS231n Instructors*

# Neural networks: 2 layers

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

(In practice we will usually add a learnable bias at each layer as well)

# Neural networks: also called fully connected network

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

"Neural Network" is a very broad term; these are more accurately called "fully-connected networks" or sometimes "multi-layer perceptrons" (MLP)

(In practice we will usually add a learnable bias at each layer as well)

*Slide credit: Stanford CS231n Instructors*

# Neural networks: 3 layers

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$
or 3-layer Neural Network

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

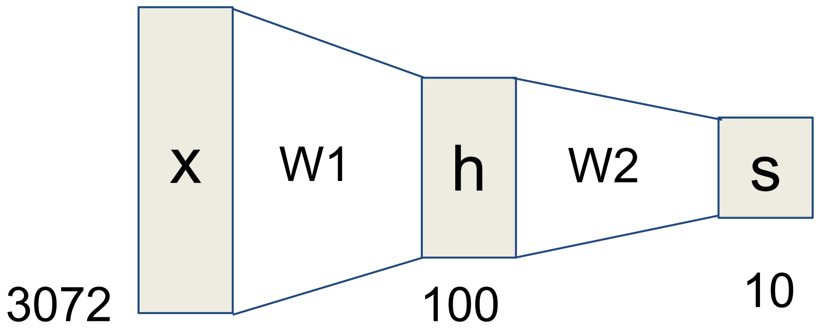
(In practice we will usually add a learnable bias at each layer as well)

*Slide credit: Stanford CS231n Instructors*

# Neural networks: hierarchical computation

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$




$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

*Slide credit: Stanford CS231n Instructors*

# Neural networks: why is max operator important?

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \boxed{\max(0,} W_1 x)$

The function $\max(0, z)$ is called the **activation function.**

**Q:** What if we try to build a neural network without one?

$$f = W_2 W_1 x$$

*Slide credit: Stanford CS231n Instructors*

# Neural networks: why is max operator important?

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \boxed{\max(0,} W_1 x)$

The function $\max(0, z)$ is called the **activation function.**
**Q:** What if we try to build a neural network without one?

$$f = W_2 W_1 x \qquad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

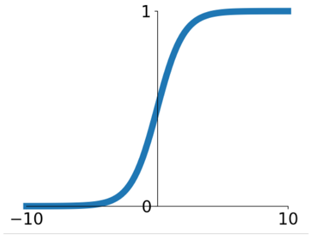
**A**: We end up with a linear classifier again!
(Non-linear) activation function allows us to build non-linear functions / neural networks

*Slide credit: Stanford CS231n Instructors*

# Activation functions

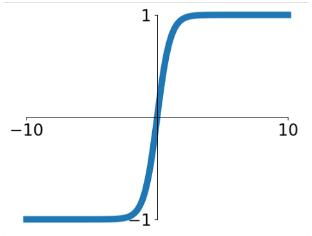
**Sigmoid**

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



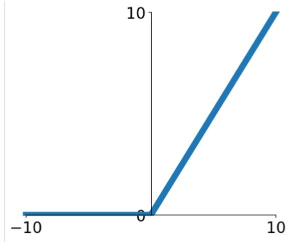

**tanh**

$\tanh(x)$



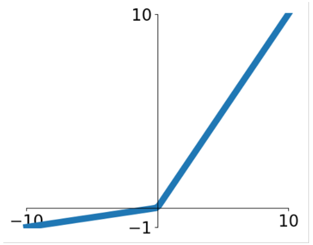

**ReLU**

$\max(0, x)$




**Leaky ReLU**

$\max(0.1x, x)$

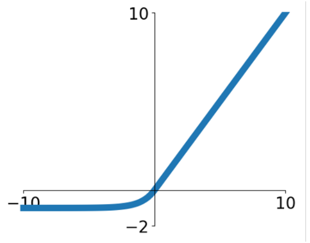



**Maxout**

$\max(w_1^T x + b_1, w_2^T x + b_2)$

**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



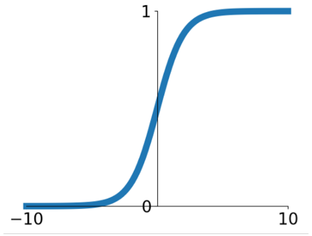

*Slide credit: Stanford CS231n Instructors*

# Activation functions

ReLU is a good default choice for most problems

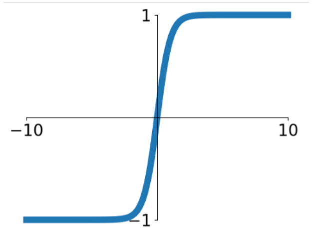
**Sigmoid**

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



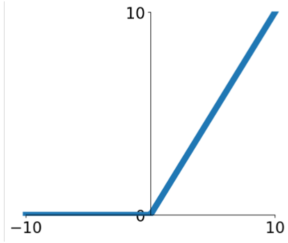

**tanh**

$$\tanh(x)$$



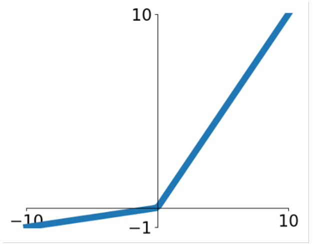

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

$$\max(0.1x, x)$$

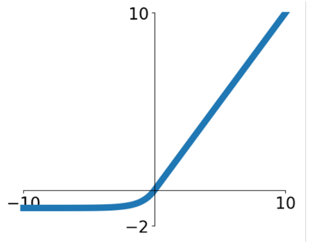



**Maxout**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

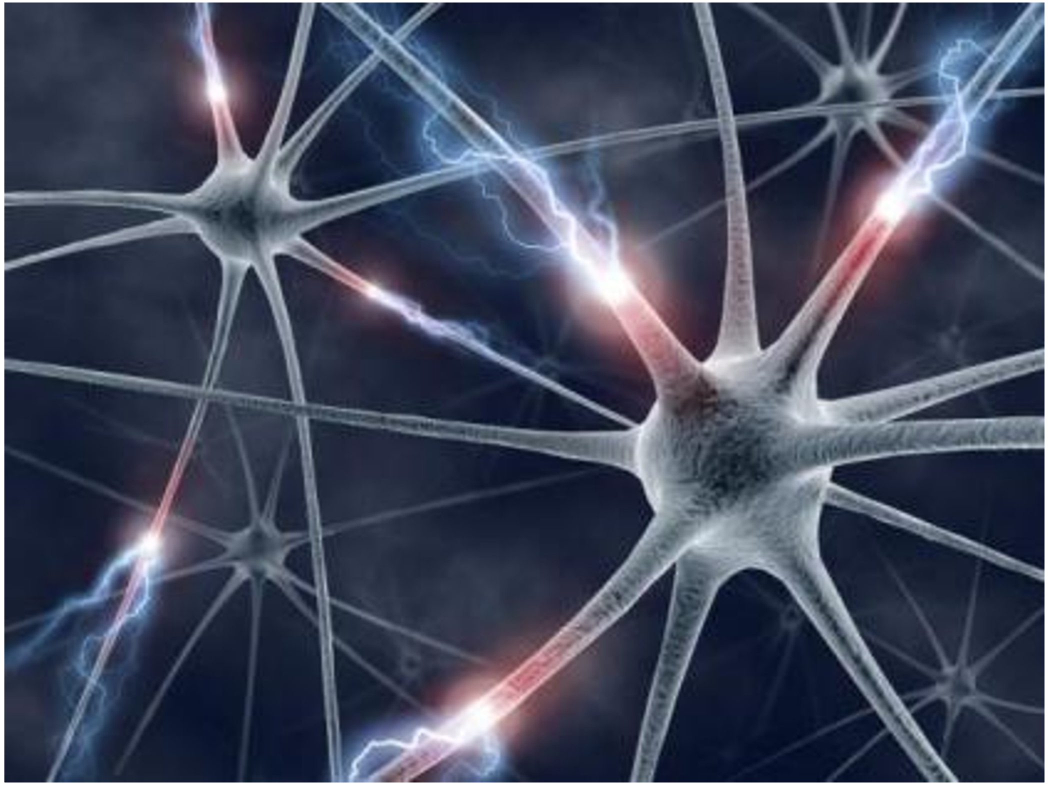
**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



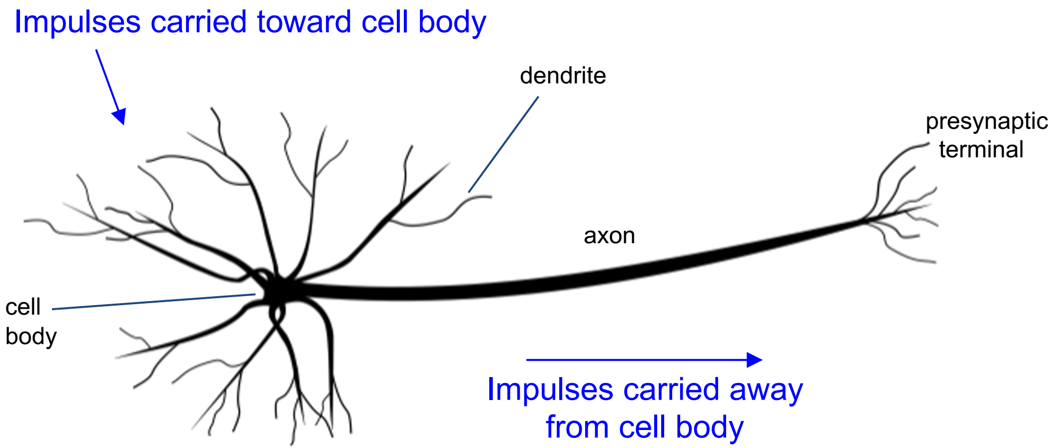

*Slide credit: Stanford CS231n Instructors*

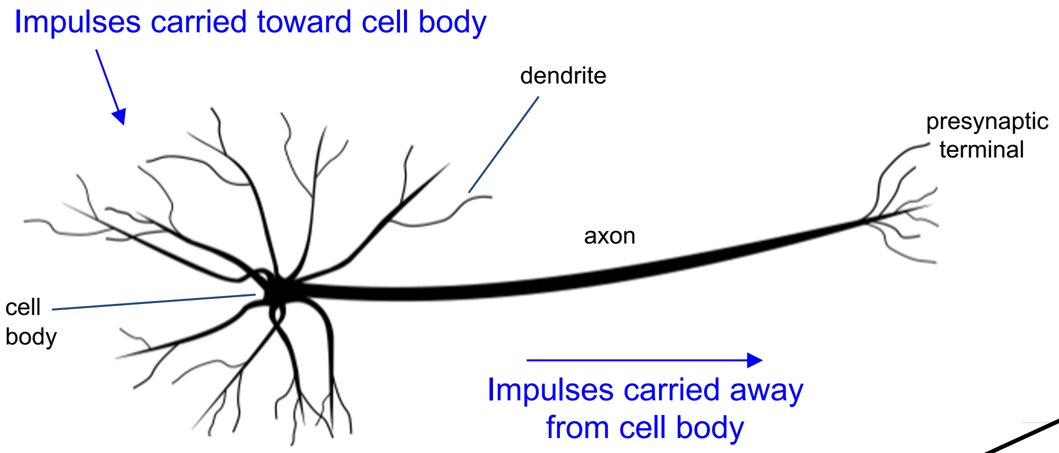
# Why are they called Neural Networks anyways?



