

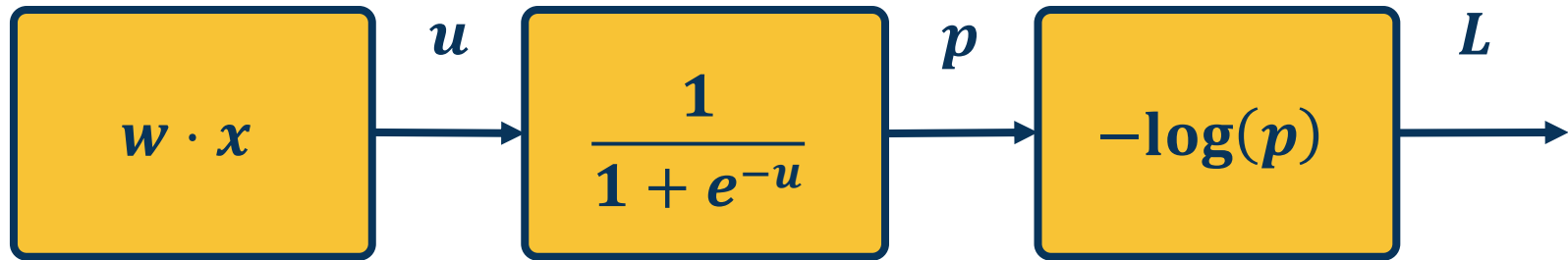
CS 4644 / 7643-A: LECTURE 5

DANFEI XU

Topics:

- Backpropagation
- Neural Networks
- Jacobians

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



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun



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

Chain rule and Backpropagation!

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

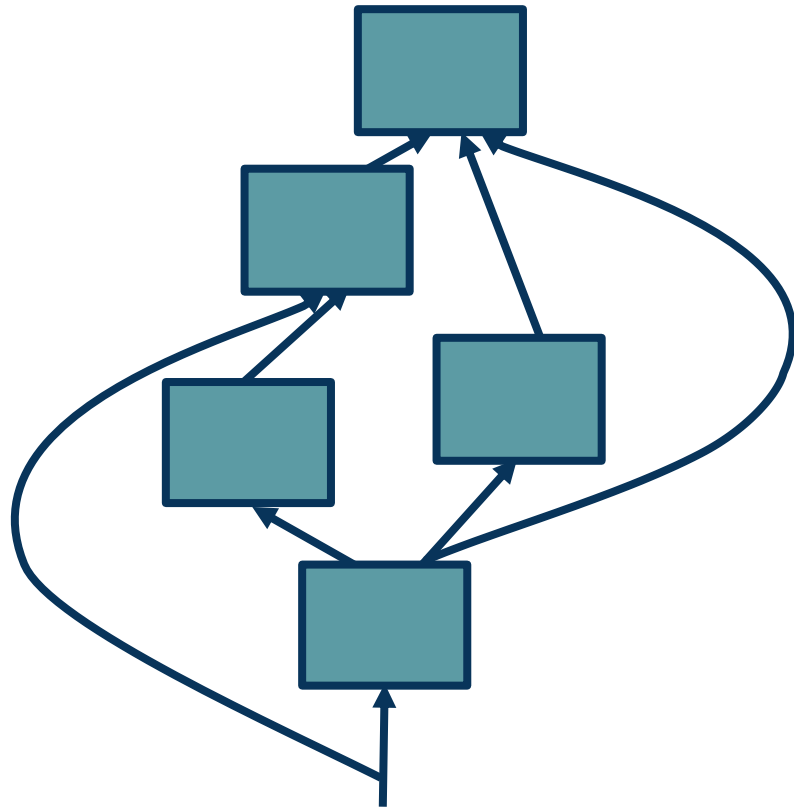
Recap: Computation Graph

We will view the function / model as a **computation graph**

Key idea: break a complex model into atomic computation nodes that can be computed efficiently.

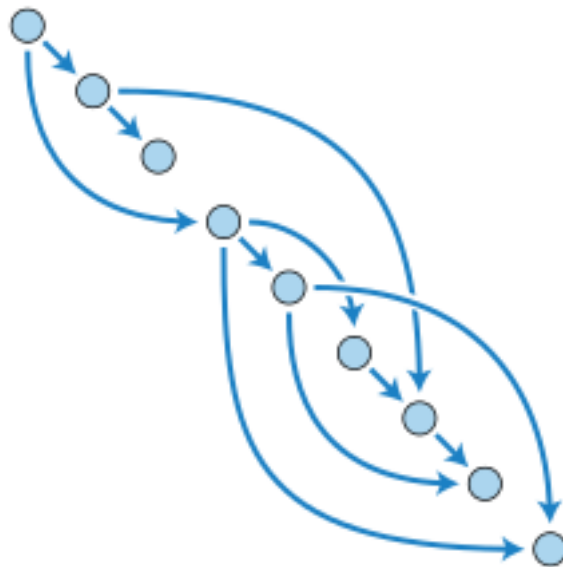
Graph can be any **directed acyclic graph (DAG)**

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

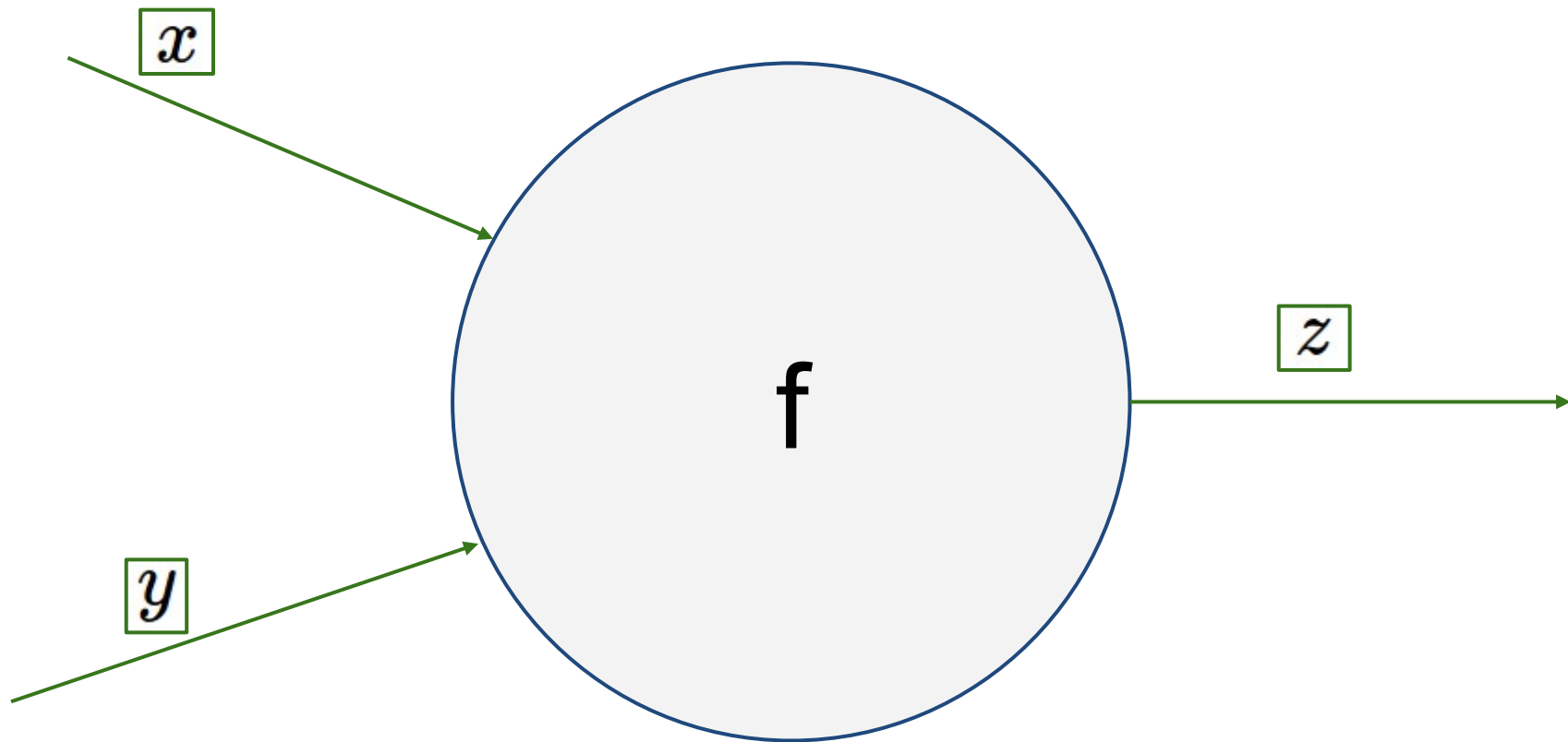


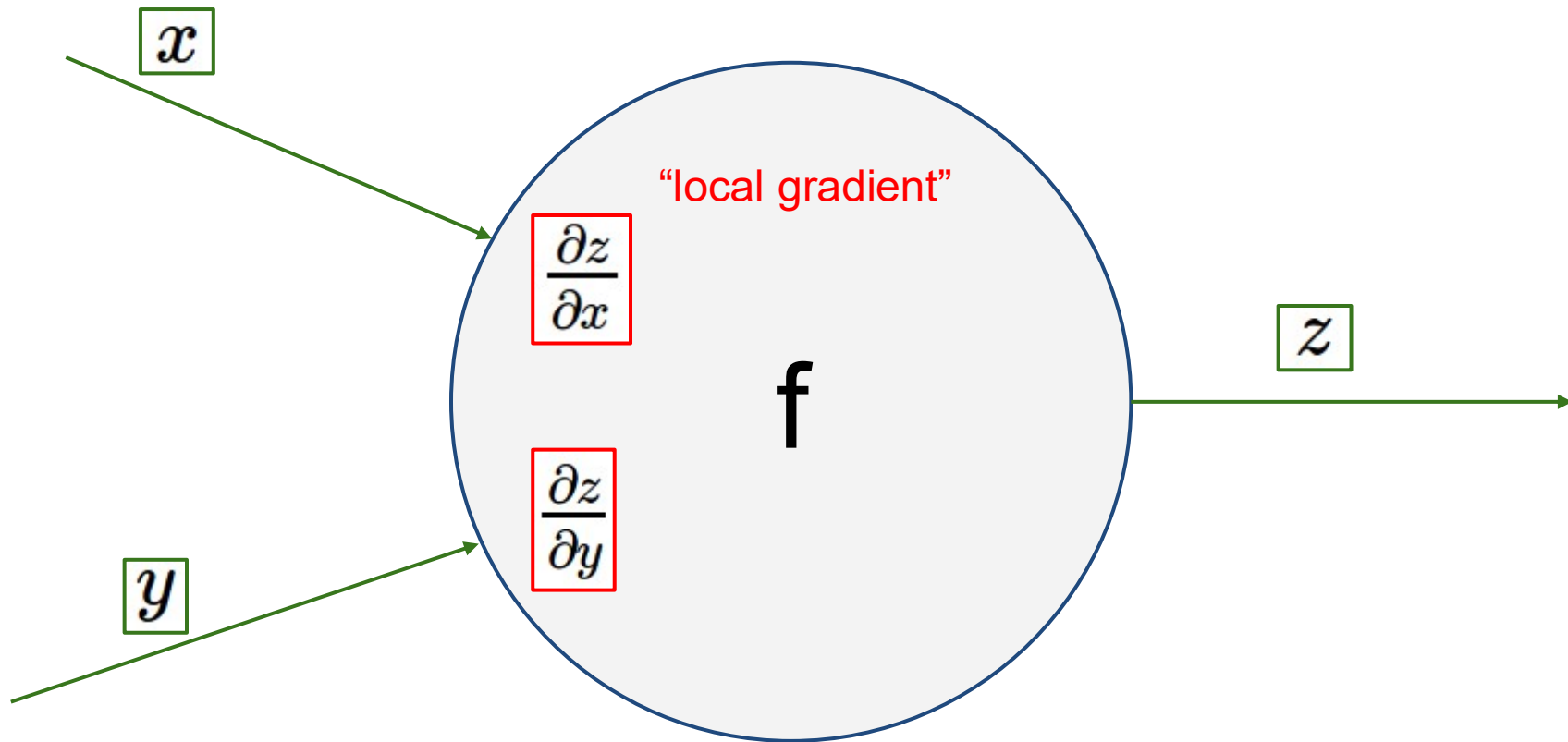
Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

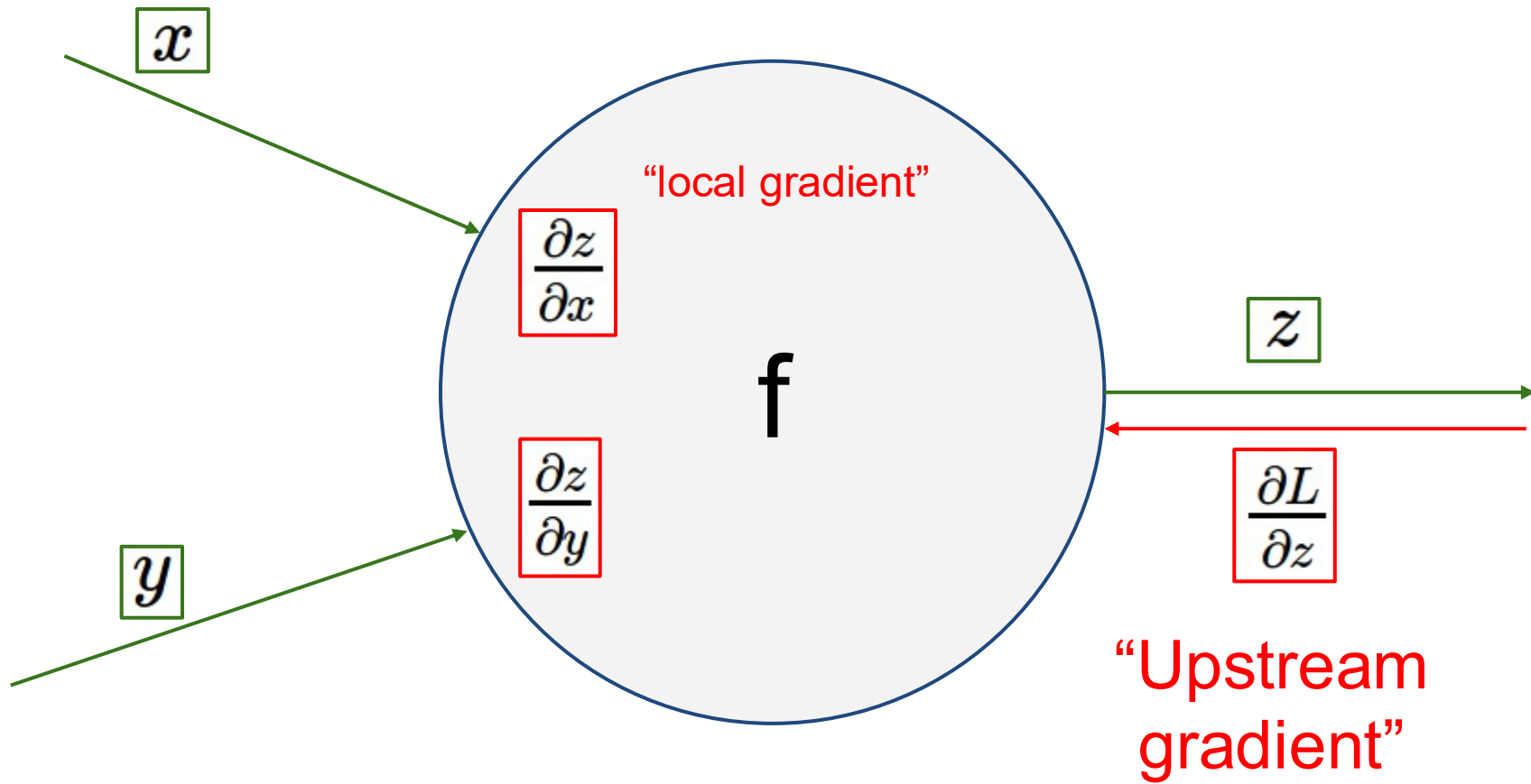
Directed Acyclic Graphs (DAGs)