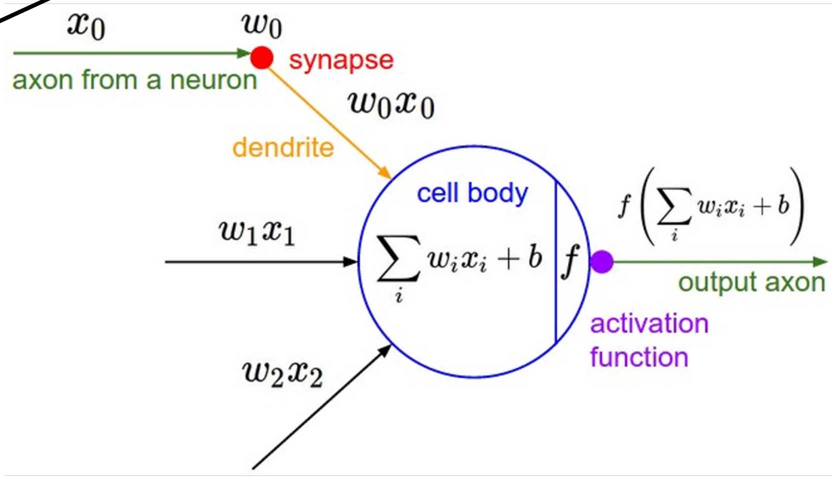
This image by Fotis Bobolas is licensed under CC-BY 2.0

Slide credit: Stanford CS231n Instructors

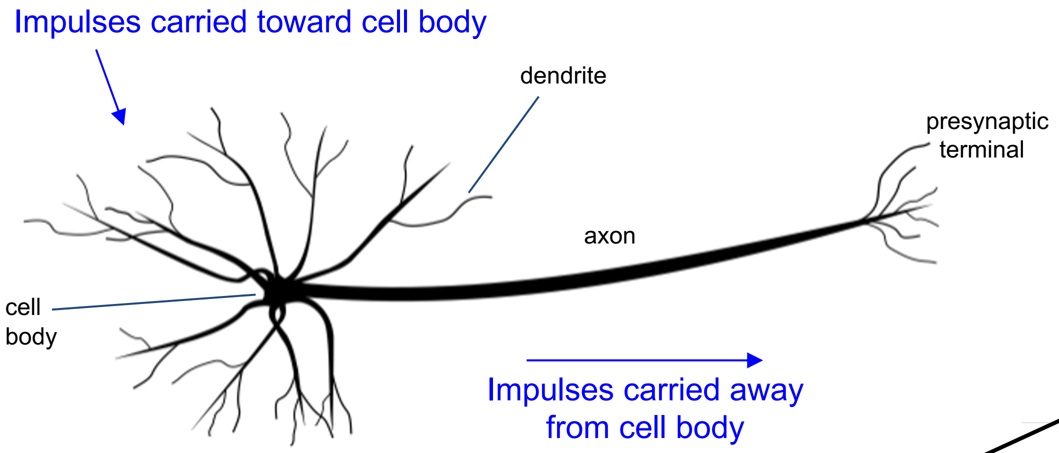

Impulses carried toward cell body
dendrite
presynaptic
terminal
axon
cell
body
Impulses carried away
from cell body



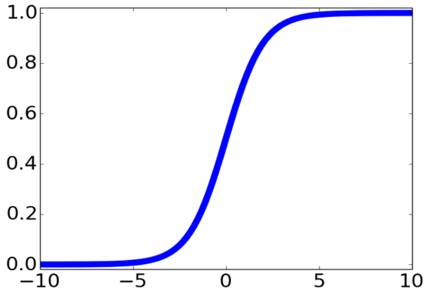



*Slide credit: Stanford CS231n Instructors*

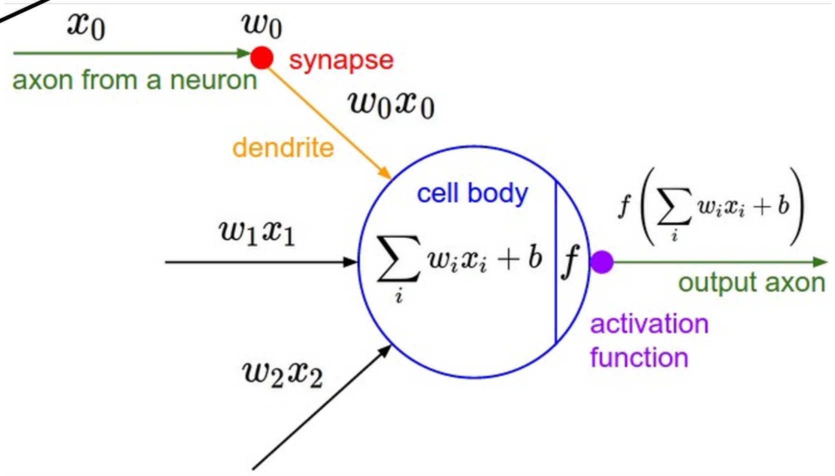




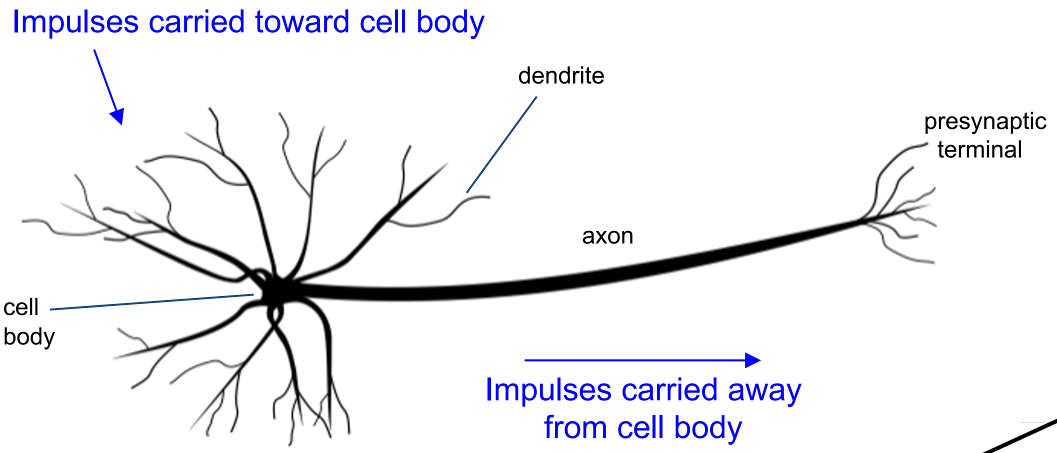




sigmoid activation function

$$\frac{1}{1+e^{-x}}$$

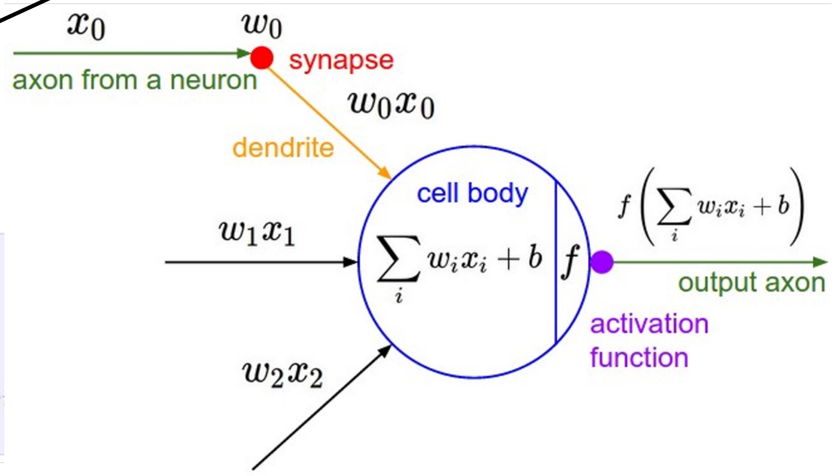



*Slide credit: Stanford CS231n Instructors*



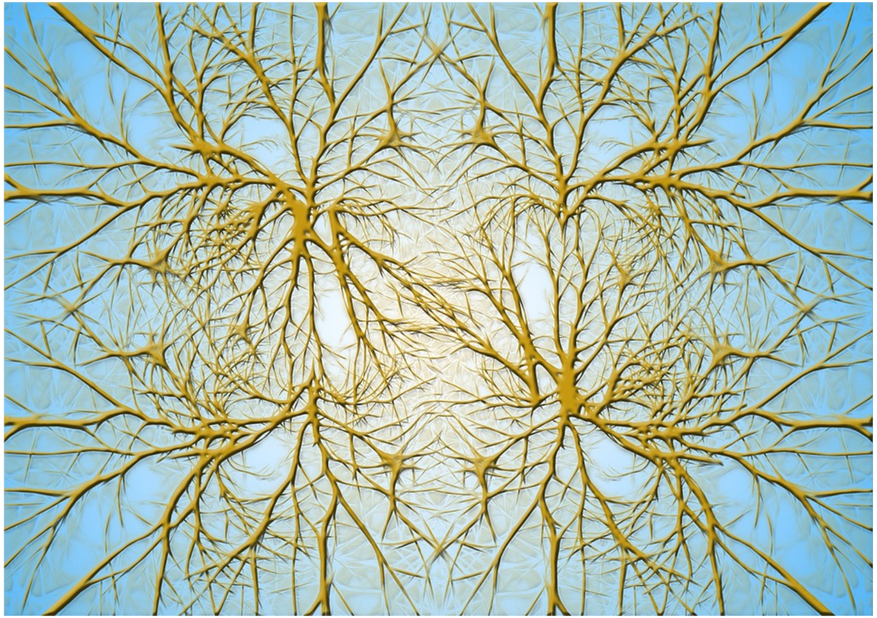



```python
class Neuron:
  # ...
  def neuron_tick(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation func
    return firing_rate
```