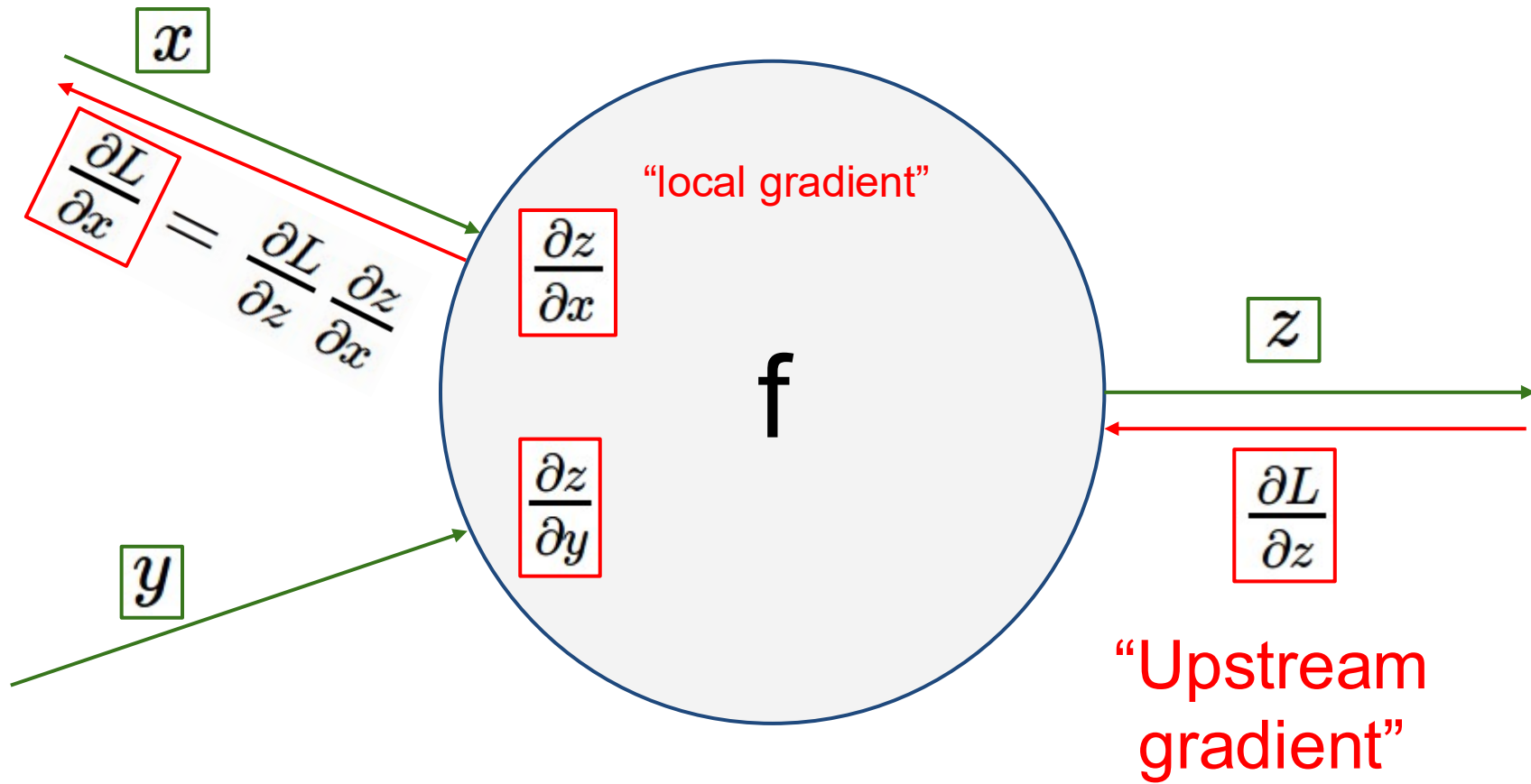


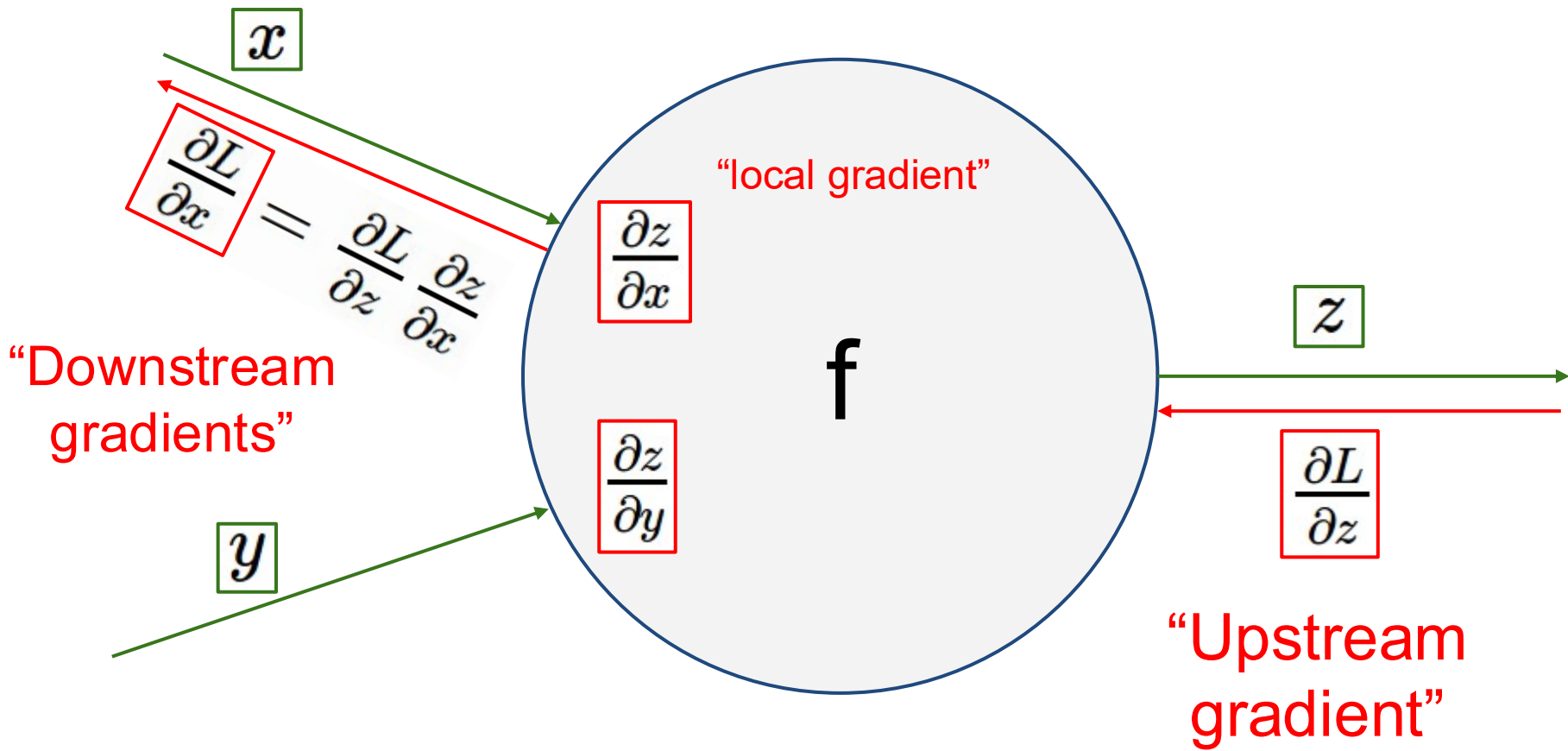
A computation node

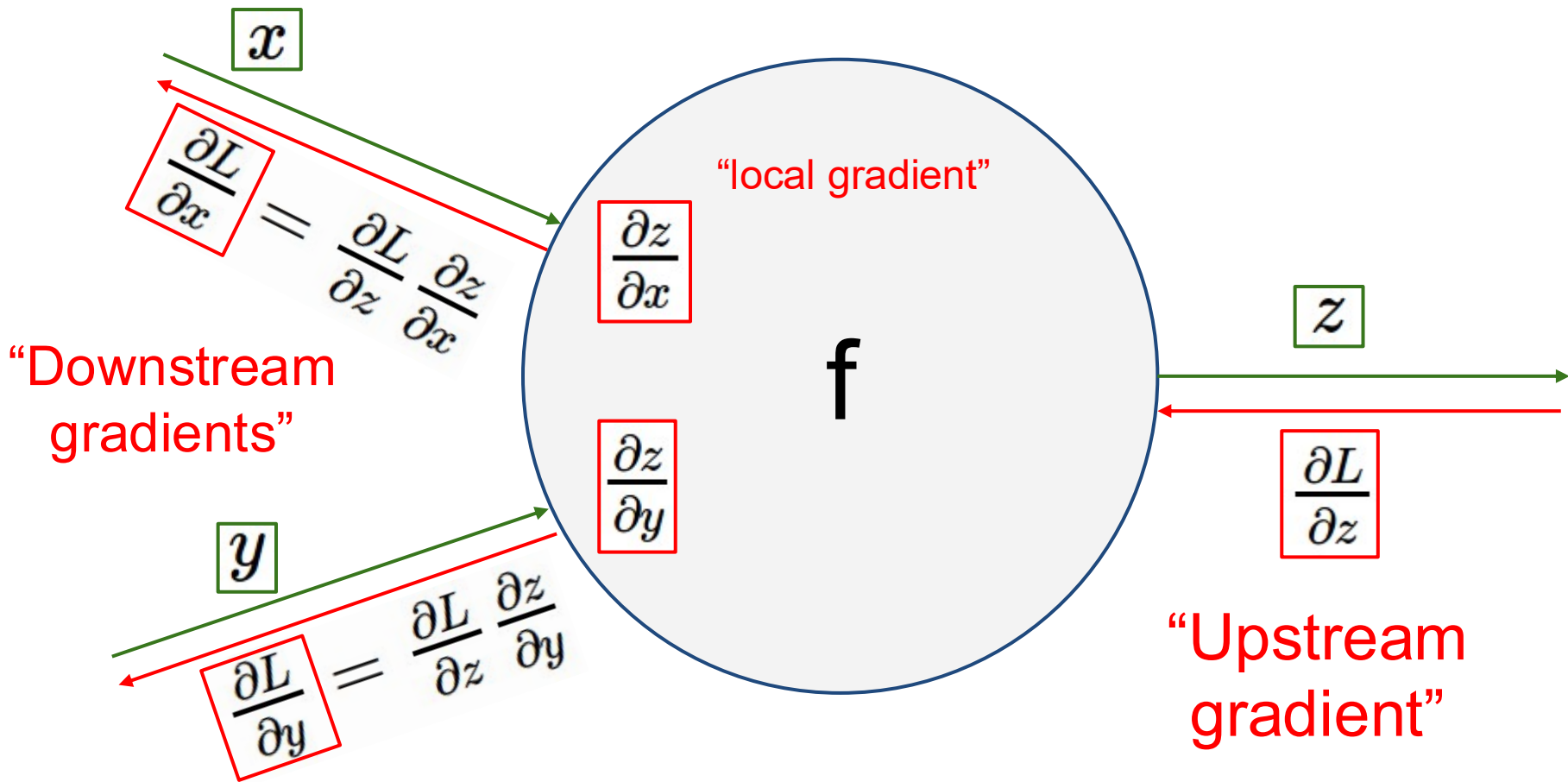


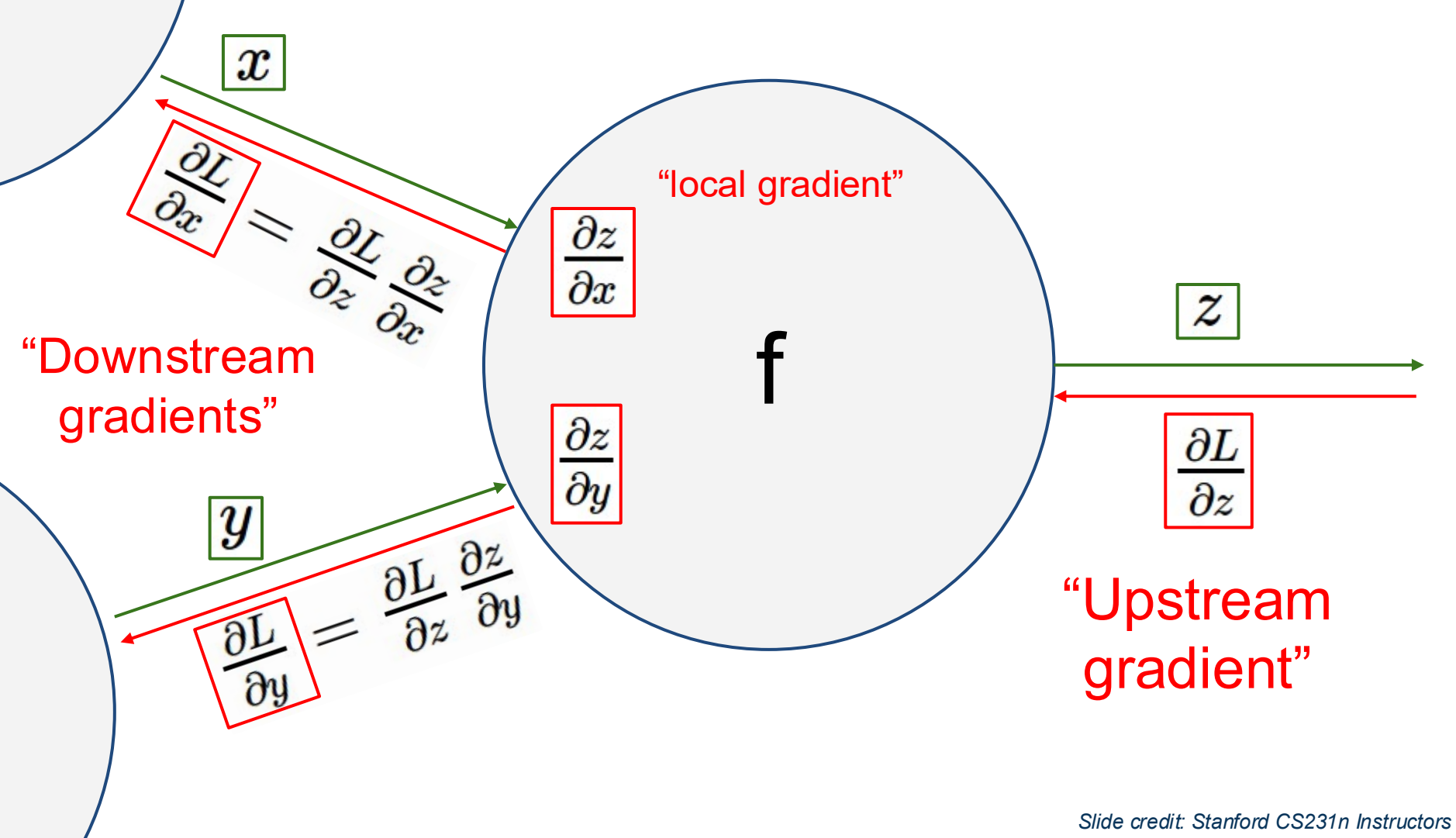










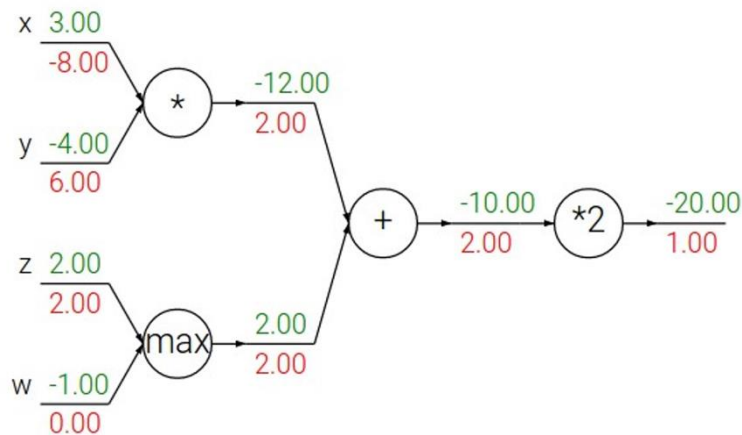


Patterns in backward flow

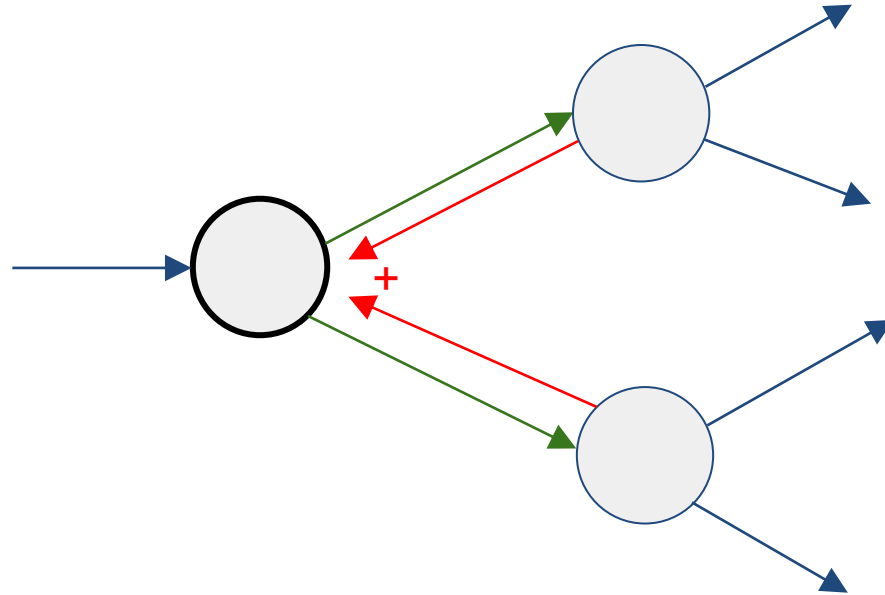
add gate: gradient distributor

max gate: gradient router

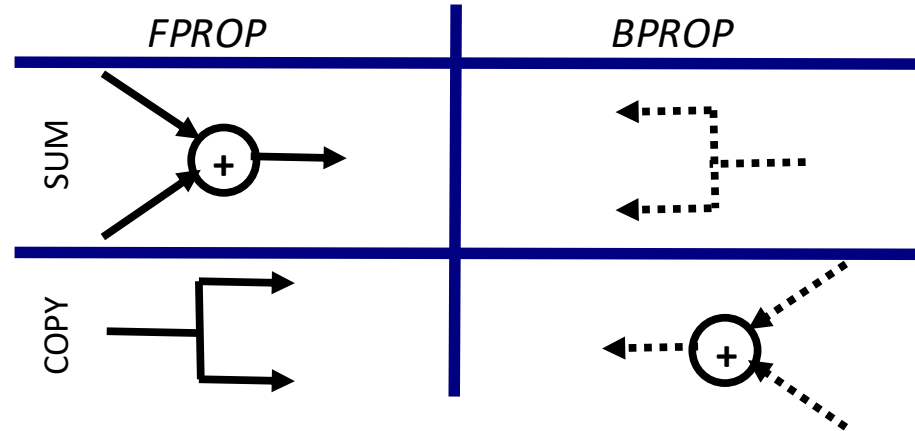
mul gate: gradient switcher



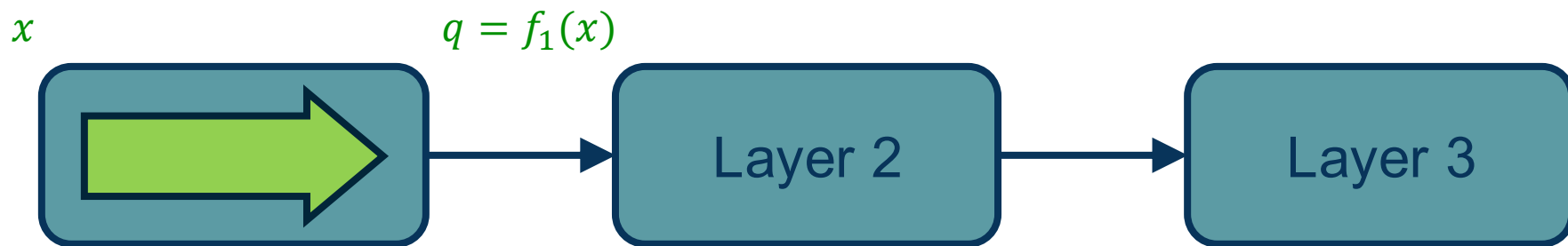
Gradients add at branches



Duality in Fprop and Bprop

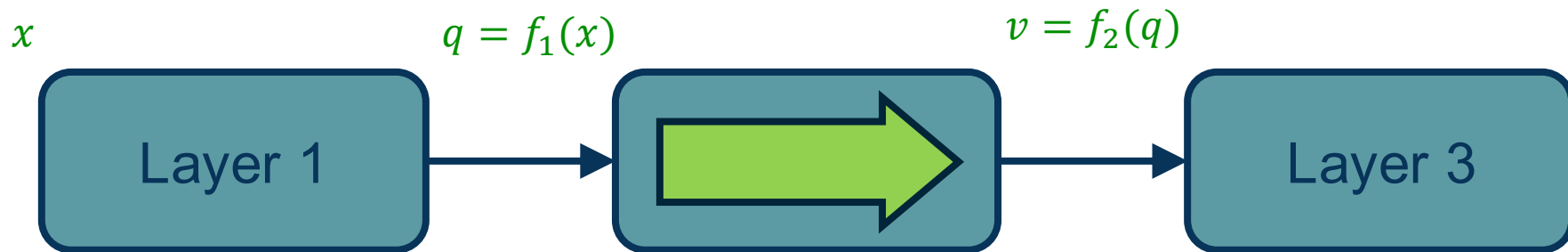


Step 1: Compute Loss on Mini-Batch: Forward Pass



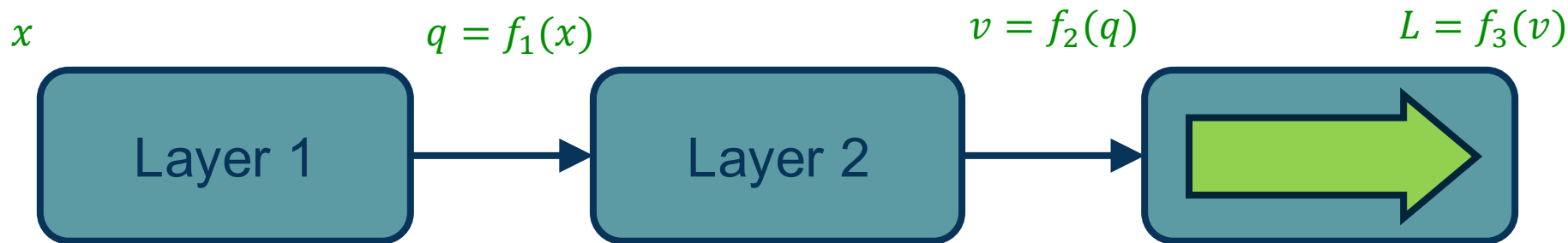
Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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



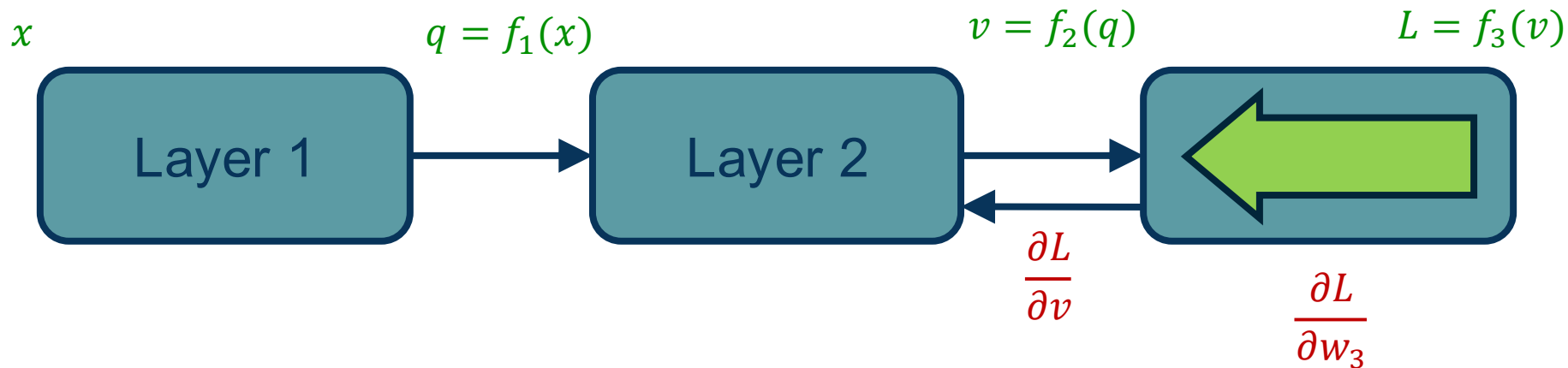
Note that we must store the **intermediate outputs of all layers**!

- ⬢ 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

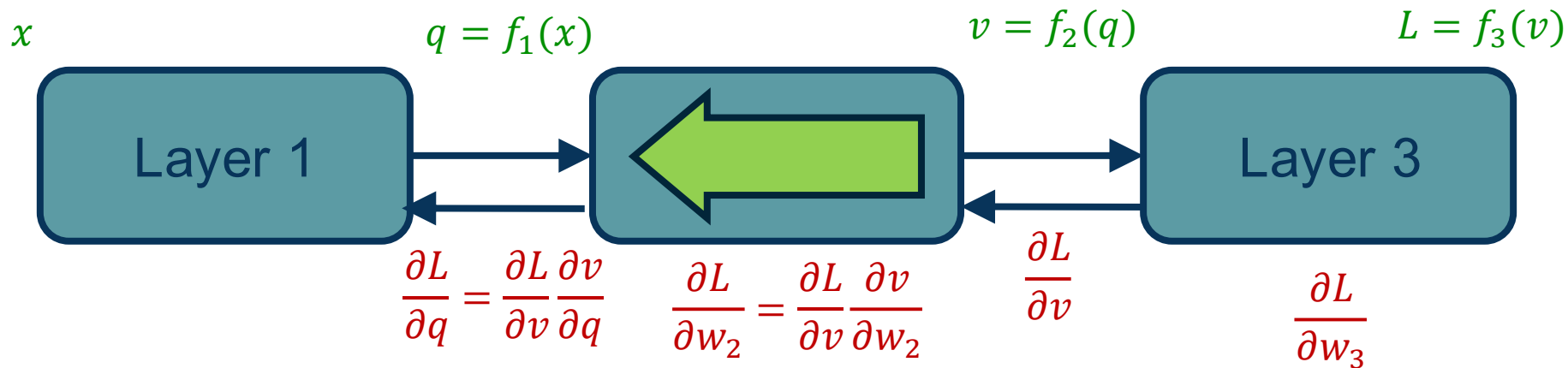
Step 2: Compute Gradients wrt parameters: Backward Pass



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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

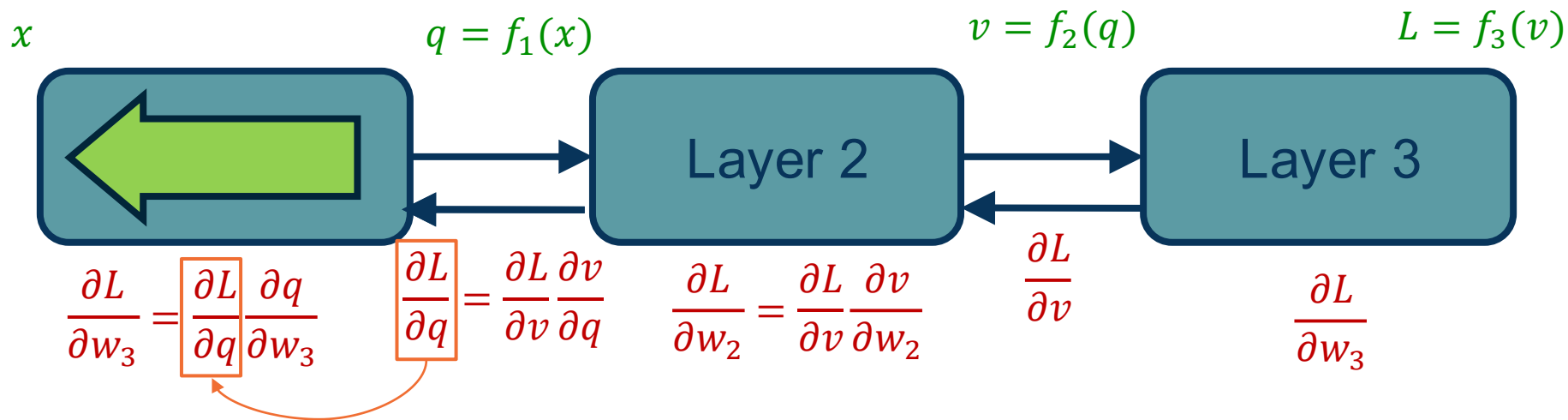
Step 2: Compute Gradients wrt parameters: Backward Pass



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

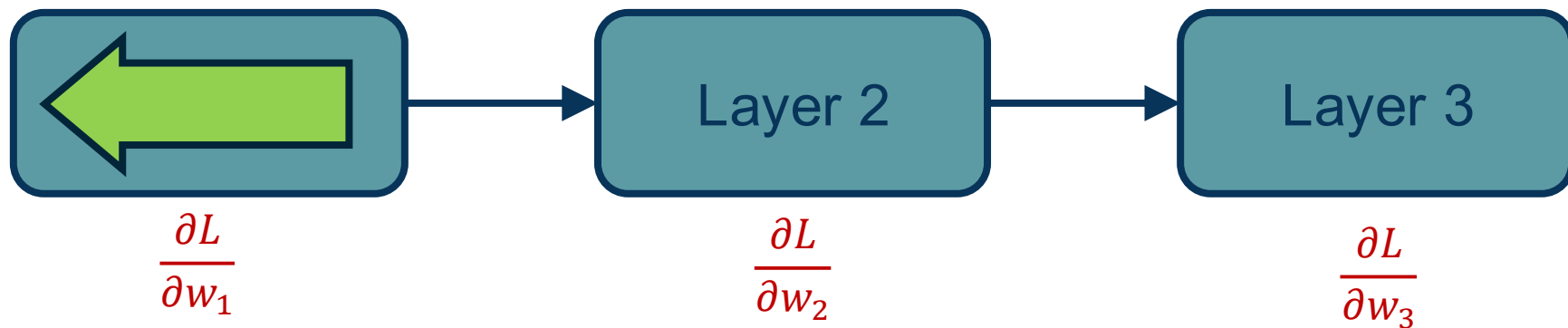


Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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



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

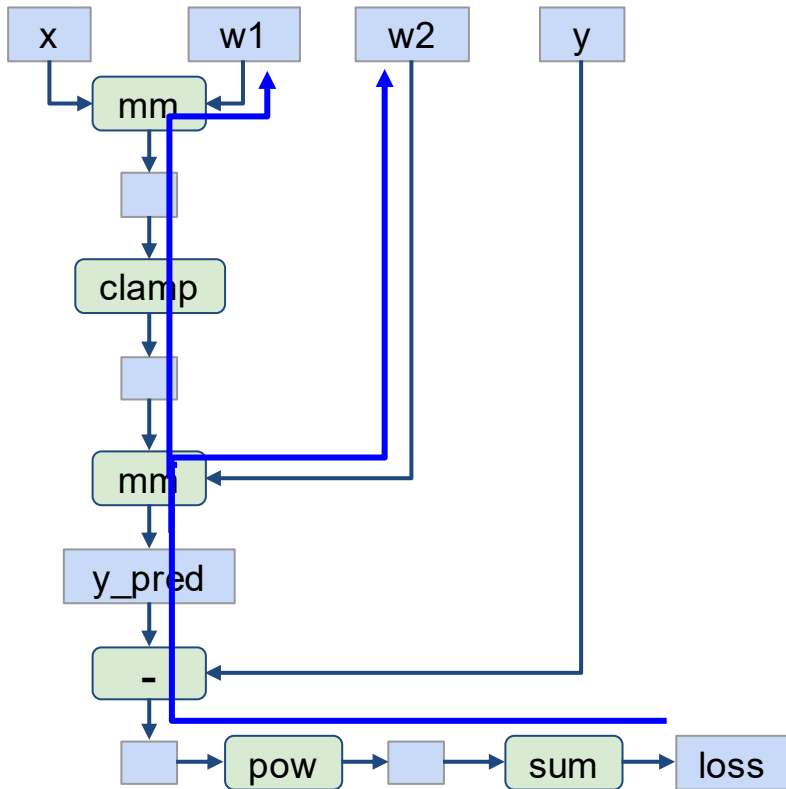
Gradient Descent!

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Deep Learning Framework = Differentiable Programming Engine

- Computation = Graph
 - Input = Data + Parameters
 - Output = Loss
 - Scheduling = Topological ordering
- What do we need to do?
 - Generic code for representing the graph of modules
 - Specify modules (both forward and backward function)

PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Search for path between loss and w1, w2