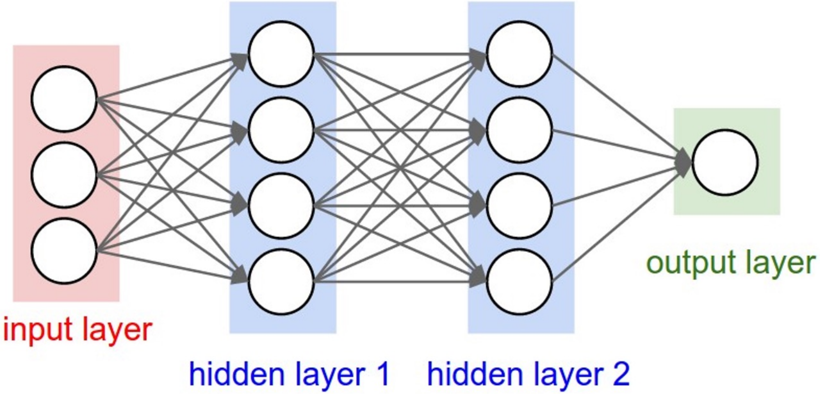

*Slide credit: Stanford CS231n Instructors*

Biological Neurons:
Complex connectivity patterns

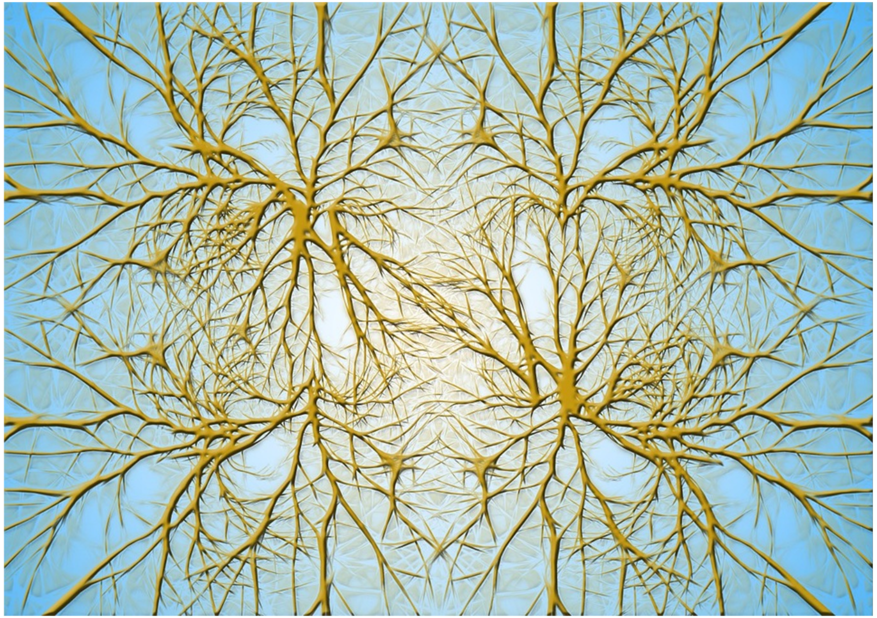




Neurons in a neural network:
Organized into regular layers for computational efficiency

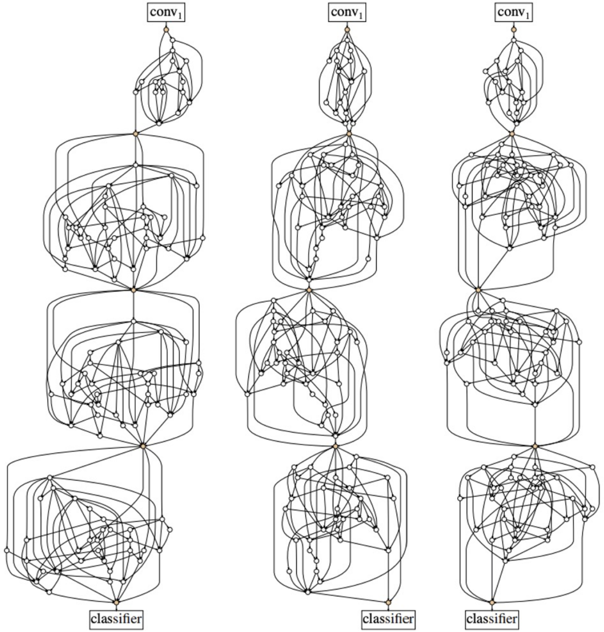


# Biological Neurons: Complex connectivity patterns



This image is CC0 Public Domain

# But neural networks with random connections can work too!




Xie et al, “Exploring Randomly Wired Neural Networks for Image Recognition”, arXiv 2019

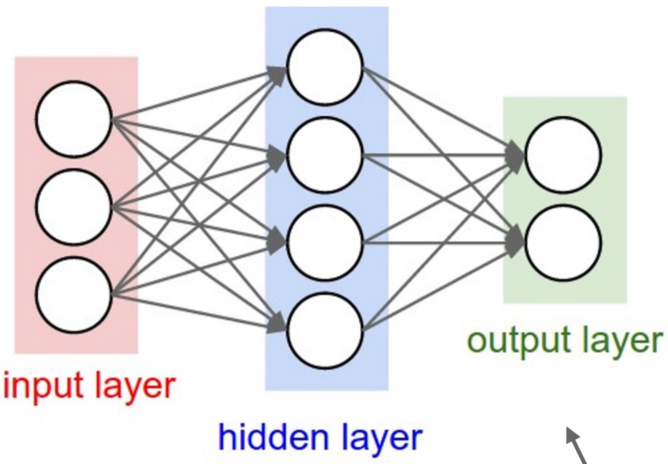
*Slide credit: Stanford CS231n Instructors*

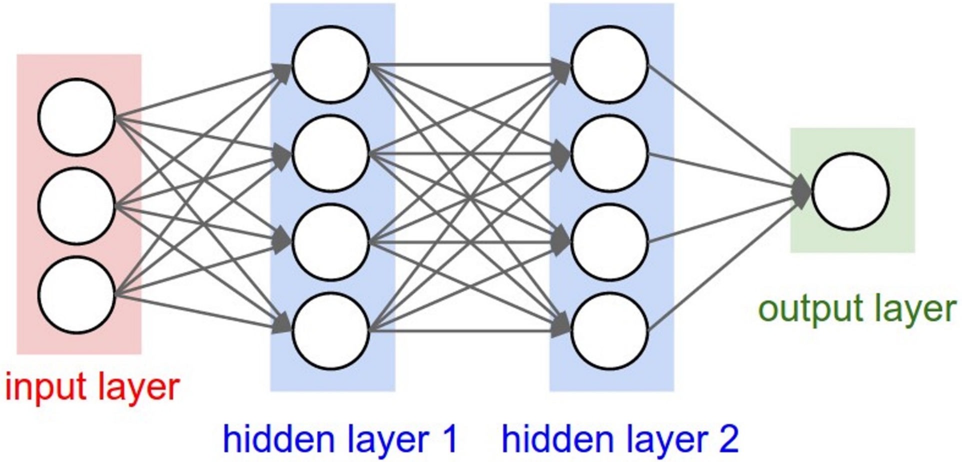
# Be very careful with your brain analogies!

**Biological Neurons:**
- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system

[Dendritic Computation. London and Hausser]

# Neural networks: Architectures



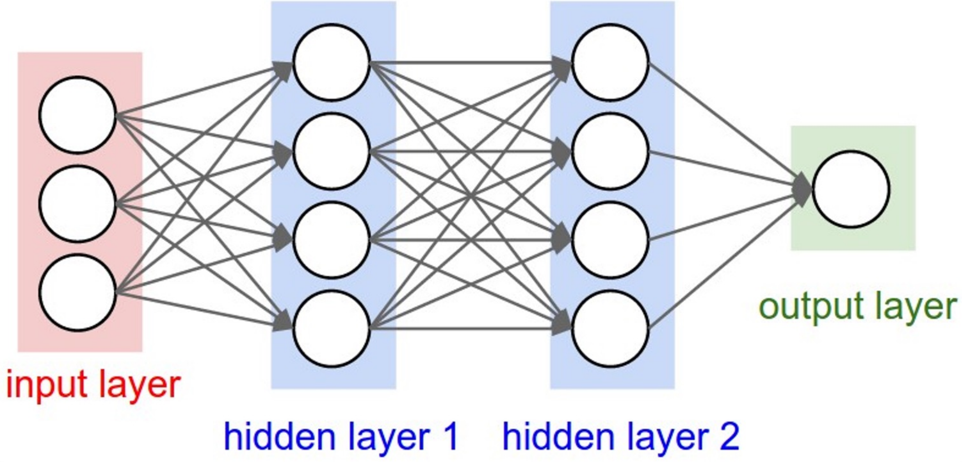

"2-layer Neural Net", or
"1-hidden-layer Neural Net"