(for backprop) AND perform computation

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?
- Non-linear functions
- How do we run backpropagation on neural nets? (Matrix Calculus!)

Neural Network

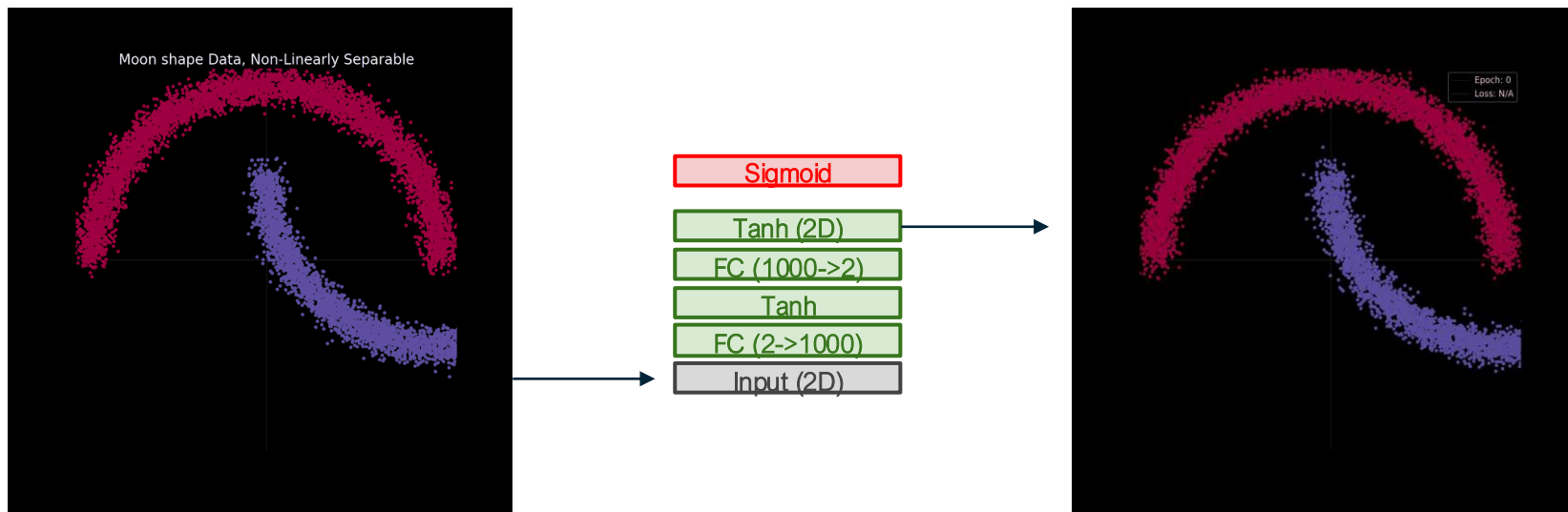
Linear
classifier



[This image](#) is [CC0 1.0](#) public domain

(Deep) Representation Learning for Classification

A function that transforms raw data space into a linearly-separable space



Neural networks: the original linear classifier

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

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

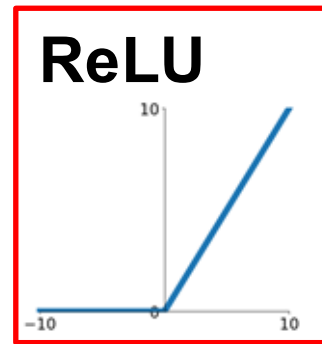
Neural networks: 2 layers

(**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: 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

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

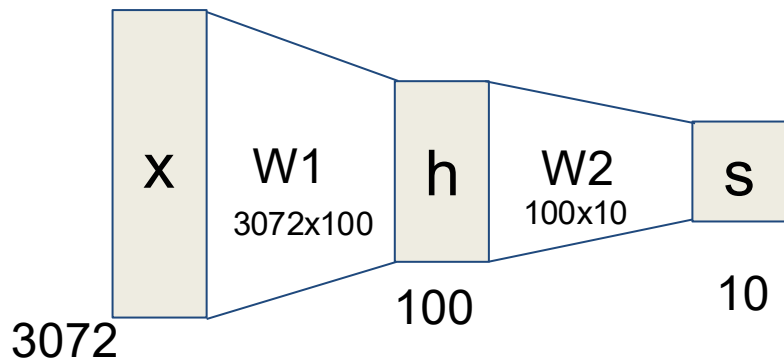
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

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

Neural networks: hierarchical computation

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

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



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

Neural networks: why is max operator important?

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

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

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

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

$$f = W_2 W_1 x$$

Neural networks: why is max operator important?

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

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

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

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

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

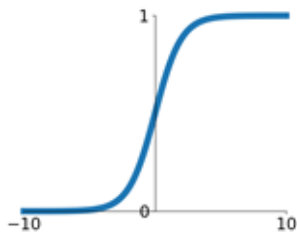
A: We end up with a linear classifier again!

(Non-linear) activation function allows us to build non-linear functions with NNs.