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

**"Fully-connected" layers**

*Slide credit: Stanford CS231n Instructors*

# Example feed-forward computation of a neural network




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

*Slide credit: Stanford CS231n Instructors*

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

*Slide credit: Stanford CS231n Instructors*

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

Define the network

*Slide credit: Stanford CS231n Instructors*

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

*Slide credit: Stanford CS231n Instructors*

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

*Slide credit: Stanford CS231n Instructors*

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

Gradient descent

*Slide credit: Stanford CS231n Instructors*

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

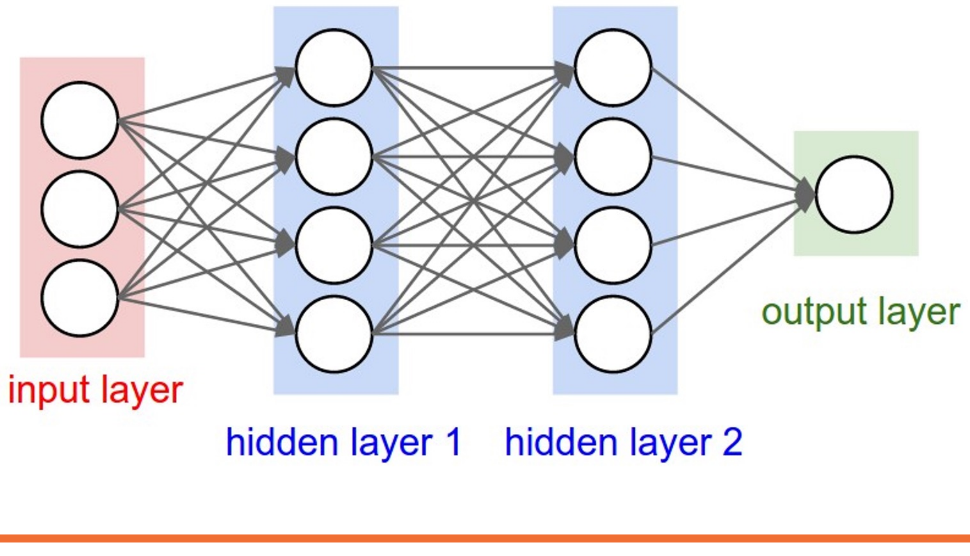
```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

matrix

Calculate the analytical gradients

How?

# Next: Vector Calculus!




How do we do backpropagation with neural nets?

Slide credit: Stanford CS231n Instructors

# Recap: Vector derivatives

## Scalar to Scalar

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

Regular derivative:

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

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

*Slide credit: Stanford CS231n Instructors*

# Recap: Vector derivatives

## Scalar to Scalar

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

Regular derivative:

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

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

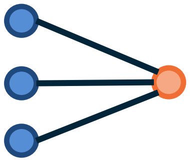
## Vector to Scalar

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

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x}\right)_n = \frac{\partial y}{\partial x_n}$$

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





*Slide credit: Stanford CS231n Instructors*

# Recap: Vector derivatives

## Scalar to Scalar

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

Regular derivative:

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

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

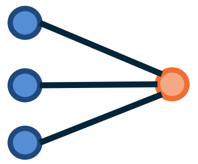
## Vector to Scalar

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

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x}\right)_n = \frac{\partial y}{\partial x_n}$$

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



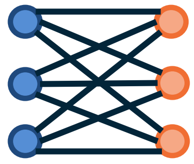
## Vector to Vector

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

Derivative is **Jacobian**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left(\frac{\partial y}{\partial x}\right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

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

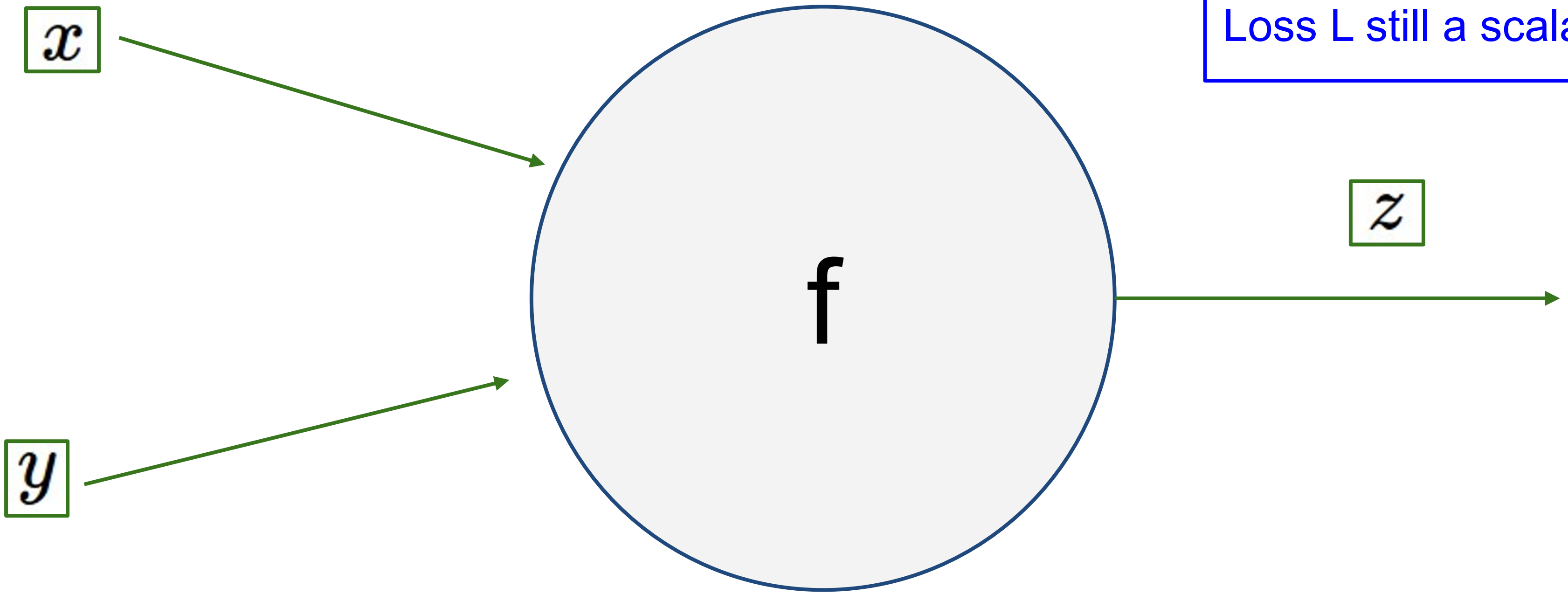




*Slide credit: Stanford CS231n Instructors*

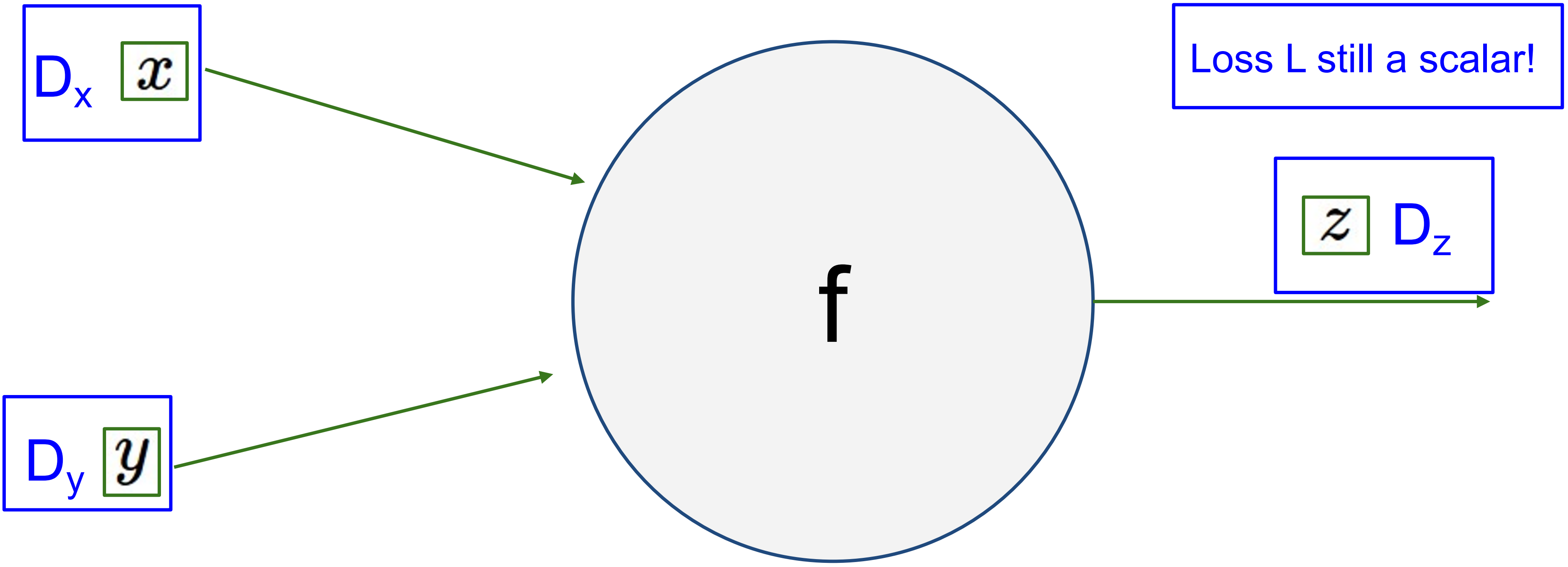