Activation functions

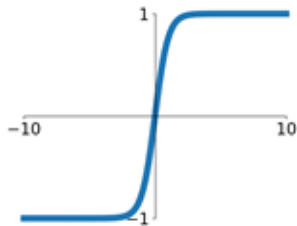
Sigmoid

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



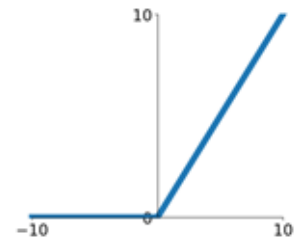
tanh

$$\tanh(x)$$



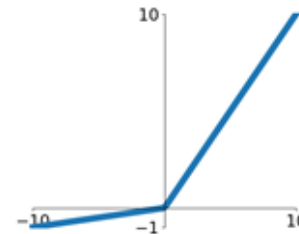
ReLU

$$\max(0, x)$$



Leaky ReLU

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

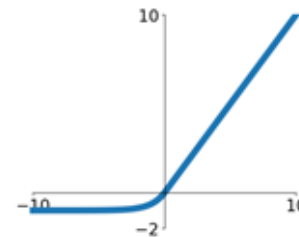


Maxout

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

ELU

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

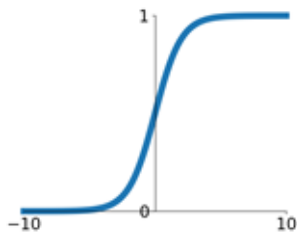


More on this in week 6...

Activation functions

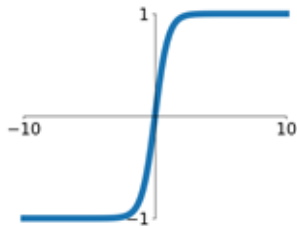
Sigmoid

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



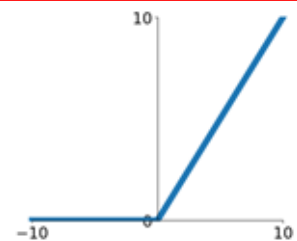
tanh

$$\tanh(x)$$



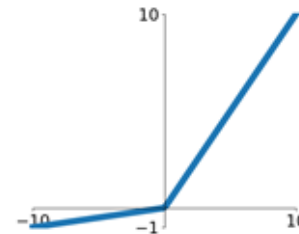
ReLU

$$\max(0, x)$$



Leaky ReLU

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

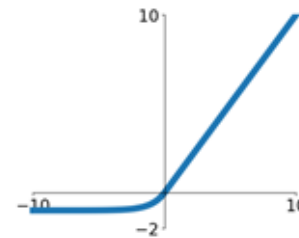


Maxout

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

ELU

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



ReLU is a good default
choice for most problems

Neural Networks are Universal Function Approximators

Claim: Neural Networks with certain non-linear activation functions are universal function approximators.

- What the heck are universal function approximators?
- Why are NNs considered universal function approximators?
- Why does it matter?

Neural Networks are Universal Function Approximators

Claim: Neural Networks with certain non-linear activation functions are universal function approximators.

A quick primer on approximation theory.

A branch of mathematics that deals with how functions can be approximated by simpler or more tractable functions, while maintaining some measure of closeness to the original function.

Example: approximating $f(x) = e^x$.

e^x are known as *transcendental functions*: you cannot calculate its exact value with finitely many basic algebraic operations like multiplication, addition, and power.

But we can approximate e^x with a polynomial with bounded error:

$$\sum_{k=1}^N \frac{1}{k!} x^k$$

Neural Networks are Universal Function Approximators

Claim: Neural Networks with certain non-linear activation functions are universal function approximators.

NNs as function approximators

A single layer network with a sigmoid activation $\sigma = \frac{1}{1+e^{-x}}$ can be written as

$$F(x) = \sum_{i=1}^M v_i \sigma(w_i^T x + b_i)$$

Is the family of single layer network with sigmoid activation enough to approximate any reasonable function (more on this next slide)?

$$\mathcal{F} = \left\{ \sum_{i=1}^M v_i \sigma(w_i^T x + b_i) : w_i, b_i \in \mathbb{R}^N, v_i \in \mathbb{R} \right\}$$

Neural Networks are Universal Function Approximators

Claim: Neural Networks with certain non-linear activation functions are universal function approximators.

The universal approximation theorem (Cybenko, G. 1989)

Theorem 1. *Let σ be any continuous discriminatory function. Then finite sums of the form*

$$G(x) = \sum_{j=1}^N \alpha_j \sigma(y_j^T x + \theta_j) \quad (2)$$

are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\varepsilon > 0$, there is a sum, $G(x)$, of the above form, for which

$$|G(x) - f(x)| < \varepsilon \quad \text{for all } x \in I_n.$$

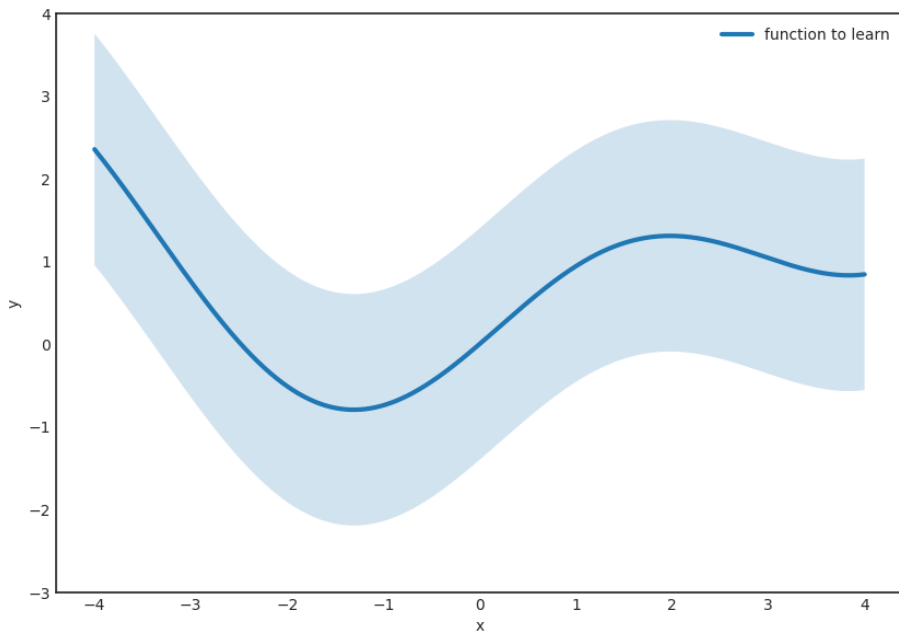
Plain English: as long as the activation function is sigmoid-like and the function to be approximated is continuous, there exists a neural network with a single hidden layer that can approximate it with certain error (bounded uniform norm).

Neural Networks are Universal Function Approximators

Claim: Neural Networks with certain non-linear activation functions are universal function approximators.

A 1-D illustration of the universal approximation theorem

We want to approximate $g(x)$ bounded by some small error ϵ (shaded band) with a single layer NN $F(x)$



Neural Networks are Universal Function Approximators

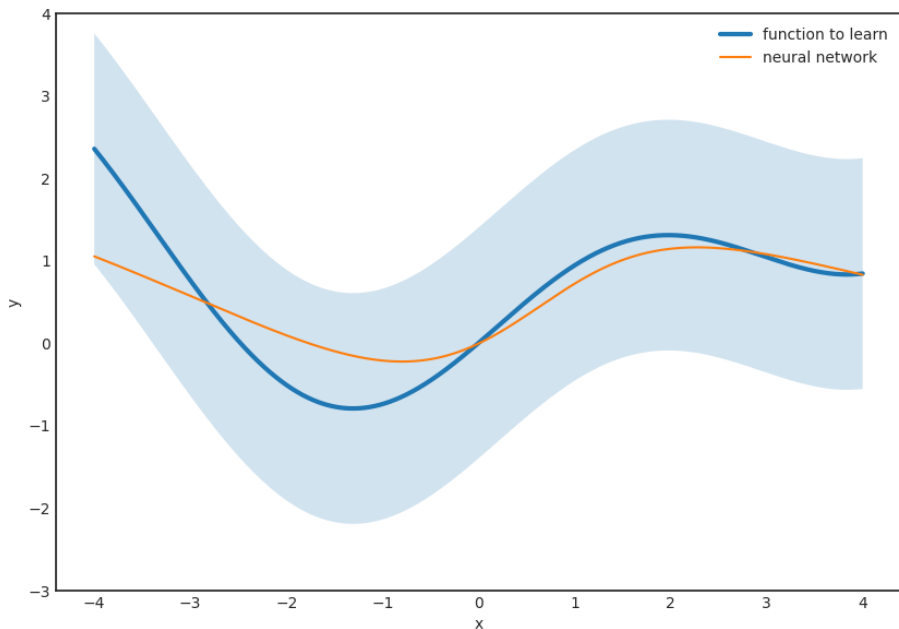
Claim: Neural Networks with certain non-linear activation functions are universal function approximators.

A 1-D illustration of the universal approximation theorem

We want to approximate $g(x)$ bounded by some small error ϵ (shaded band) with a single layer NN $F(x)$

The universal approximation theorem guarantees the existence of such an $F(x)$

... but it doesn't tell us how to get it or what the size of the model (M) should be

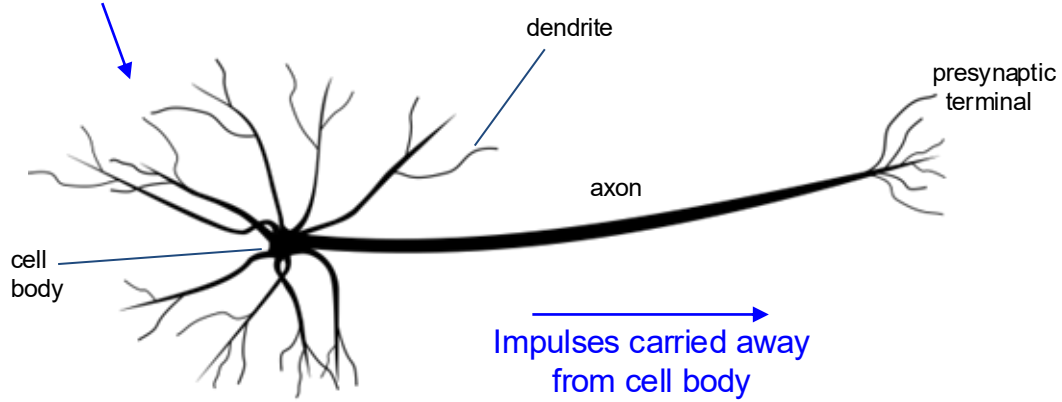


Why are they called Neural Networks, anyway?



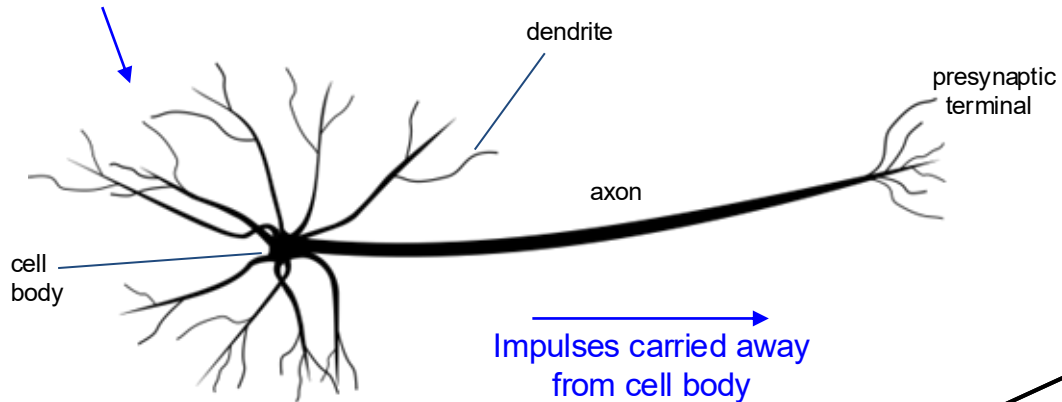
This image by [Fotis Bobolas](#) is
licensed under [CC-BY 2.0](#)

Impulses carried toward cell body

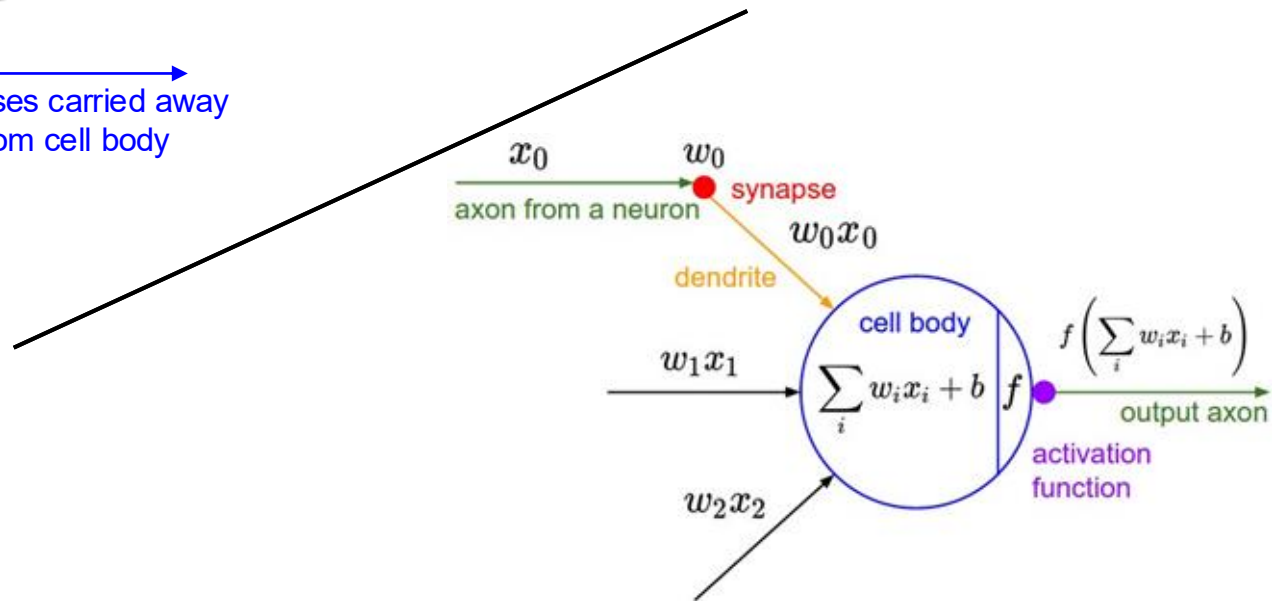


[This image](#) by Felipe Perucho
is licensed under [CC-BY 3.0](#)

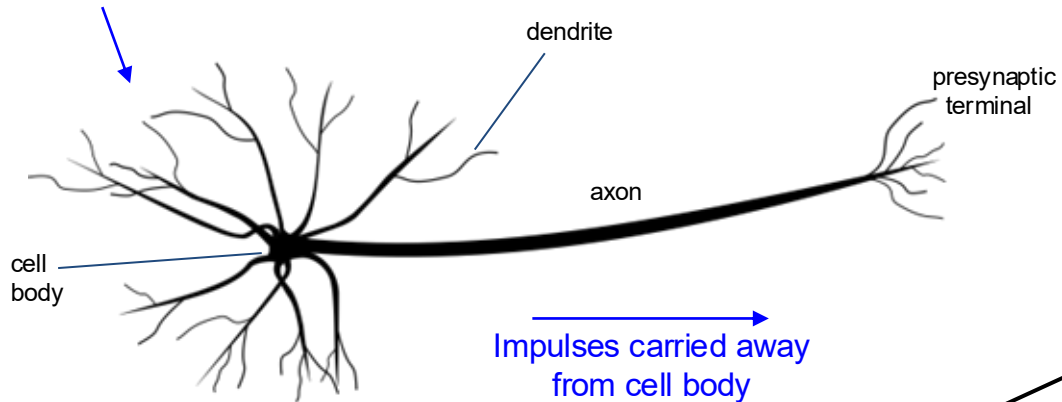
Impulses carried toward cell body



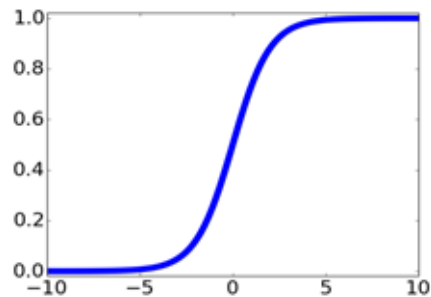
[This image](#) by Felipe Perucho
is licensed under [CC-BY 3.0](#)



Impulses carried toward cell body

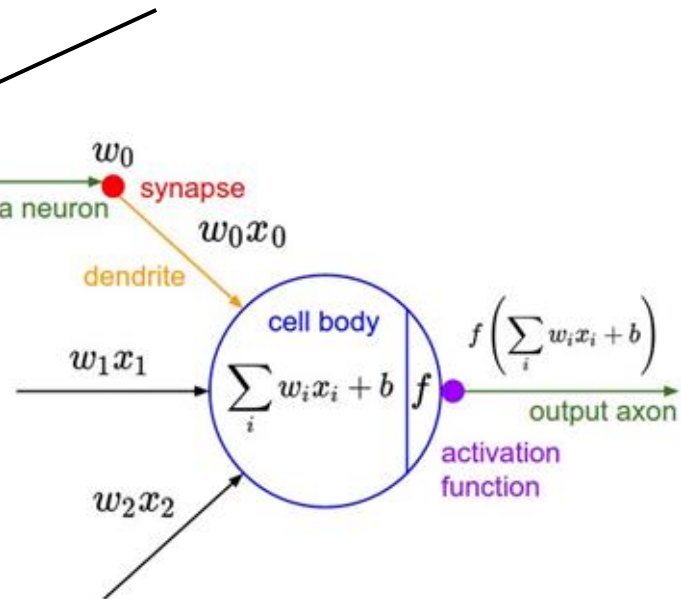


[This image](#) by Felipe Perucho is licensed under [CC-BY 3.0](#)

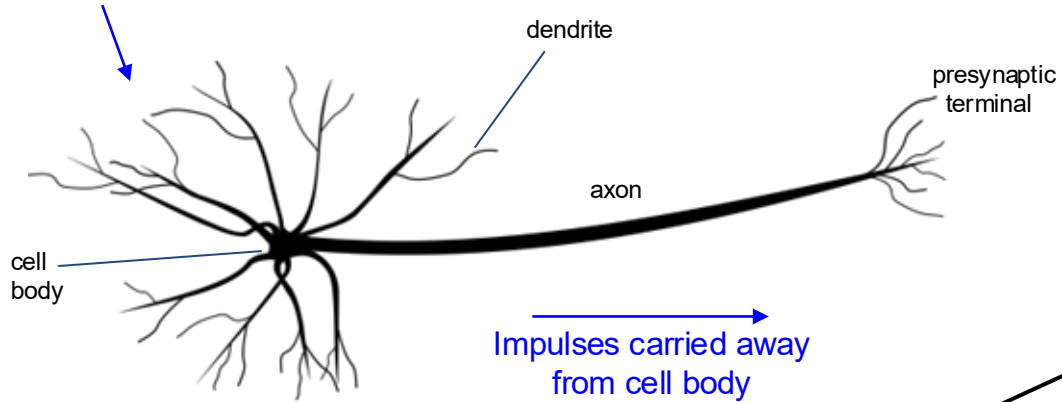


sigmoid activation function

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

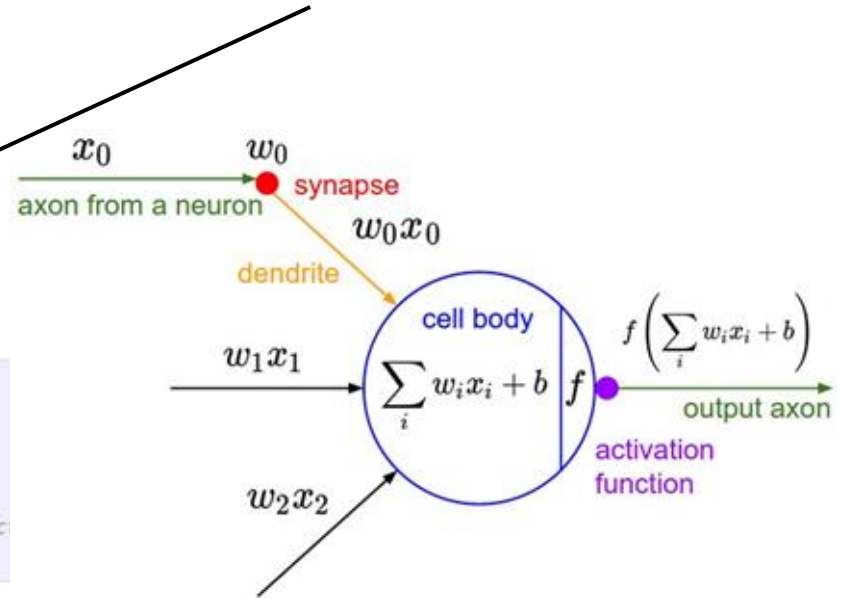


Impulses carried toward cell body

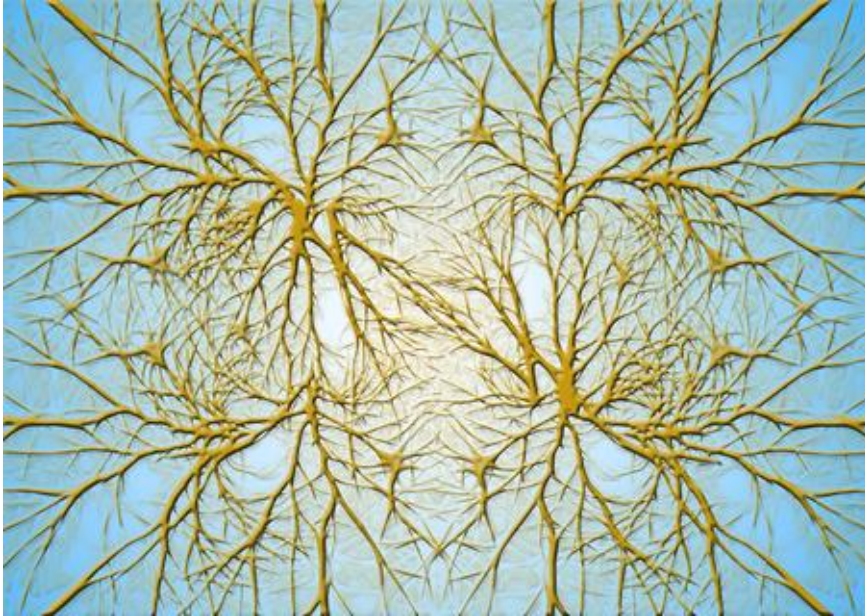


[This image](#) by Felipe Perucho
is licensed under [CC-BY 3.0](#)

```
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
```

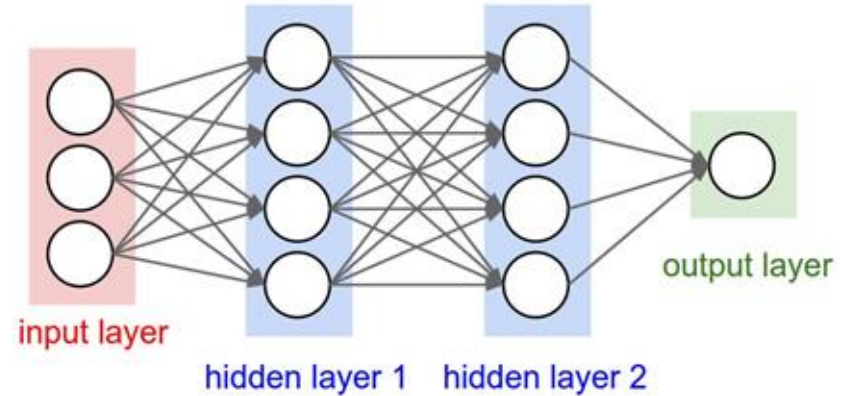


Biological Neurons: Complex connectivity patterns



[This image](#) is [CC0 Public Domain](#)

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



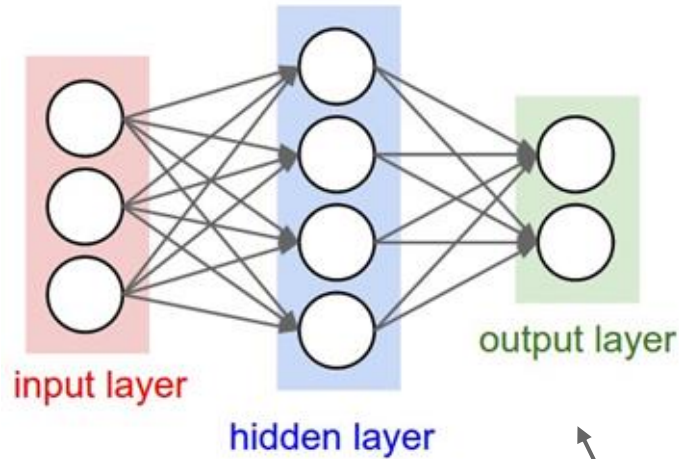
Be very careful with your brain analogies!

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

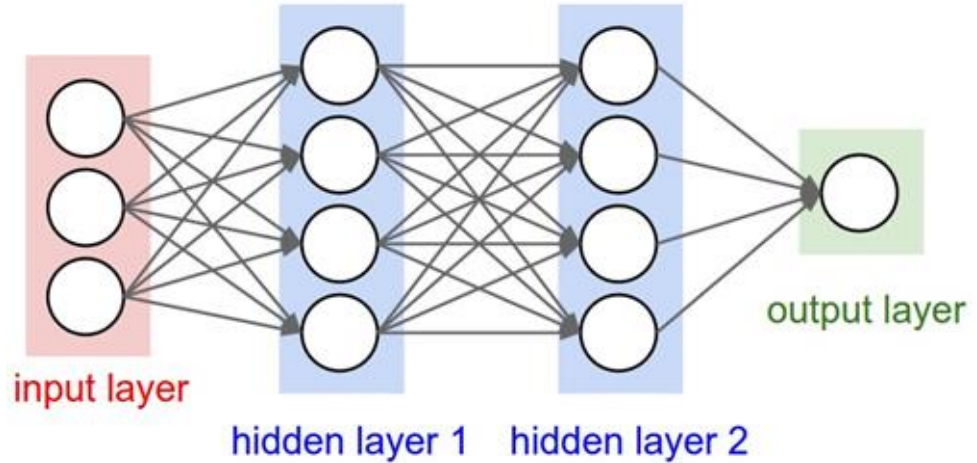
[Dendritic Computation. London and Hausser]

Neural networks: Architectures



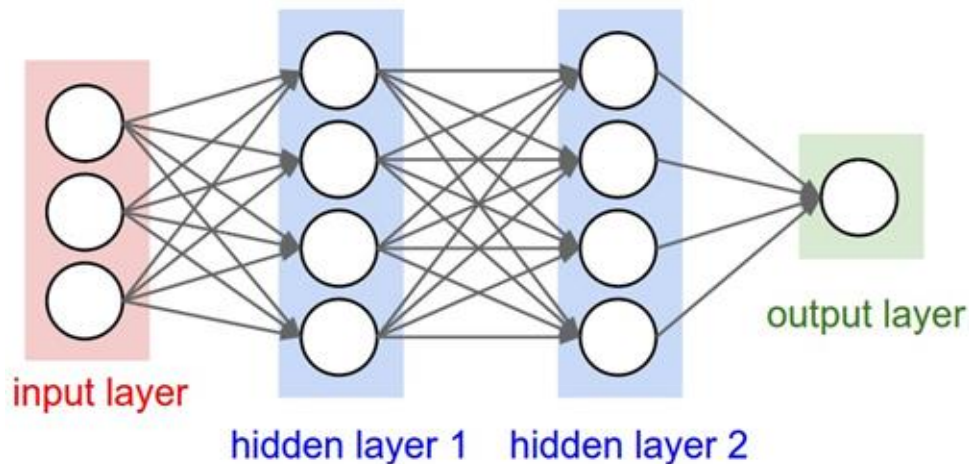
“2-layer Neural Net”, or
“1-hidden-layer Neural Net”

“Fully-connected” layers



“3-layer Neural Net”, or
“2-hidden-layer Neural Net”

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)
```

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

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

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

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

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

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
```

Define the network

```
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
```

Forward pass

```
13
14 grad_y_pred = 2.0 * (y_pred - y)
15 grad_w2 = h.T.dot(grad_y_pred)
16 grad_h = grad_y_pred.dot(w2.T)
17 grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19 w1 -= 1e-4 * grad_w1
20 w2 -= 1e-4 * grad_w2
```