# Backprop with Vectors

Loss L still a scalar!



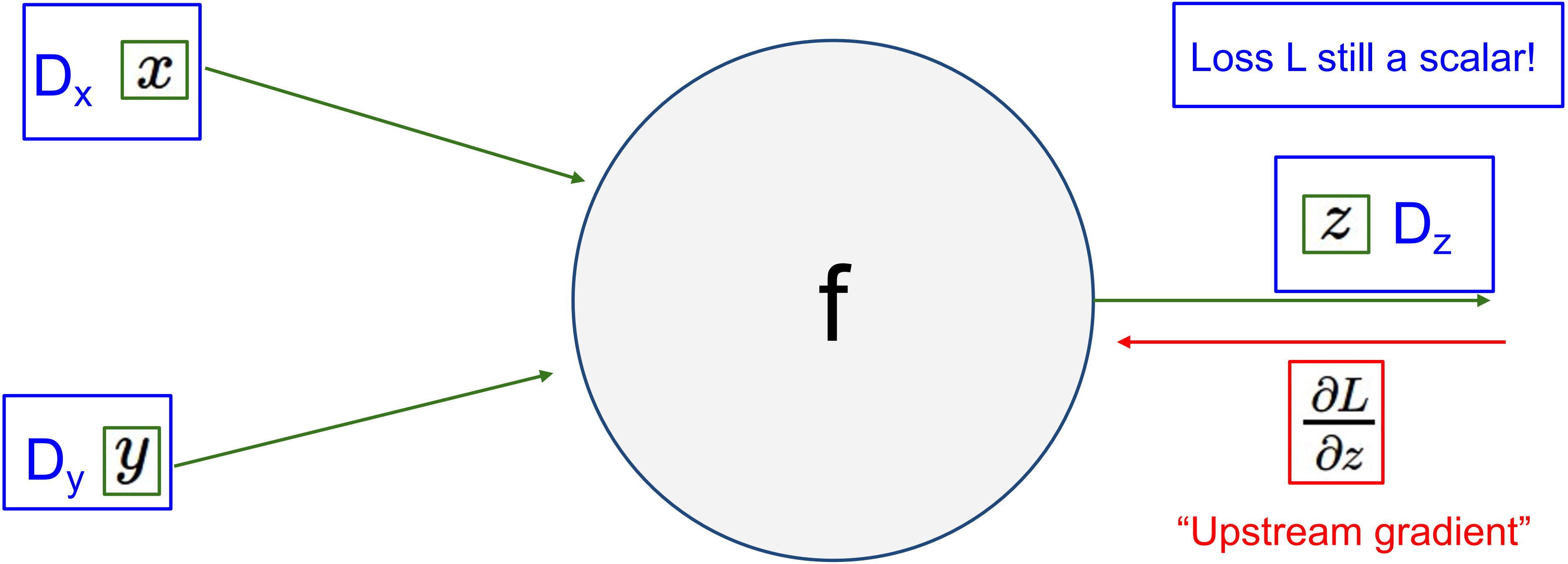

Slide credit: Stanford CS231n Instructors

# Backprop with Vectors




Slide credit: Stanford CS231n Instructors

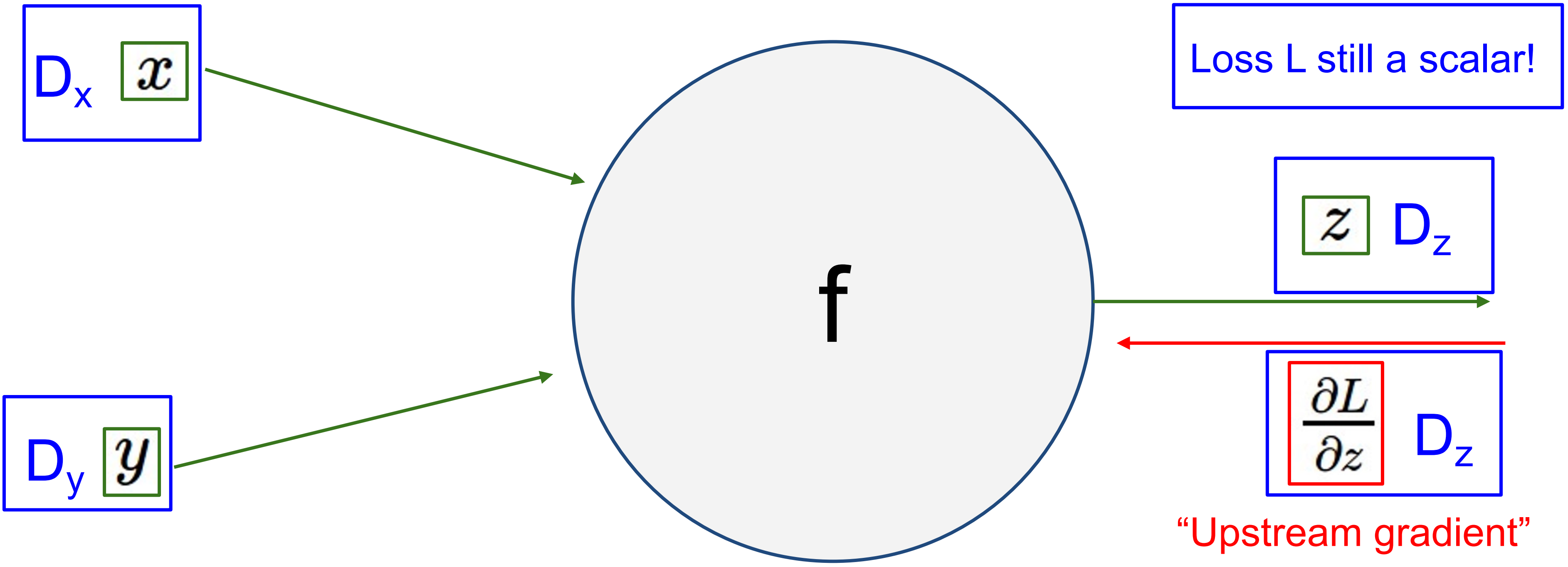
# Backprop with Vectors




What's the shape of $\frac{\partial L}{\partial z}$?

*Slide credit: Stanford CS231n Instructors*

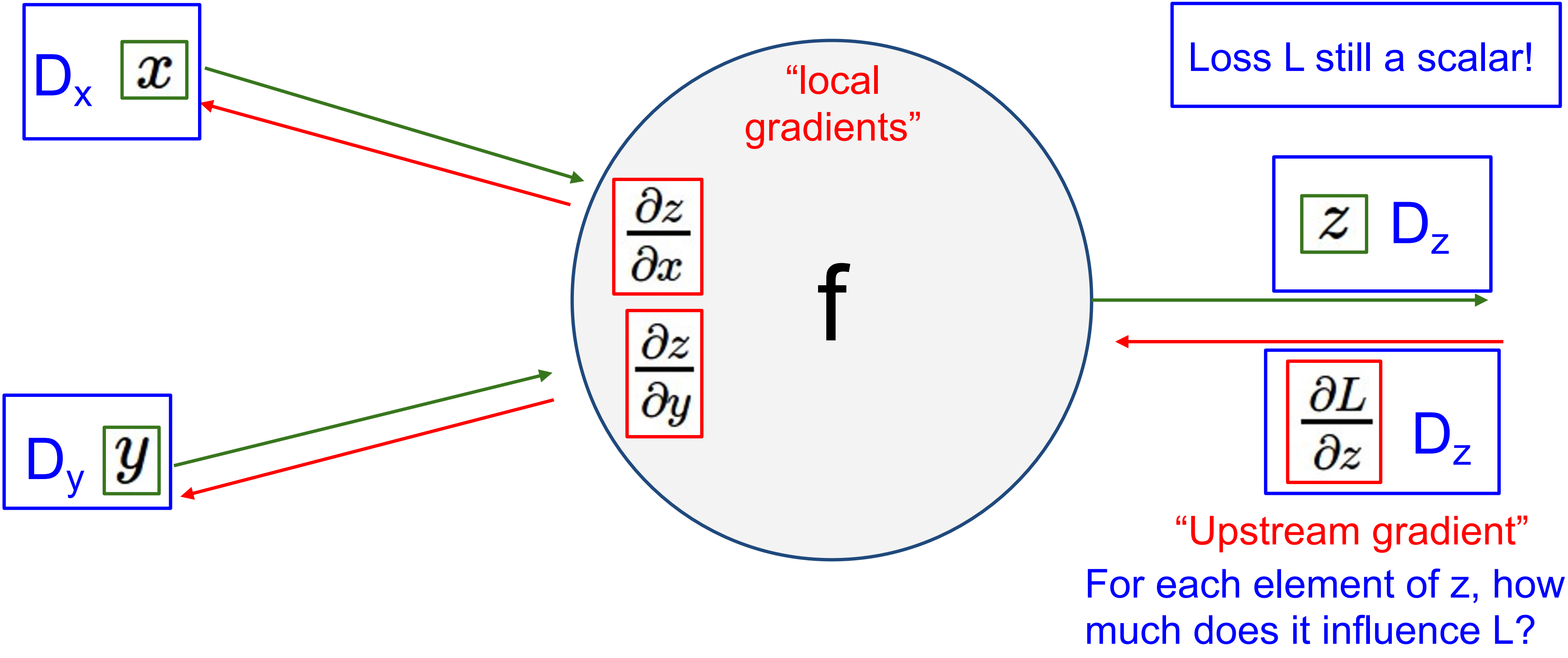
# Backprop with Vectors




"Upstream gradient"

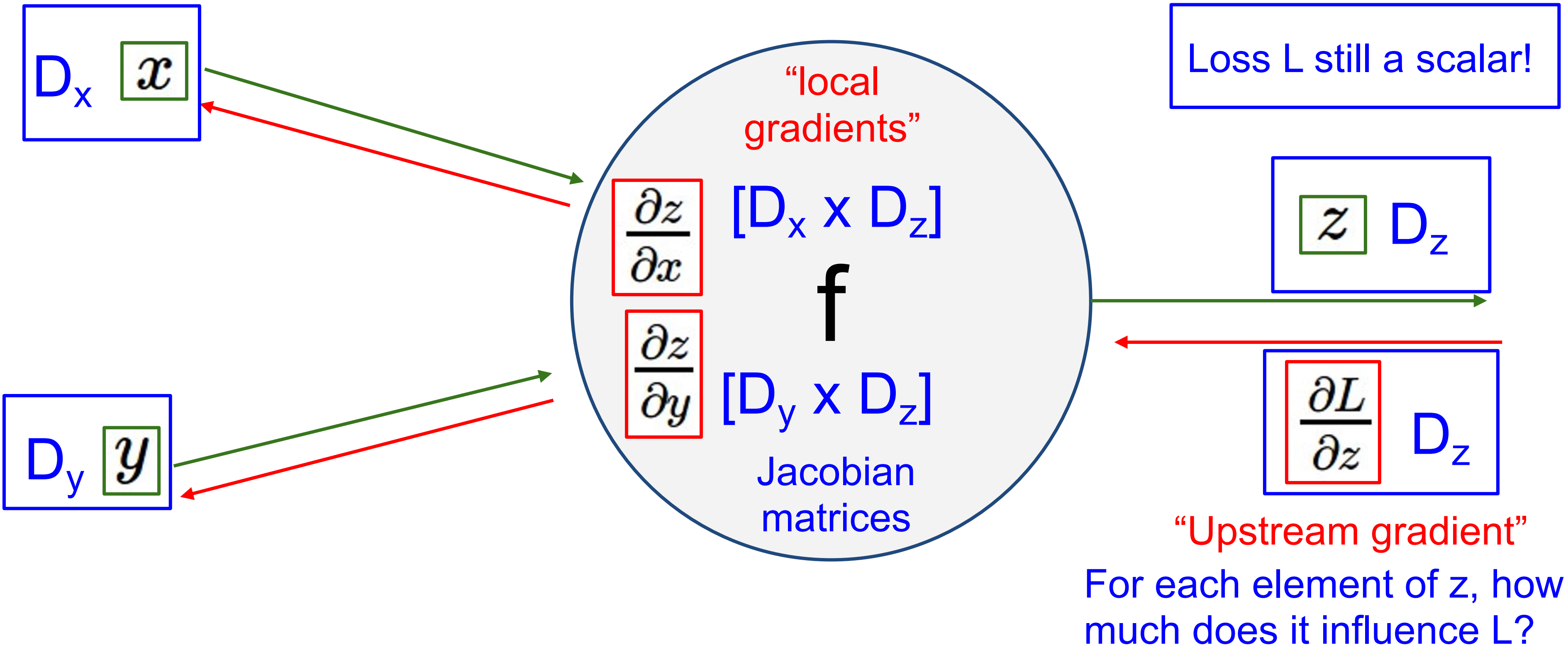
For each element of z, how much does it influence L?

# Backprop with Vectors



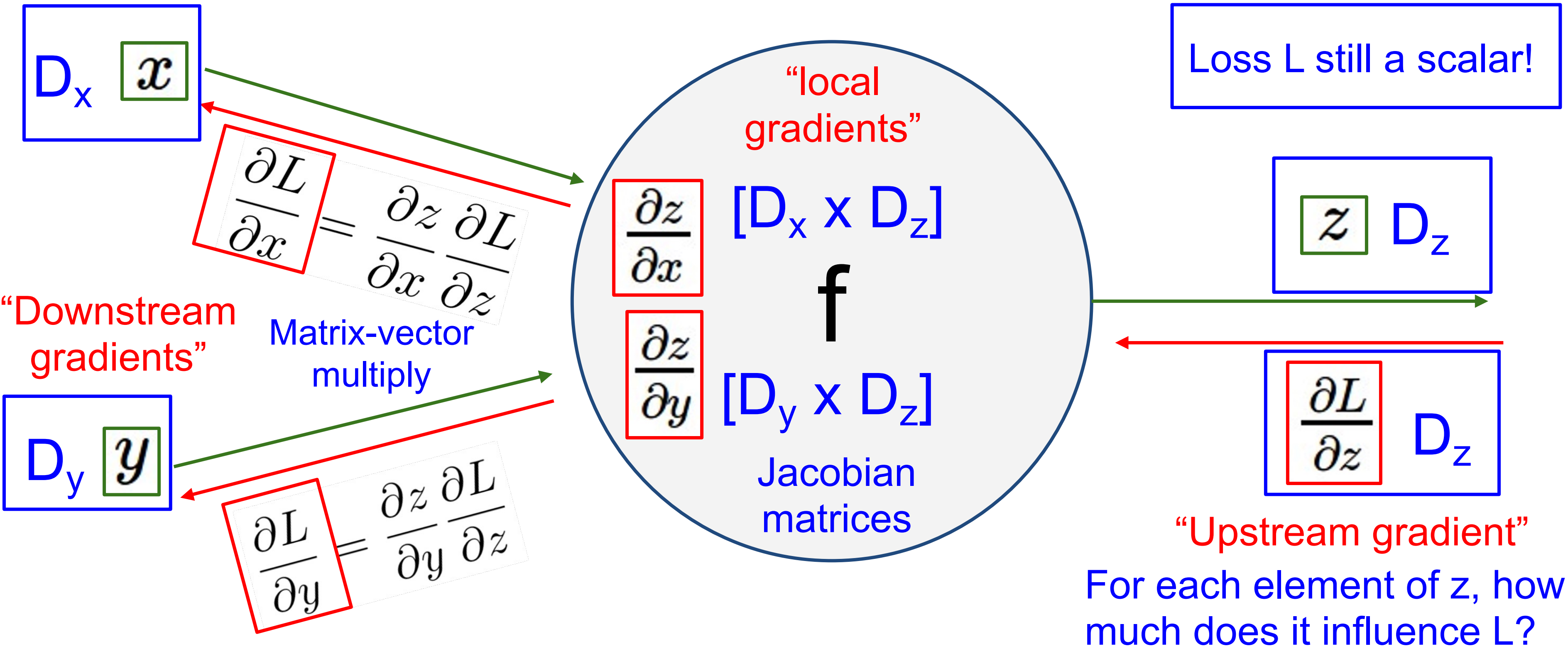

Slide credit: Stanford CS231n Instructors

# Backprop with Vectors



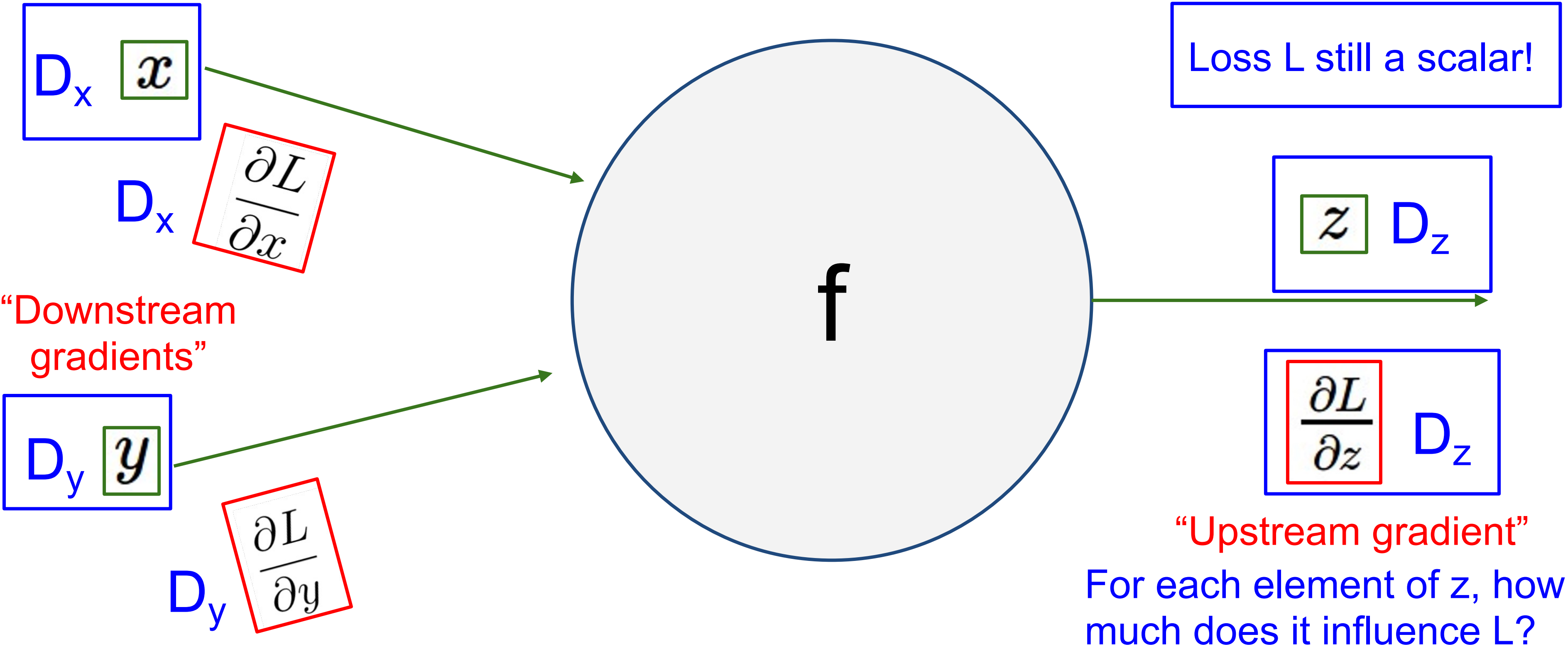

*Slide credit: Stanford CS231n Instructors*

# Backprop with Vectors



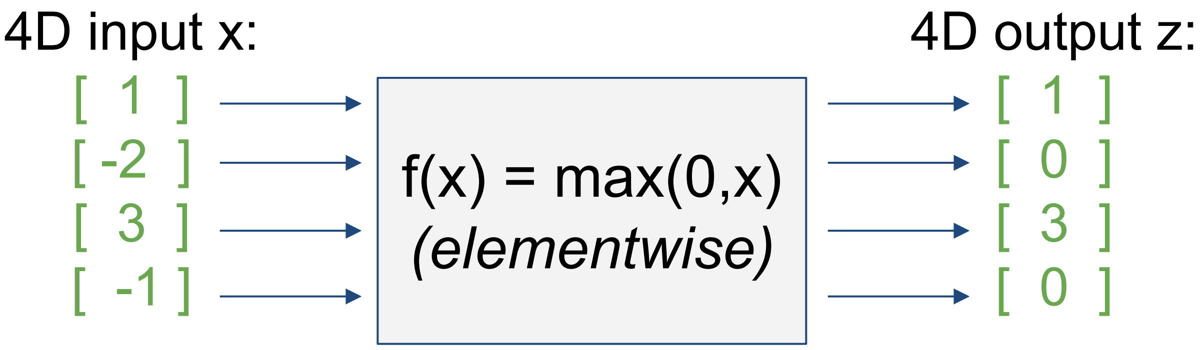

*Slide credit: Stanford CS231n Instructors*

# Gradients loss of wrt a variable have same dims as the original variable



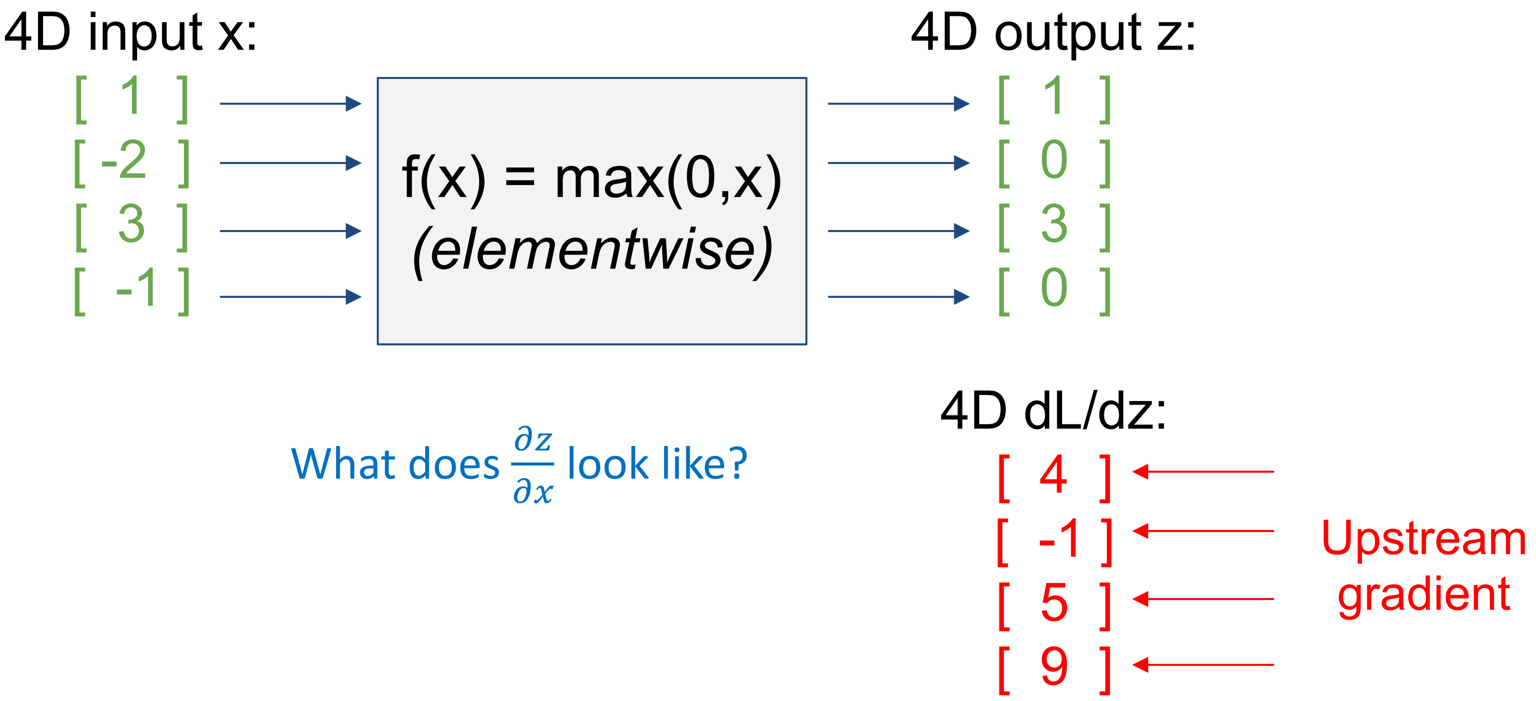

*Slide credit: Stanford CS231n Instructors*

# Backprop with Vectors




*Slide credit: Stanford CS231n Instructors*

# Backprop with Vectors




What does $\frac{\partial z}{\partial x}$ look like?

*Slide credit: Stanford CS231n Instructors*

# Backprop with Vectors

4D input x:

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

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

4D output z:

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

Jacobian dz/dx

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

4D dL/dz:

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

Upstream gradient

*Slide credit: Stanford CS231n Instructors*

# Backprop with Vectors

4D input x:

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

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

4D output z:

[ 1 ]
[ 0 ]
[ 3 ]
[ 0 ]

[dz/dx] [dL/dz]

[ 1 0 0 0 ] [ 4 ]
[ 0 0 0 0 ] [ -1 ]
[ 0 0 1 0 ] [ 5 ]
[ 0 0 0 0 ] [ 9 ]

4D dL/dz:

[ 4 ]
[ -1 ]
[ 5 ]
[ 9 ]

Upstream gradient

*Slide credit: Stanford CS231n Instructors*

# Backprop with Vectors

4D input x:

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

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

4D output z:

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

4D dL/dx:

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

[dz/dx] [dL/dz]

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

4D dL/dz:

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

Upstream gradient

*Slide credit: Stanford CS231n Instructors*

# Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \rightarrow \boxed{\begin{array}{c} f(x) = \max(0,x) \\ \textit{(elementwise)} \end{array}} \rightarrow \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

4D output z:

For element-wise ops, jacobian is **sparse**: off-diagonal entries always zero! Never **explicitly** form Jacobian -- instead use Hadamard (element-wise) multiplication

4D dL/dx:

[dz/dx] [dL/dz]

4D dL/dz:

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

Upstream gradient

*Slide credit: Stanford CS231n Instructors*

# Backprop with Vectors

For element-wise ops, jacobian is **sparse**: off-diagonal entries always zero! Never **explicitly** form Jacobian -- instead use Hadamard (element-wise) multiplication

4D input x:

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

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

4D output z:

[ 1 ]
[ 0 ]
[ 3 ]
[ 0 ]

4D dL/dx:

[ 4 ]
[ 0 ]
[ 5 ]
[ 0 ]

[dz/dx] [dL/dz]

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

4D dL/dz:

[ 4 ]
[ -1 ]
[ 5 ]
[ 9 ]

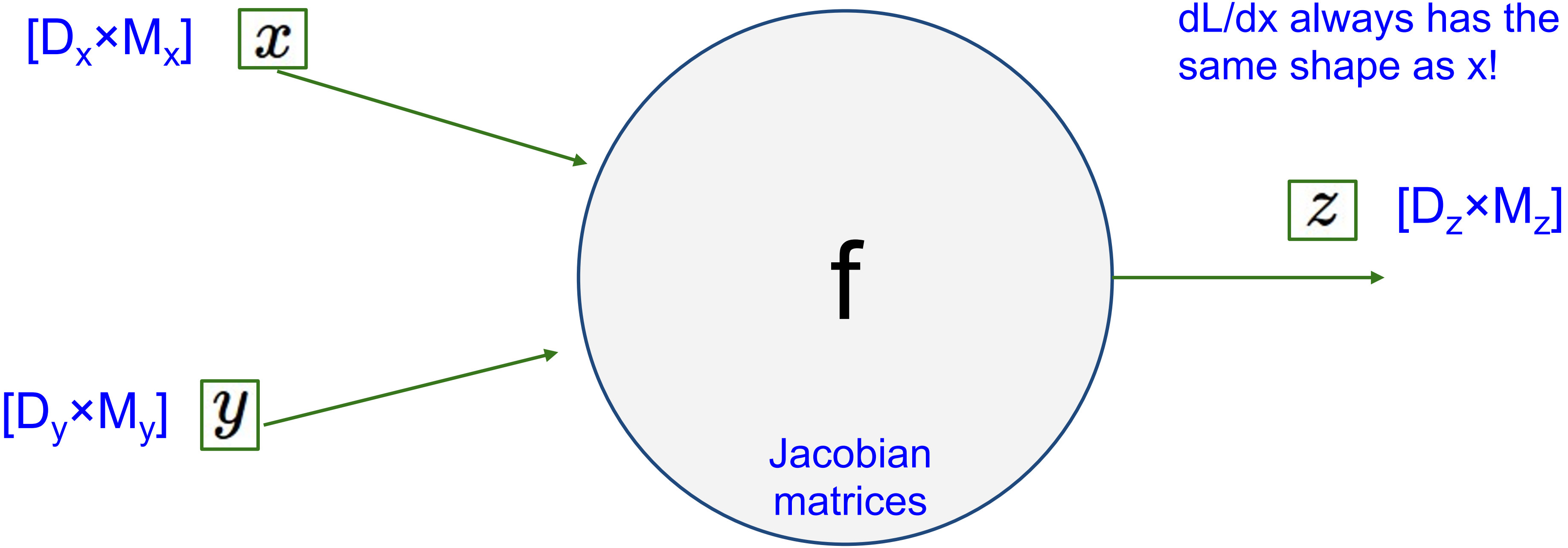
Upstream gradient

*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices (or Tensors)

Loss L still a scalar!

dL/dx always has the same shape as x!

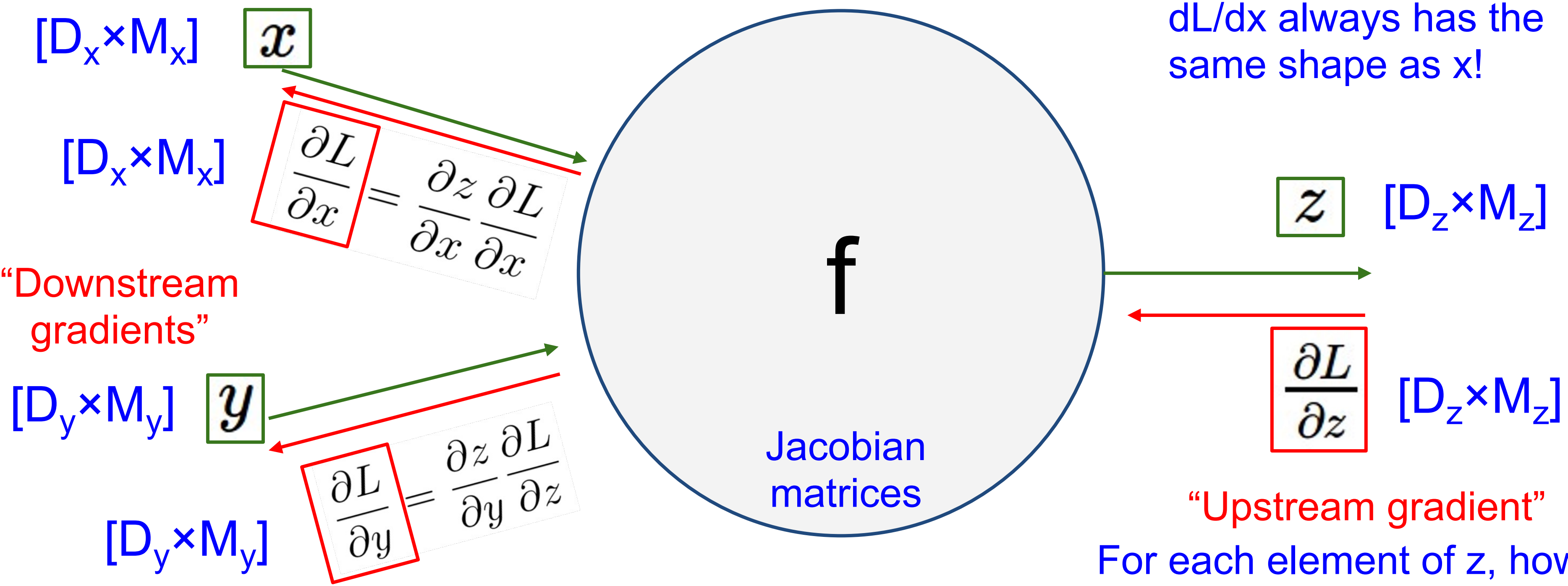



*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices (or Tensors)

Loss L still a scalar!

dL/dx always has the same shape as x!




“Upstream gradient”

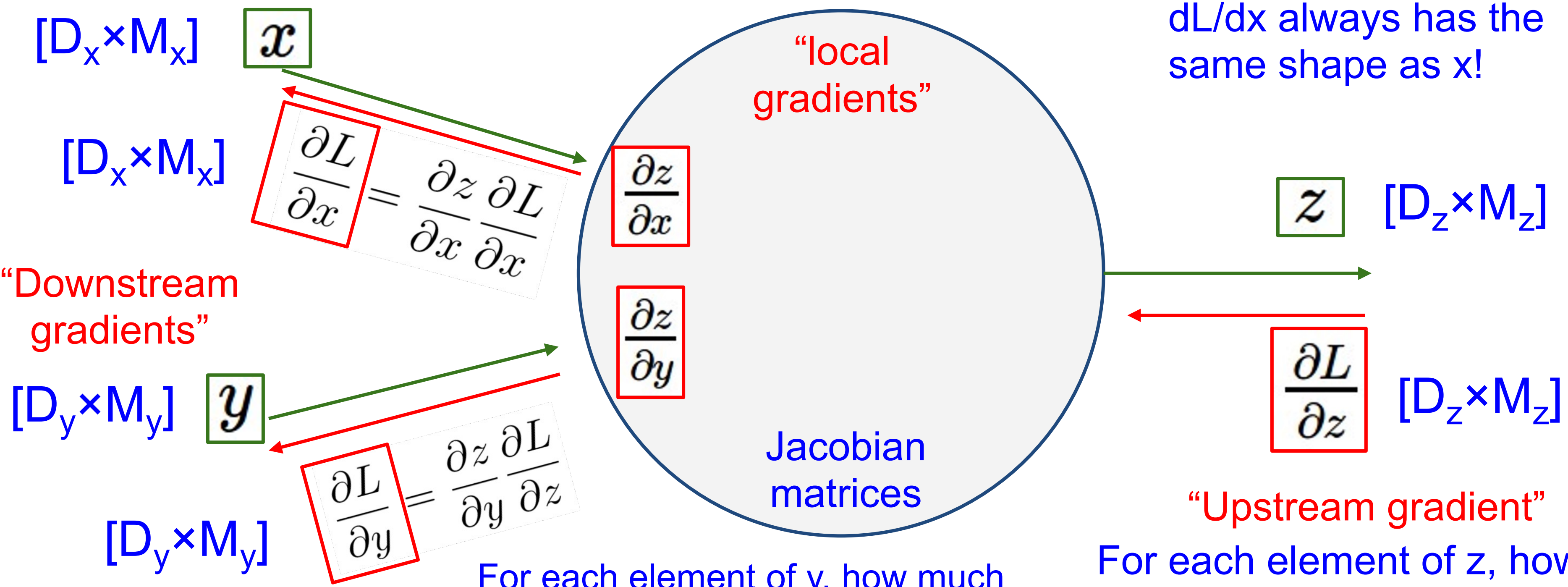
For each element of z, how much does it influence L?

*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices (or Tensors)

Loss L still a scalar!

dL/dx always has the same shape as x!




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

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

*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices (or Tensors)

Loss L still a scalar!

dL/dx always has the same shape as x!

$[D_x \times M_x]$ $x$

$[D_x \times M_x]$ $\boxed{\frac{\partial L}{\partial x}} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z}$

"Downstream gradients"

$[D_y \times M_y]$ $y$

$[D_y \times M_y]$ $\boxed{\frac{\partial L}{\partial y}} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}$

"local gradients"

$\boxed{\frac{\partial z}{\partial x}}$ $[(D_x \times M_x) \times (D_z \times M_z)]$

$\boxed{\frac{\partial z}{\partial y}}$ $[(D_y \times M_y) \times (D_z \times M_z)]$

Jacobian matrices

$z$ $[D_z \times M_z]$

$\boxed{\frac{\partial L}{\partial z}}$ $[D_z \times M_z]$

"Upstream gradient"

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

Flatten the two matrices -> vector-vector gradients -> jacobian matrices!

*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices

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**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]
**[13 9 -2 -6 ]**
[ 5 2 17 1 ]

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



*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices

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

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]
**[13 9 -2 -6 ]**
[ 5 2 17 1 ]

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

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

What does the jacobian matrix look like?



Slide credit: Stanford CS231n Instructors

# Backprop with Matrices

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**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]