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

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

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

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

Gradient descent

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

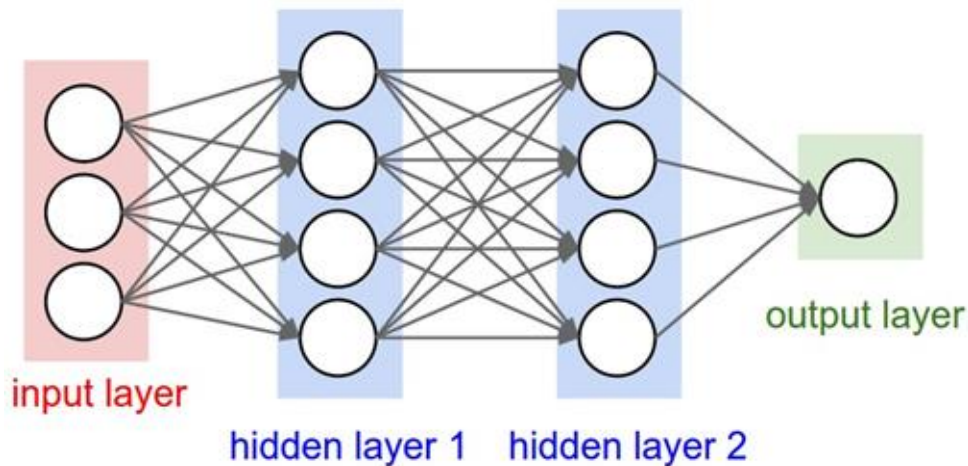
```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

matrix

Calculate the analytical gradients

How to derive this?

Next: Vector Calculus!



How do we do backpropagation with neural nets?

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?



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?



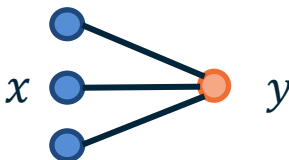
Vector to Scalar

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

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?



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?



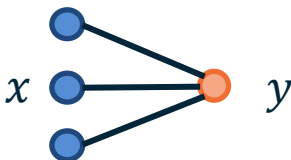
Vector to Scalar

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

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?



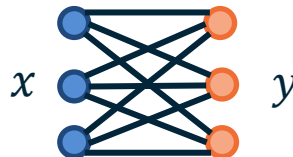
Vector to Vector

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

Derivative is **Jacobian**:

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

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

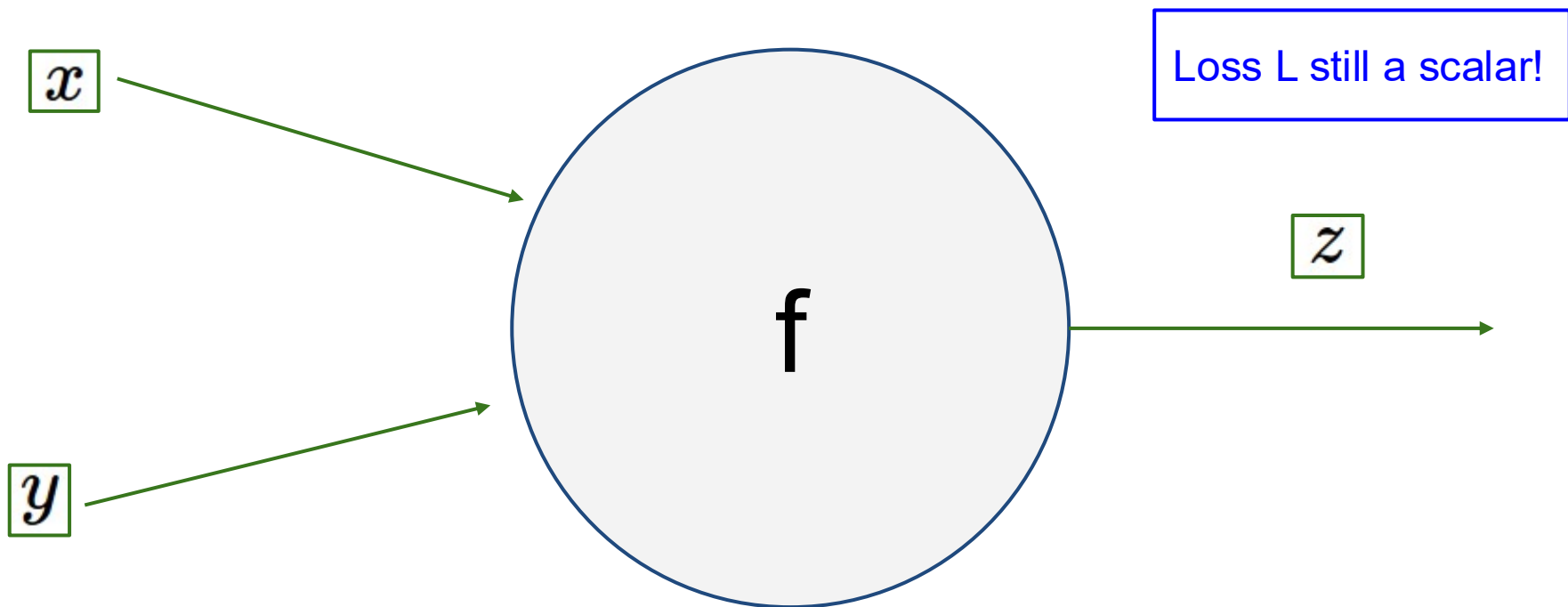


Jacobians

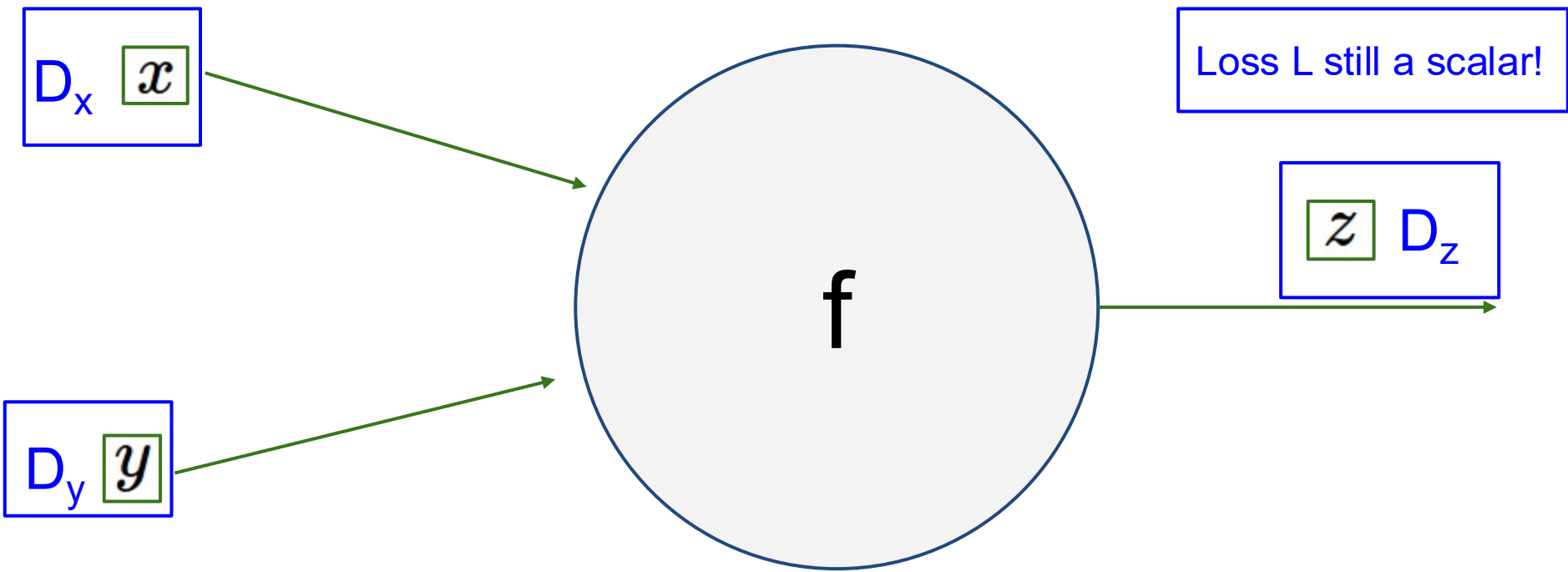
Given a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$, we have the Jacobian matrix $\mathbf{J}$ of shape $\mathbf{m} \times \mathbf{n}$, where $\mathbf{J}_{i,j} = \frac{\partial f_i}{\partial x_j}$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^T f_1 \\ \vdots \\ \nabla^T f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

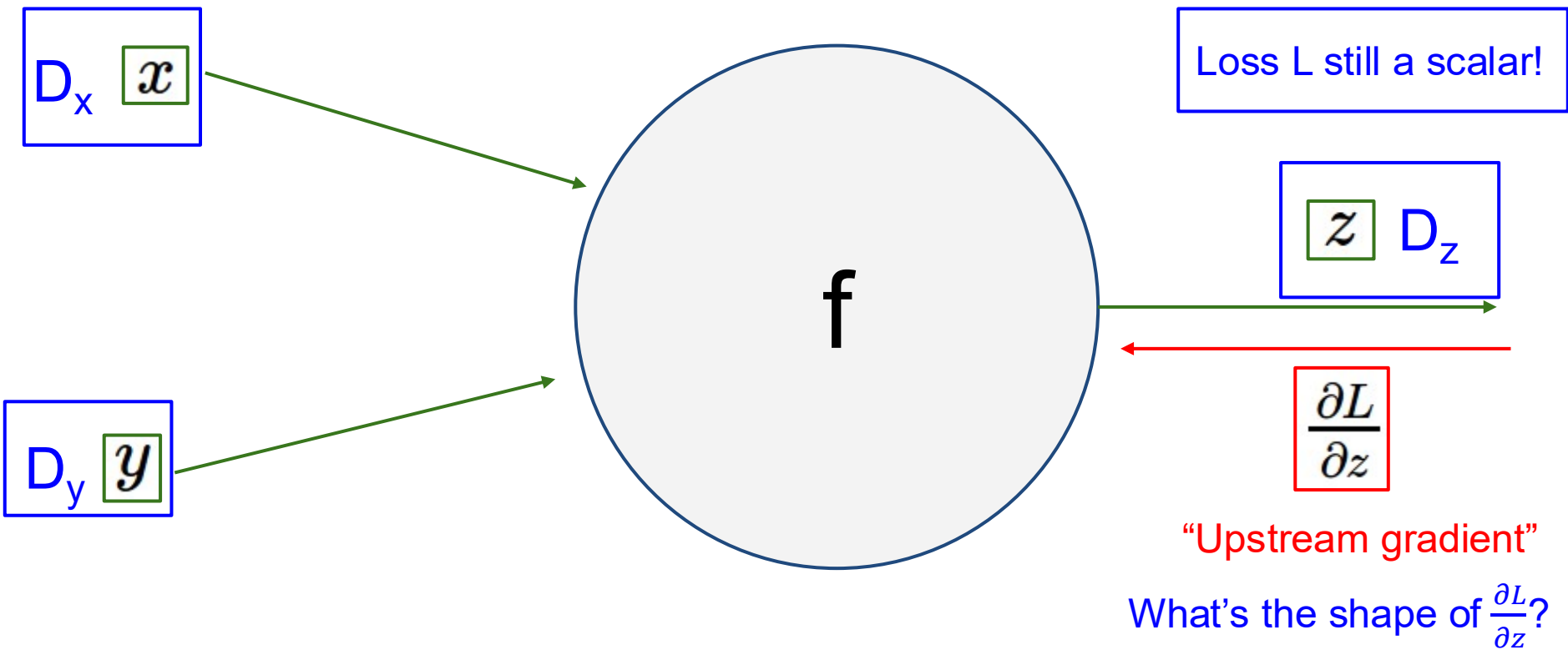
Backprop with Vectors



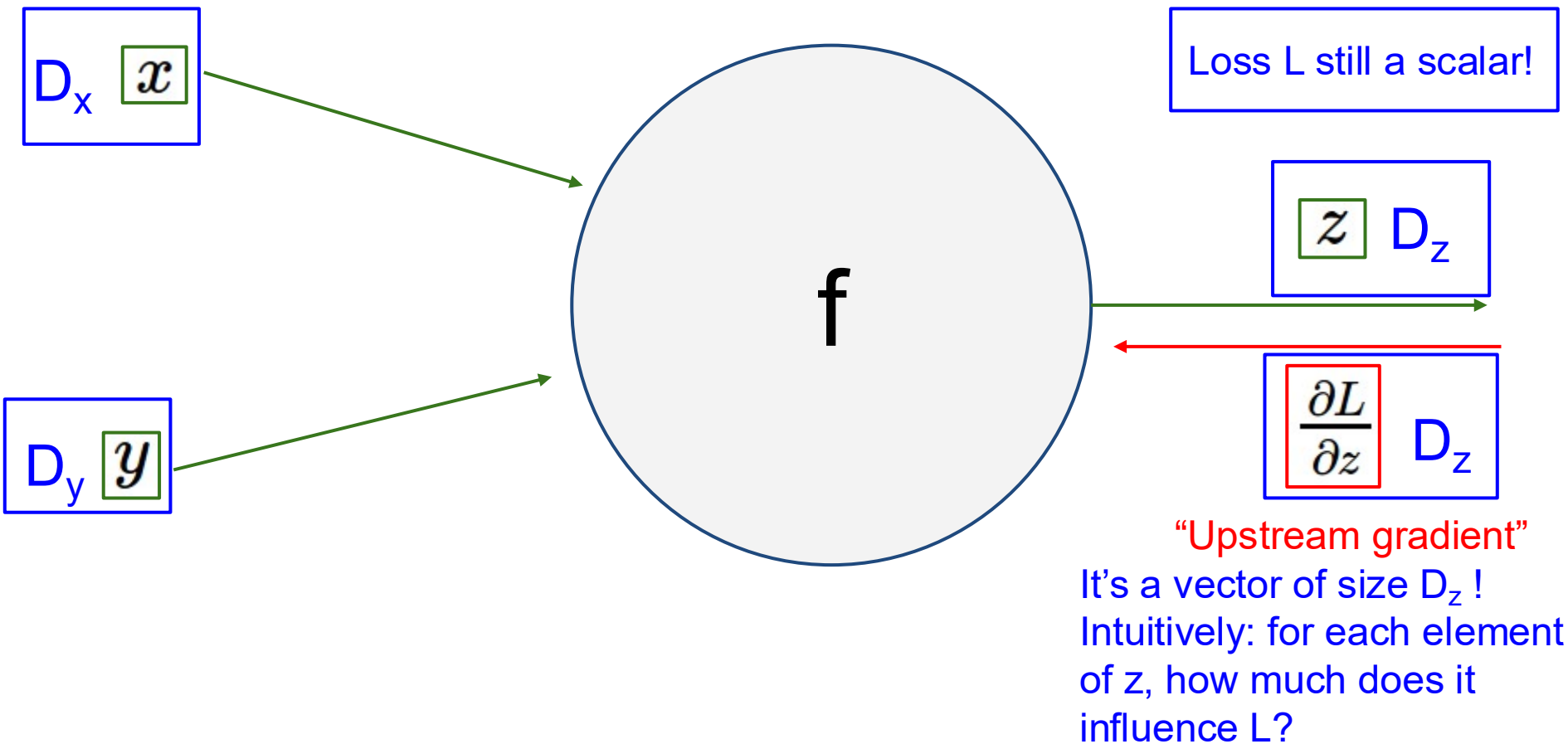
Backprop with Vectors



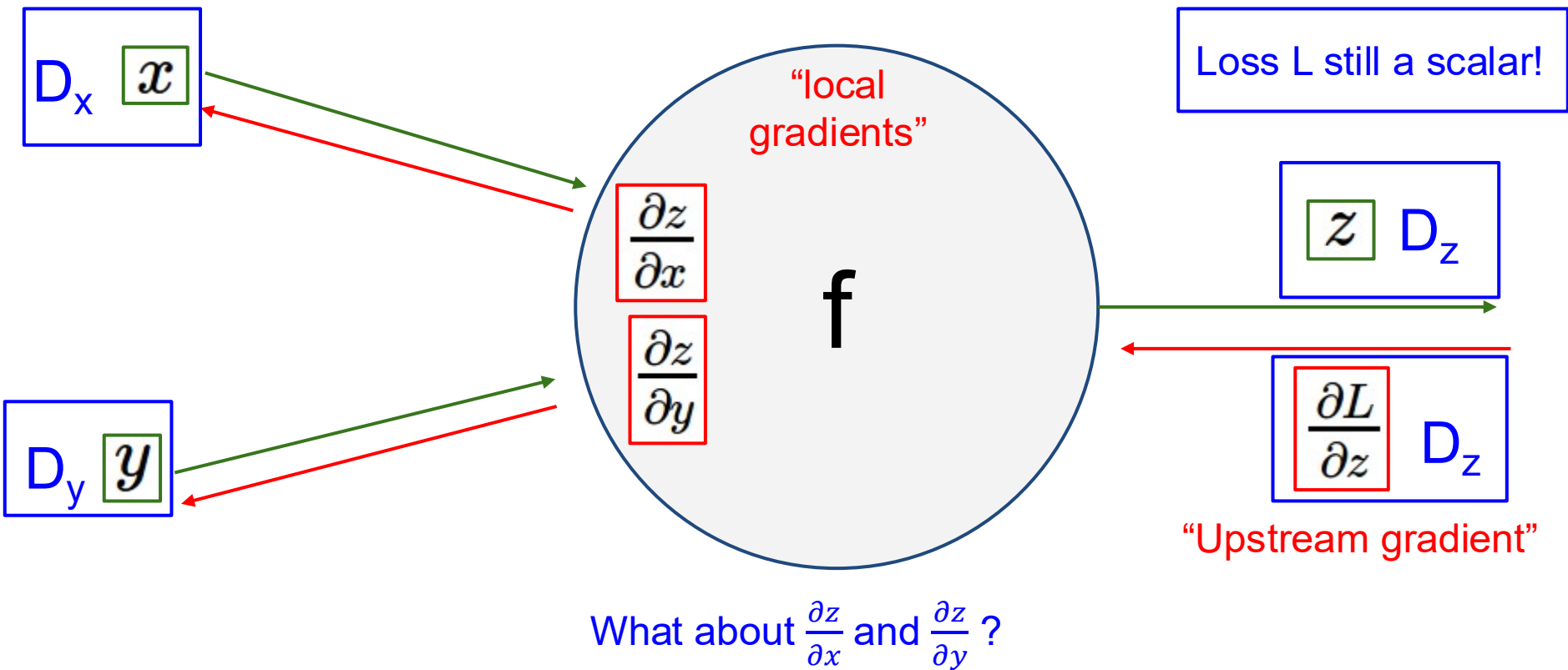
Backprop with Vectors



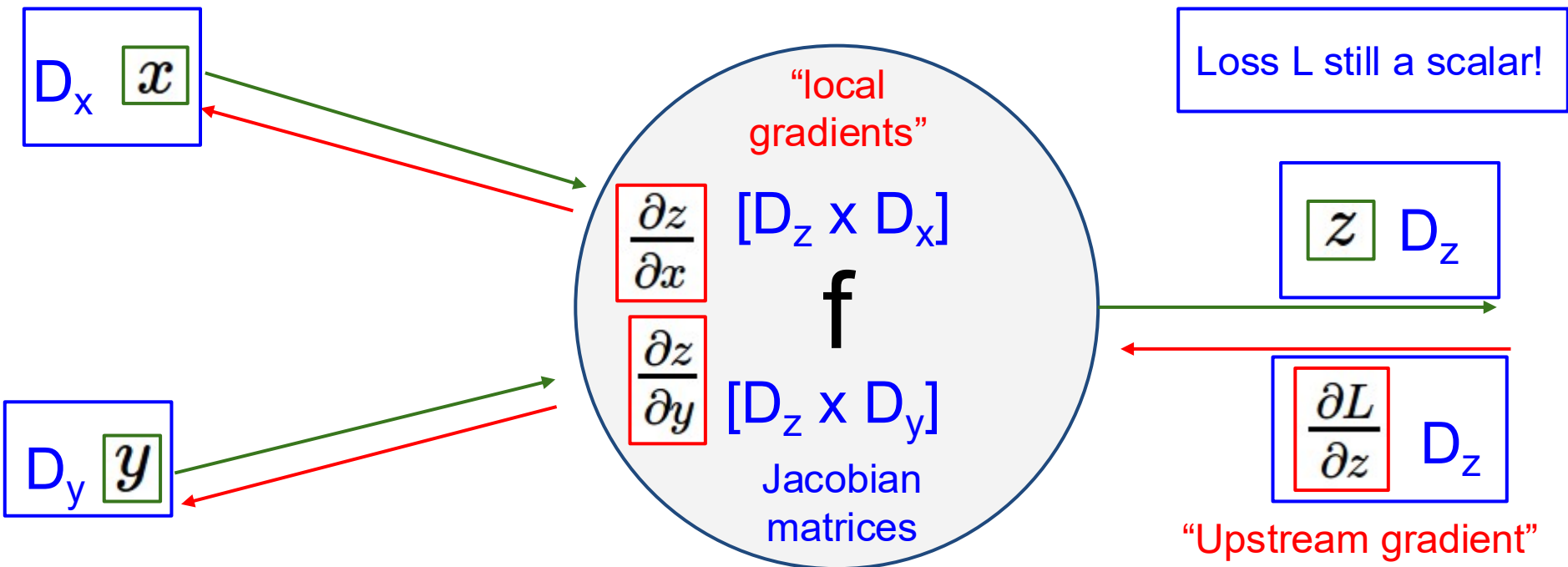
Backprop with Vectors



Backprop with Vectors



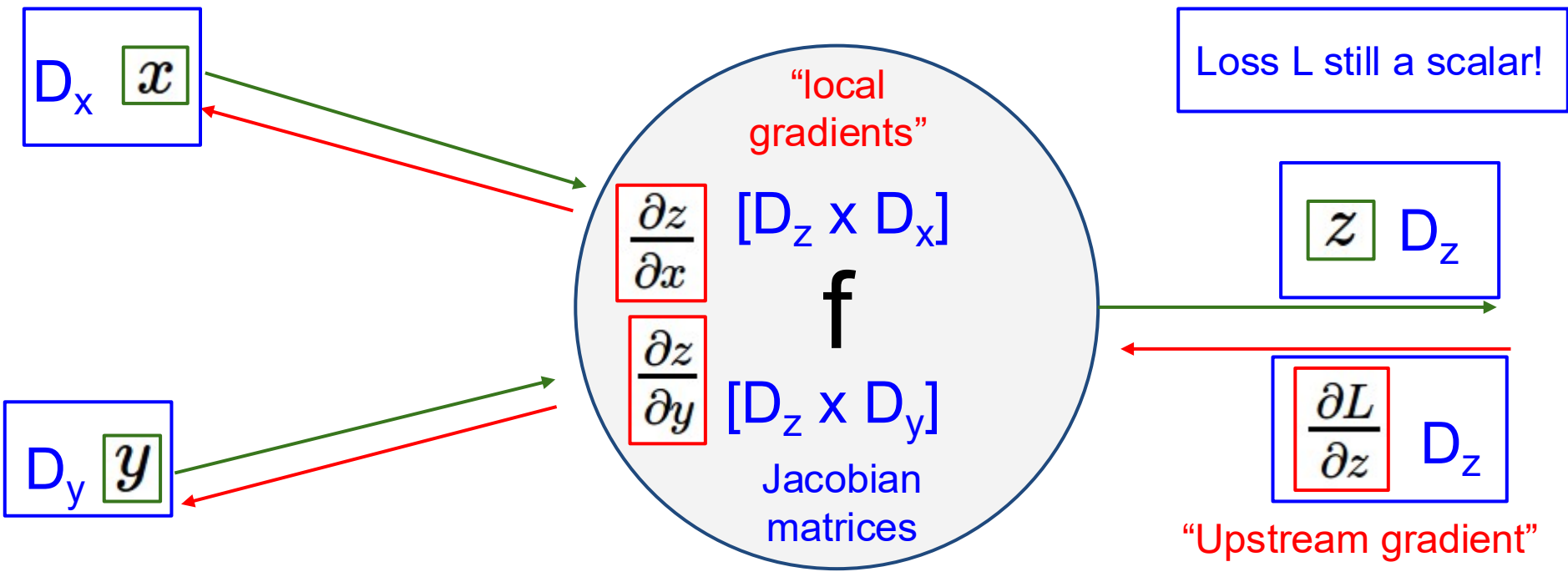
Backprop with Vectors



What about $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$?

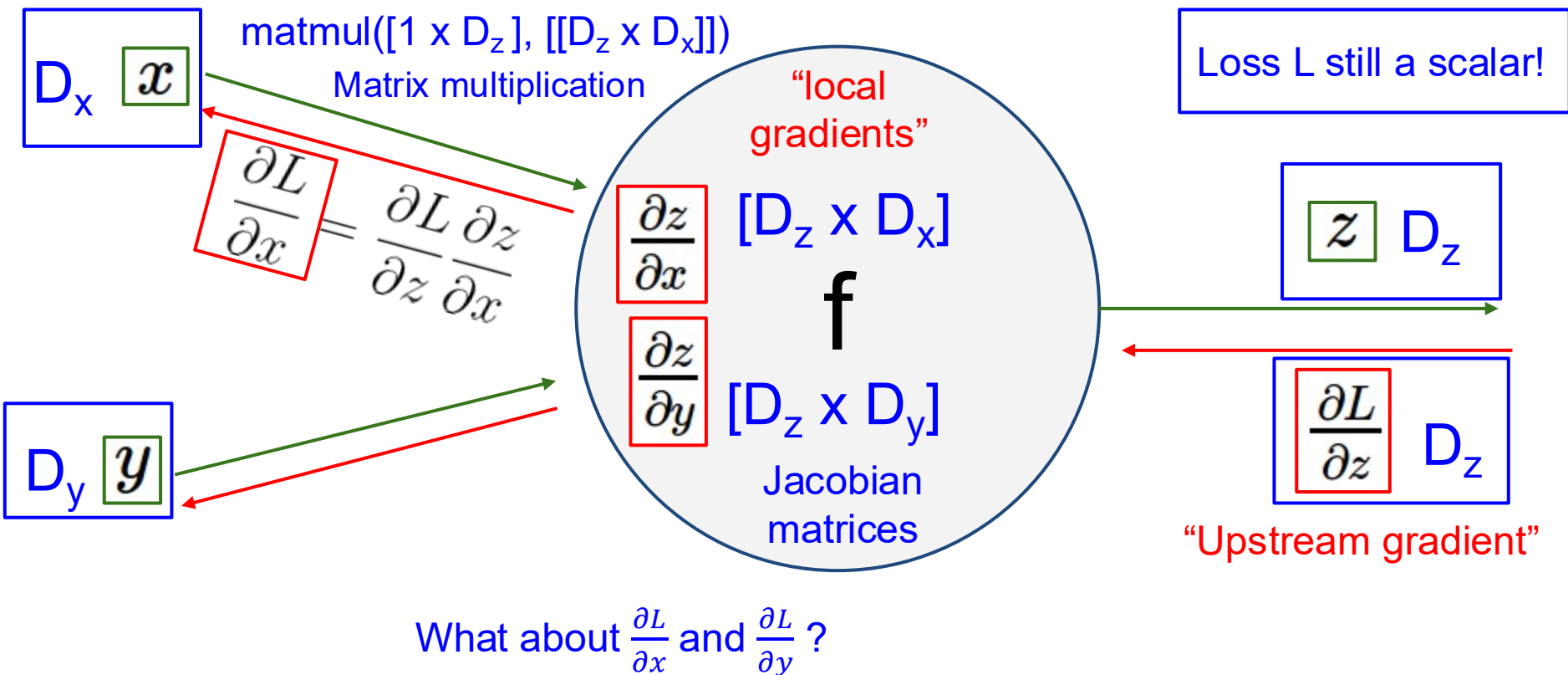
How much does each element in x influence each element in z

Backprop with Vectors



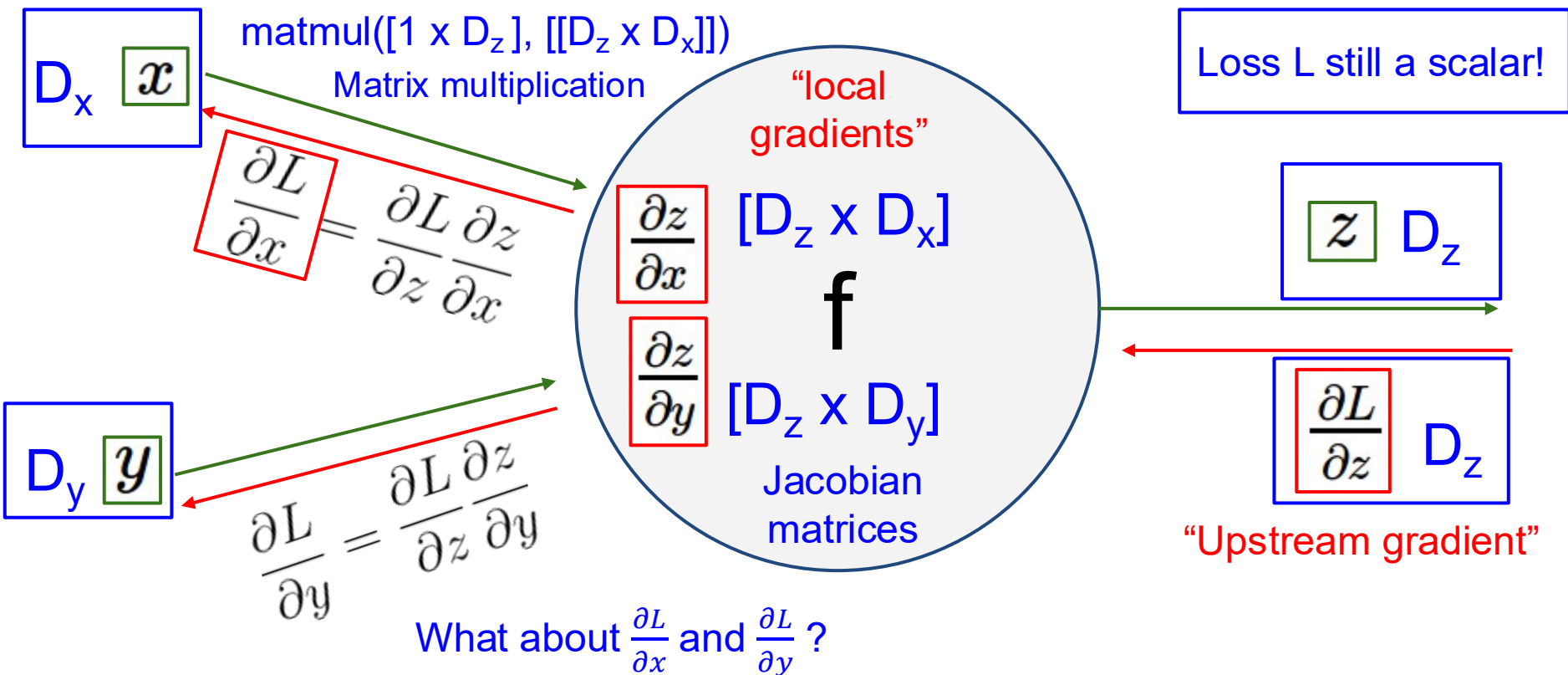
What about $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$?

Backprop with Vectors

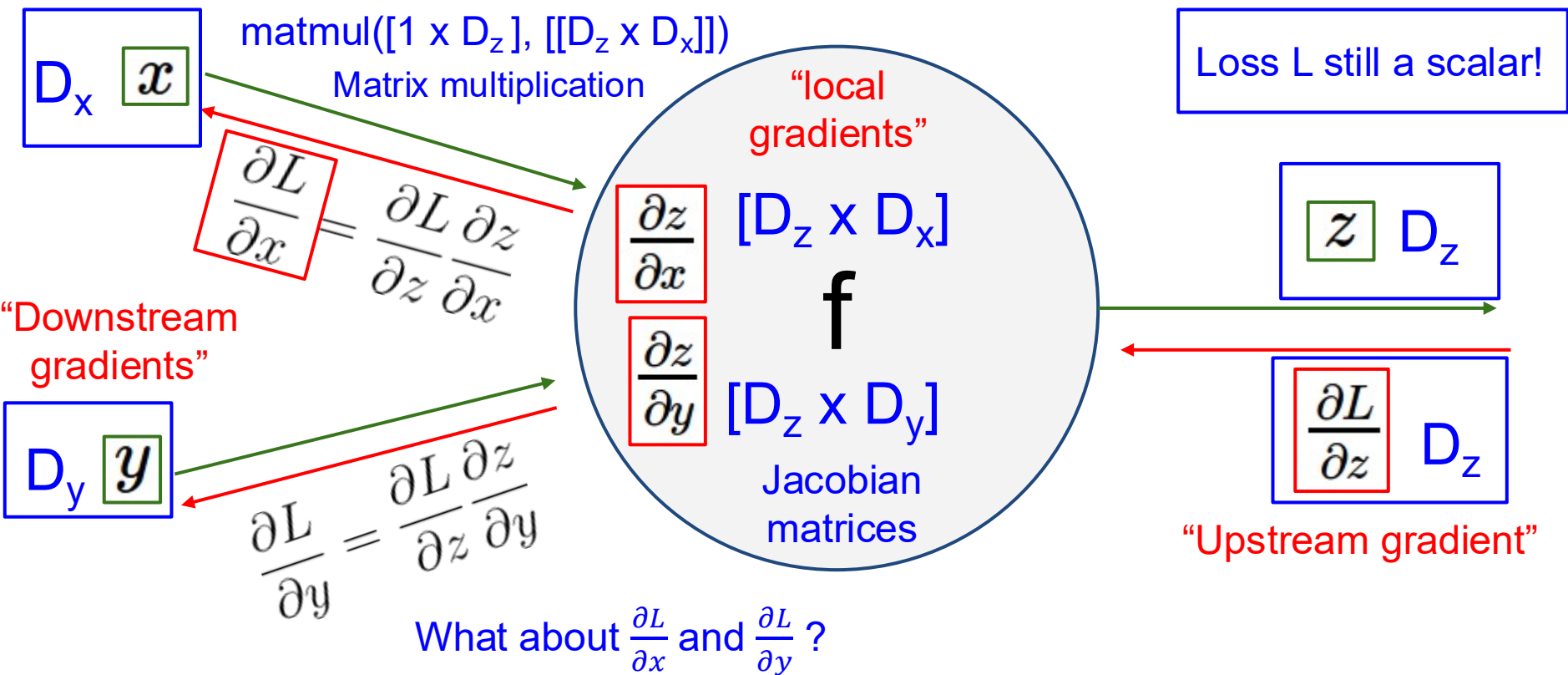


What about $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$?

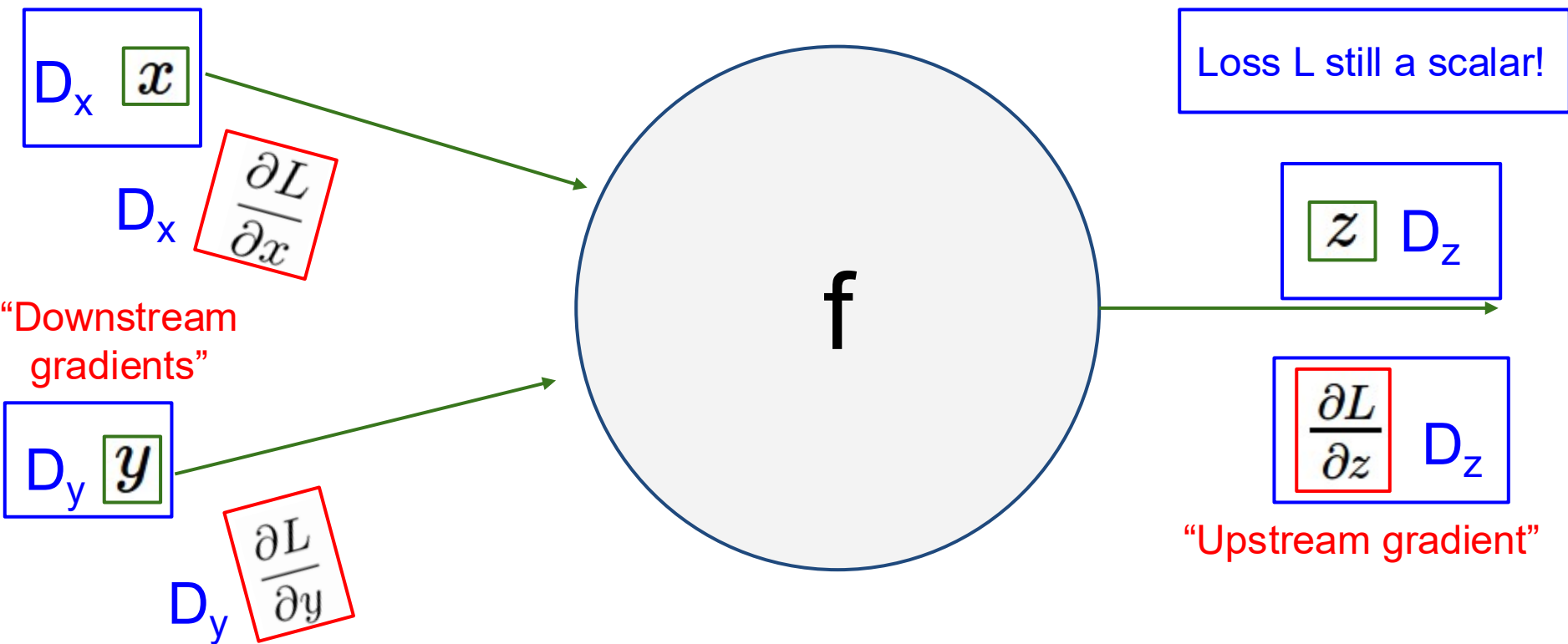
Backprop with Vectors



Backprop with Vectors



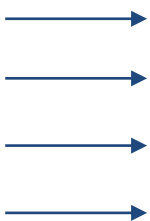
Gradients loss of wrt a variable have same dims as the original variable



Backprop with Vectors

4D input x:

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

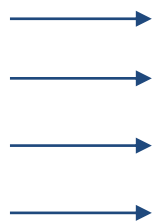


$$f(x) = \max(0, x)$$

(elementwise)

4D output z:

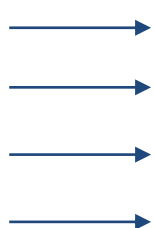
$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$



Backprop with Vectors

4D input x:

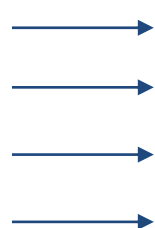
$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$



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

4D output z:

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

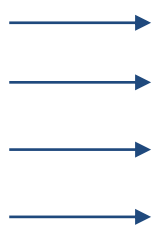


What does $\frac{\partial z}{\partial x}$ look like?

Backprop with Vectors

4D input x:

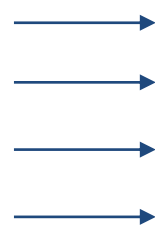
$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$



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

4D output z:

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



$[dz/dx]$

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

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

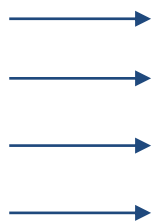
$\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$

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

Backprop with Vectors

4D input x:

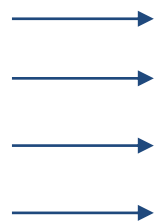
$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$



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

4D output z:

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



$[dL/dz]$

$[4 \ -1 \ 5 \ 9]$

$[dz/dx]$

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

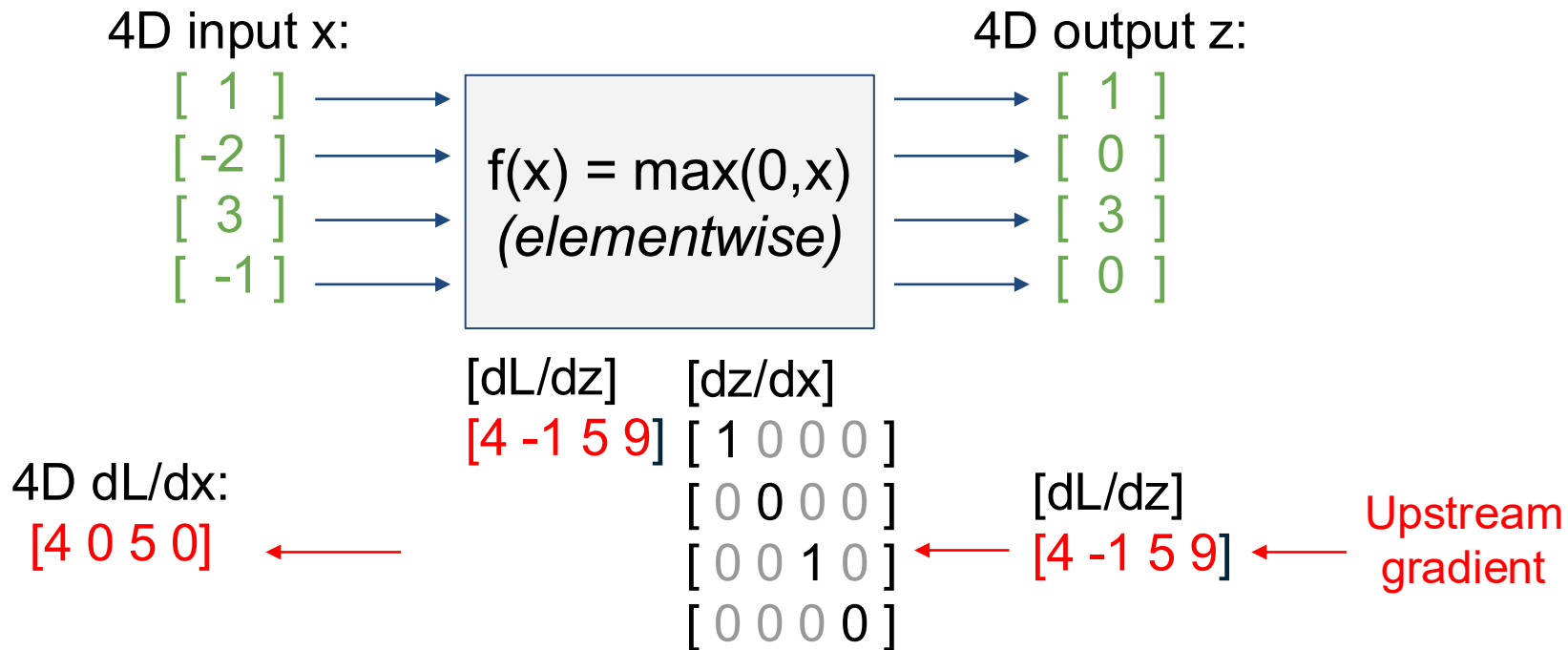
$[dL/dz]$

$[4 \ -1 \ 5 \ 9]$

Upstream
gradient



Backprop with Vectors



Backprop with Vectors

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

4D input x:

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

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

4D output z:

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

4D dL/dx :

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

$[dL/dz]$

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

$[dz/dx]$

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

$[dL/dz]$

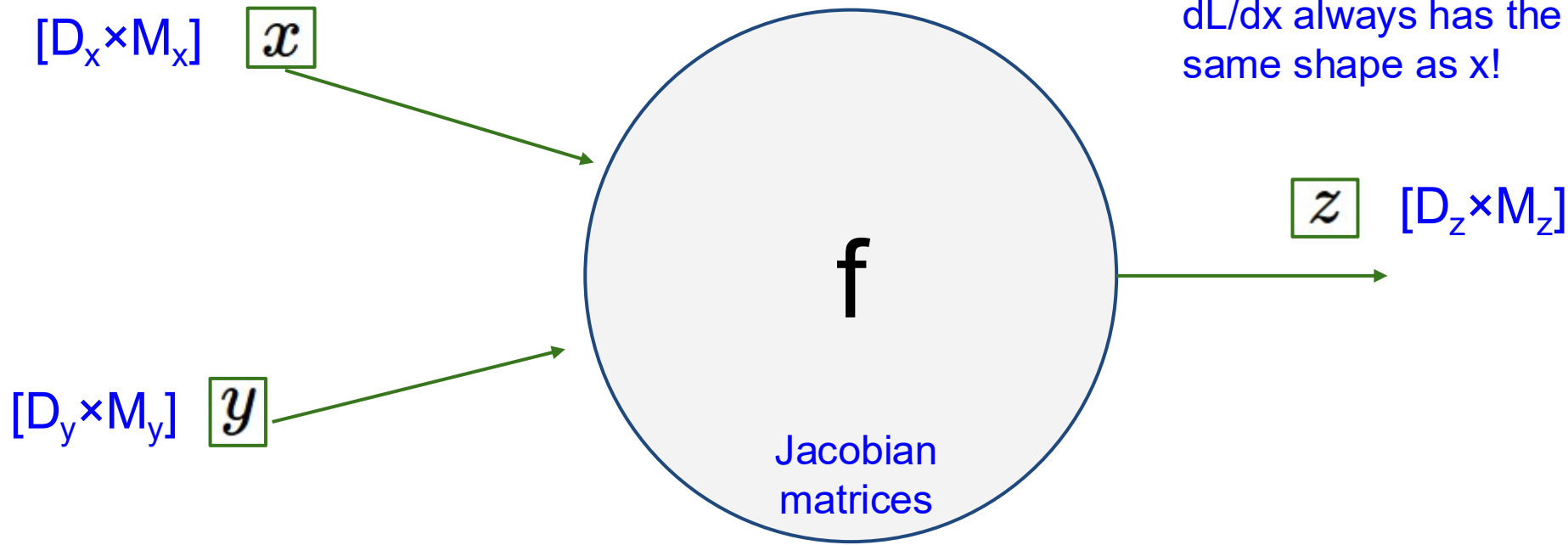
$\begin{bmatrix} 4 & -1 & 5 & 9 \end{bmatrix}$

Upstream gradient

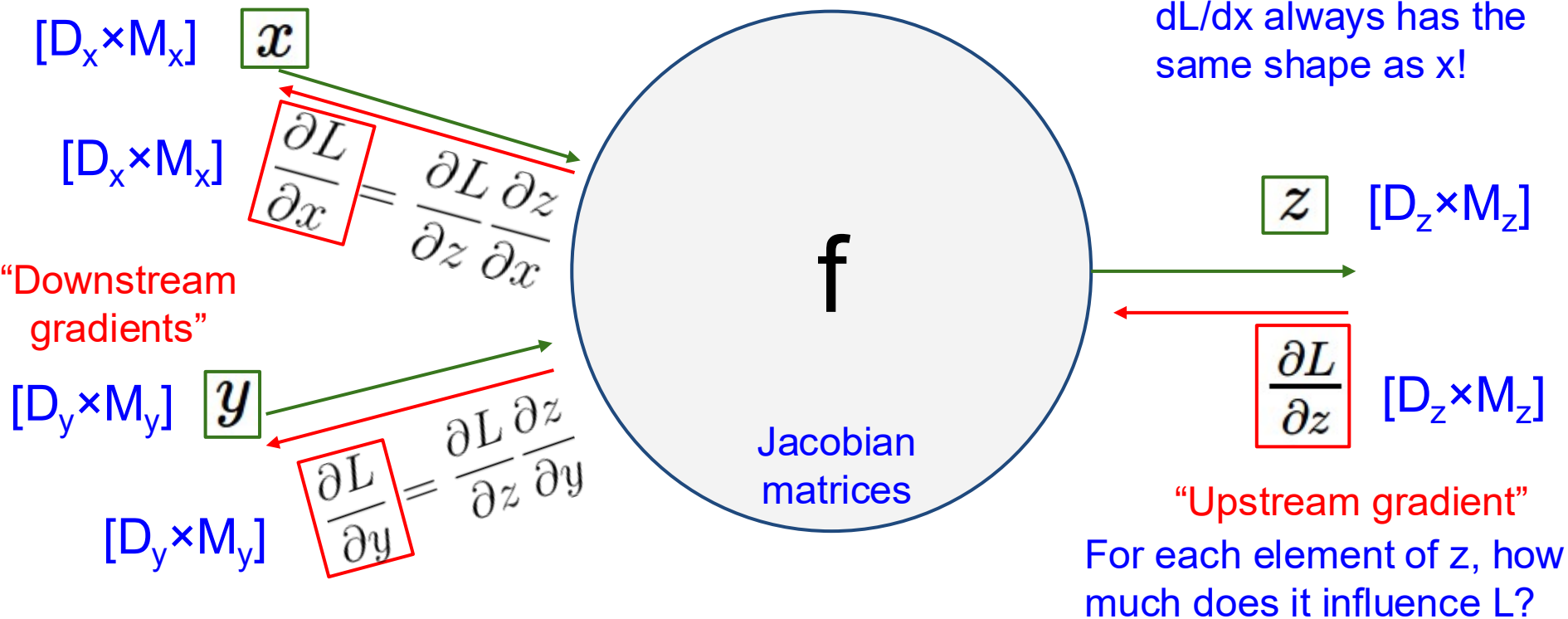
Backprop with Matrices (or Tensors)

Loss L still a scalar!

dL/dx always has the same shape as x !



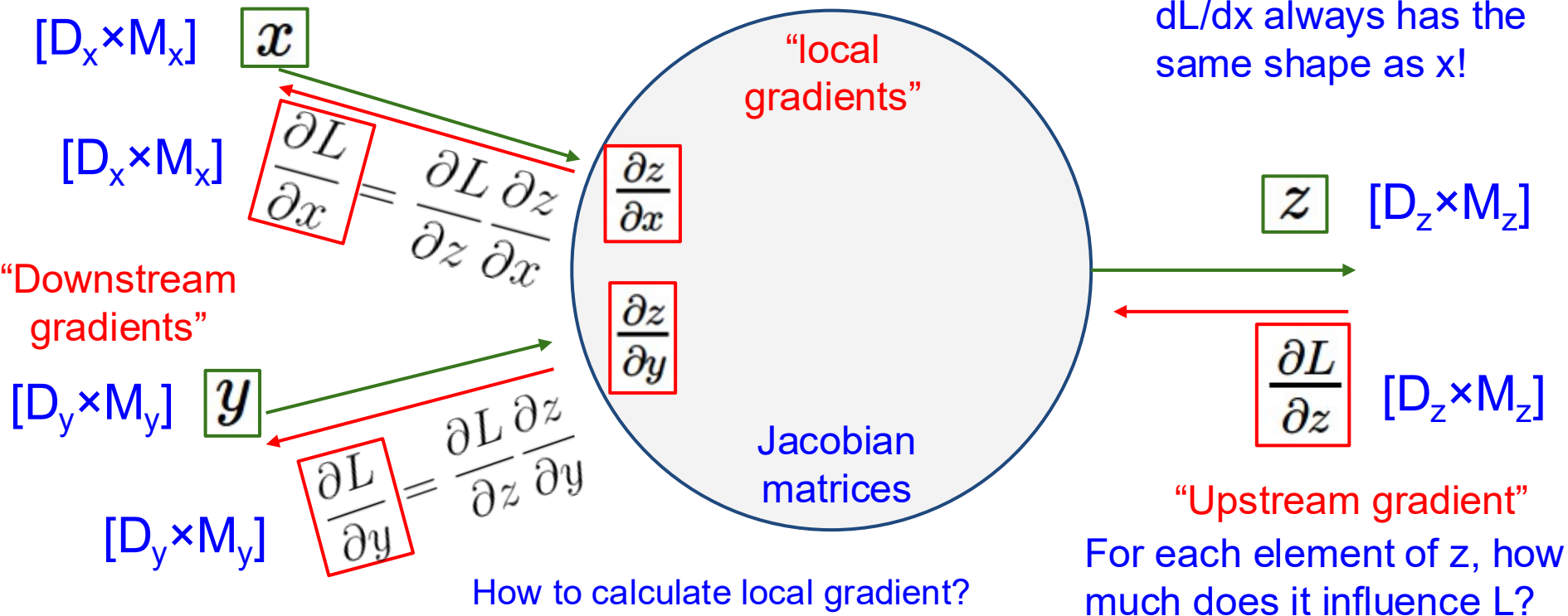
Backprop with Matrices (or Tensors)



Backprop with Matrices (or Tensors)

Loss L still a scalar!

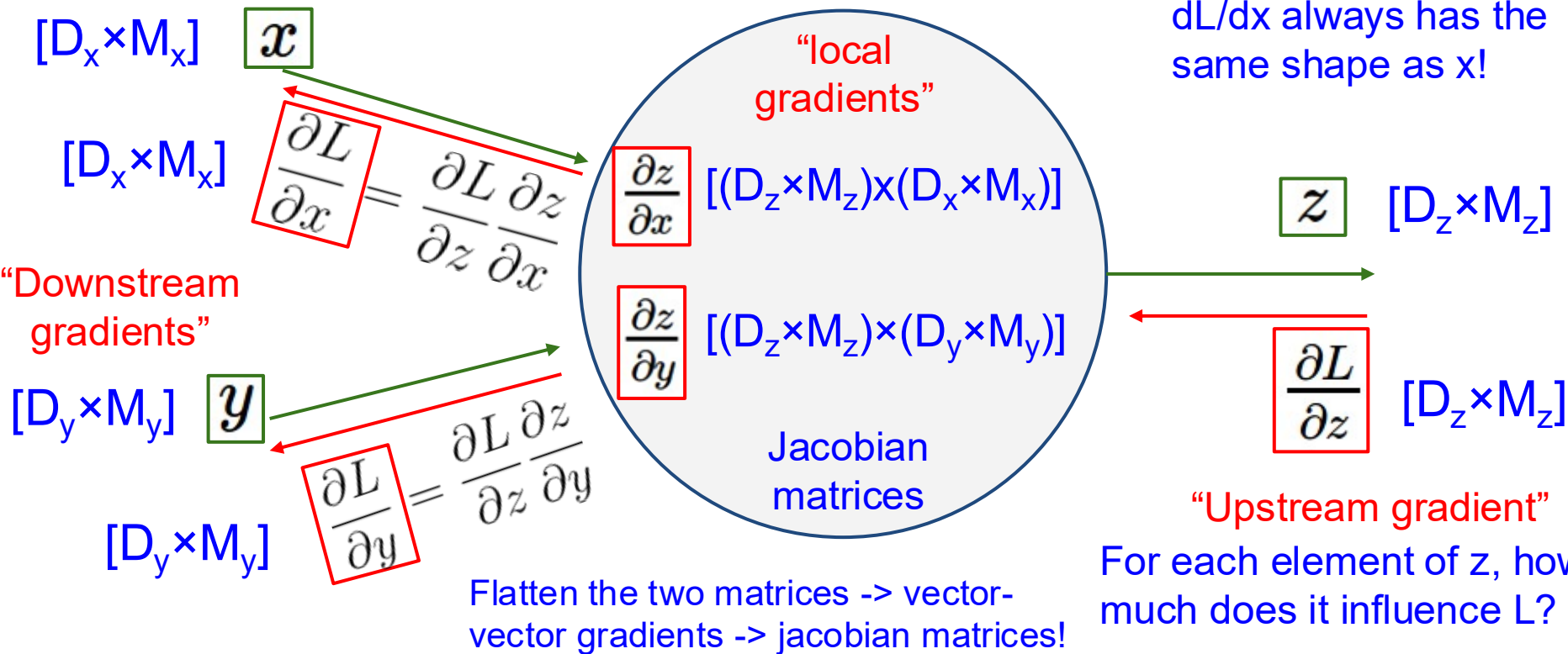
dL/dx always has the same shape as x !



Backprop with Matrices (or Tensors)

Loss L still a scalar!

dL/dx always has the same shape as x !



Backprop with Matrices

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]

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

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

Jacobians:

dy/dx: [(N×M)×(N×D)]

dy/dw: [(N×M)×(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]

What do the jacobian matrices look like?

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

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

Jacobians:

dy/dx: [(N×M)×(N×D)]

dy/dw: [(N×M)×(D×M)]

For a neural net with

N=64, D=M=4096

y: [N×M]

[13 9 -2 -6]

[5 2 17 1]

dL/dy: [N×M]

[2 3 -3 9]

[-8 1 4 6]

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

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

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

For a neural net with
N=64, D=M=4096
~68 billion numbers!
Each Jacobian takes 256 GB of memory!
Must exploit its sparsity!

y: [N×M]
[13 9 -2 -6]
[5 2 17 1]

dL/dy: [N×M]
[2 3 -3 9]
[-8 1 4 6]

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x}$$

Should not have to
compute this!

Backprop with Matrices

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

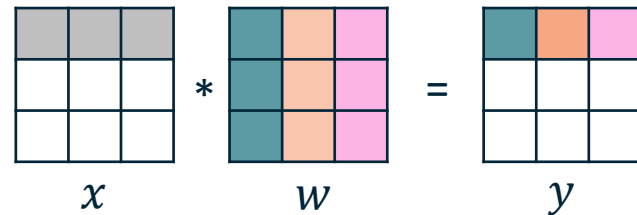
$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

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: Which part of y
 does a single element
 in x contribute to?

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



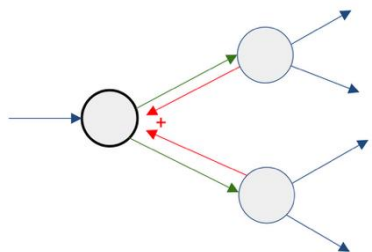
Backprop with Matrices

$$x: [N \times D]$$

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

$$w: [D \times M]$$

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



Matrix Multiply

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

$$y: [N \times M]$$

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

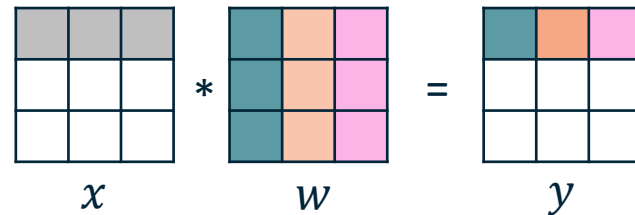
$$dL/dy: [N \times M]$$

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

Q: Which part of y does a single element in x contribute to?

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

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



We only care about the y 's that are involved in the computation!

Recall the branching gradient rule!

Backprop with Matrices

$$x: [N \times D]$$

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

$$w: [D \times M]$$

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

Matrix Multiply

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

$$y: [N \times M]$$

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

$$dL/dy: [N \times M]$$

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

Q: Which part of y does a single element in x contribute to?

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

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \underbrace{\frac{\partial L}{\partial y_{n,m}}}_{\text{Upstream gradient}} \underbrace{\frac{\partial y_{n,m}}{\partial x_{n,d}}}_{\text{local gradient}}$$

Upstream gradient local gradient

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

dL/dy: [N×M]
 $\begin{bmatrix} \boxed{2} & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$

Q: Which part of y does a single element in x contribute to?

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

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

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

How do we calculate this?

Backprop with Matrices

$$x: [N \times D]$$

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

$$w: [D \times M]$$

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

Matrix Multiply

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