[13 9 -2 -6 ]
[ 5 2 17 1 ]

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

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

For a neural net with
N=64, D=M=4096
Each Jacobian takes 256 GB of memory!
Must exploit its sparsity!



*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices

x: [N×D]

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

w: [D×M]

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

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

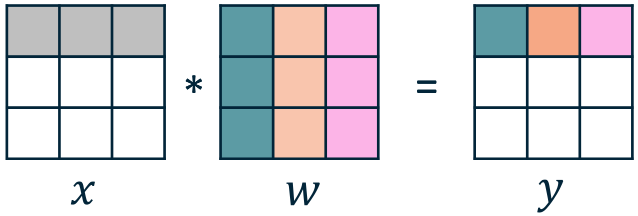
y: [N×M]

$$\begin{bmatrix} 13 & 9 & -2 & -6 \\ 5 & 2 & 17 & 1 \end{bmatrix}$$

dL/dy: [N×M]

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

**Q**: What parts of y are affected by one element of x?






*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices

x: [N×D]

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

w: [D×M]

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

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

$$\begin{bmatrix} \boxed{13 \quad 9 \quad -2 \quad -6} \\ 5 \quad 2 \quad 17 \quad 1 \end{bmatrix}$$

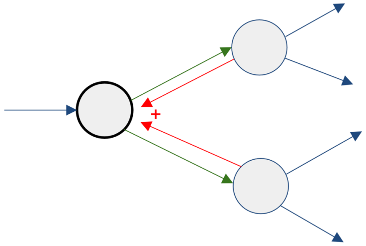
dL/dy: [N×M]

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

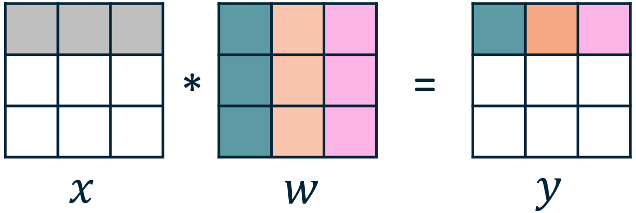
**Q**: What parts of y are affected by one element of x?

**A**: $\boxed{x_{n,d}}$ affects the whole row $y_{n,\cdot}$

$$\frac{\partial L}{\partial x_{n,d}} = \boxed{\sum_m} \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}}$$




Recall the branching gradient rule!






*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices

x: [N×D]

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

w: [D×M]

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

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

$$\begin{bmatrix} \boxed{13 \quad 9 \quad -2 \quad -6} \\ 5 \quad 2 \quad 17 \quad 1 \end{bmatrix}$$

dL/dy: [N×M]

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

**Q**: What parts of y are affected by one element of x?

**A**: $\boxed{x_{n,d}}$ affects the whole row $y_{n,\cdot}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \boxed{\frac{\partial y_{n,m}}{\partial x_{n,d}}}$$

Upstream gradient

local gradient



*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices

x: [N×D]

[ 2 $\boxed{1}$ -3 ]
[ -3 4 2 ]

w: [D×M]

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

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

[ **13 9 -2 -6** ]
[ 5 2 17 1 ]

dL/dy: [N×M]

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

**Q**: What parts of y are affected by one element of x?

**A**: $\boxed{x_{n,d}}$ affects the whole row $y_{n,\cdot}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}}$$

**Q**: How much does $\boxed{x_{n,d}}$ affect $\boxed{y_{n,m}}$?



*Slide credit: Stanford CS231n Instructors*

# Backprop with Matrices

x: [N×D]

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

w: [D×M]