$$y: [N \times M]$$

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

$$dL/dy: [N \times M]$$

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

Q: Which part of y does a single element in x contribute to?

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

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

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

How do we calculate this?

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

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

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

dL/dy: [N×M]
 $\begin{bmatrix} \boxed{2} & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$

Q: Which part of y does a single element in x contribute to?

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

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

A: $w_{d,m}$

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

$\nearrow$
 $w_{d,m}$

Backprop with Matrices

$$x: [N \times D]$$

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

$$w: [D \times M]$$

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

Matrix Multiply

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

$$y: [N \times M]$$

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

$$dL/dy: [N \times M]$$

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

Q: Which part of y does a single element in x contribute to?

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

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

A: $w_{d,m}$

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

Just a dot product!

Backprop with Matrices

$$x: [N \times D]$$

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

$$w: [D \times M]$$

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

Matrix Multiply

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

$$y: [N \times M]$$

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

$$dL/dy: [N \times M]$$

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

Q: Which part of y does a single element in x contribute to?

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

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

A: $w_{d,m}$

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

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

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

Just a matrix multiplication
No jacobian matrix needed!

Backprop with Matrices

$$x: [N \times D]$$

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

$$w: [D \times M]$$

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

Matrix Multiply

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

$$y: [N \times M]$$

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

$$dL/dy: [N \times M]$$

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

By similar logic:

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

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

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

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

Backprop with Matrices

$$x: [N \times D]$$

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

$$w: [D \times M]$$

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

Matrix Multiply

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

$$y: [N \times M]$$

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

$$dL/dy: [N \times M]$$

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

By similar logic:

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

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

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

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

For a neural net layer with
 $N=64$, $D=M=4096$
 The large matrix (W) takes
 up to 0.13 GB memory

Backprop with Matrices: Summary

- Most neural network layers are tensor-in & tensor-out, with large Jacobian matrices
- When calculating $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial x}$, we almost never compute the Jacobian matrix $\frac{\partial f}{\partial x}$.
- Instead, we exploit the **sparsity** of the Jacobian to derive **simplified analytical gradient** of loss w.r.t to the input / weights.

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

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f} \odot (x > 0)$$

$$\text{Matmul: } f(x) = xw$$

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

Summary:

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

Next Time: Convolution