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

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

$$\begin{bmatrix} 13 & 9 & -2 & -6 \\ 5 & 2 & 17 & 1 \end{bmatrix}$$

dL/dy: [N×M]

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

**Q**: What parts of y are affected by one element of x?

**A**: $\boxed{x_{n,d}}$ affects the whole row $y_{n,\cdot}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}}$$

**Q**: How much does $\boxed{x_{n,d}}$ affect $\boxed{y_{n,m}}$?

$$y_{n,m} = \sum_{i=1}^{D} x_{n,i} w_{i,m}$$

$$\frac{\partial y_{n,m}}{\partial x_{n,d}} = w_{d,m}$$

# Backprop with Matrices

x: [N×D]

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

w: [D×M]

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

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

$$\begin{bmatrix} 13 & 9 & \boxed{-2} & -6 \\ 5 & 2 & 17 & 1 \end{bmatrix}$$

dL/dy: [N×M]

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

**Q**: What parts of y are affected by one element of x?

**A**: $\boxed{x_{n,d}}$ affects the whole row $y_{n,\cdot}$

**Q**: How much does $\boxed{x_{n,d}}$ affect $\boxed{y_{n,m}}$?

**A:** $\boxed{w_{d,m}}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m} = \frac{\partial L}{\partial y_n} w_d^T$$

Just a dot product!

# Backprop with Matrices

x: [N×D]
[ 2 $\boxed{1}$ -3 ]
[ -3 4 2 ]

w: [D×M]
[ 3 2 1 -1]
[ 2 1 $\boxed{3}$ 2]
[ 3 2 1 -2]

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]
[13 9 $\boxed{-2}$ -6]
[ 5 2 17 1 ]

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

**Q**: What parts of y are affected by one element of x?

**A**: $\boxed{x_{n,d}}$ affects the whole row $y_{n,\cdot}$

**Q**: How much does $\boxed{x_{n,d}}$ affect $\boxed{y_{n,m}}$?

**A**: $\boxed{w_{d,m}}$

[N×D] [N×M] [M×D]

$$\boxed{\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial y}\right) w^T}$$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m} = \frac{\partial L}{\partial y_n} w_d^T$$

Just a matrix multiplication
No jacobian matrix needed!



Slide credit: Stanford CS231n Instructors

# Backprop with Matrices

x: [N×D]

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

w: [D×M]

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

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

$$\begin{bmatrix} 13 & 9 & \boxed{-2} & -6 \\ 5 & 2 & 17 & 1 \end{bmatrix}$$

dL/dy: [N×M]

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

By similar logic:

[N×D] [N×M] [M×D]

$$\boxed{\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial y}\right) w^T}$$

[D×M] [D×N] [N×M]

$$\boxed{\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y}\right)}$$

For a neural net layer with N=64, D=M=4096

The larges matrix ($W$) takes up to 0.13 GB memory



*Slide credit: Stanford CS231n Instructors*

Summary:

- Review backpropagation
- Neural networks, activation functions
- Neurons as biological inspirations to DNNs
- Vector Calculus
- Backpropagation through vectors / matrices

# Next Time: How to Pick a Project!