BACKPROPAGATION

→ Computation Graphy

→ Why

→ Good data structure for computing gradients

→ Node → Variable (could be scalar, vector, matrix or operations etc)

→ Variables can feed values to operations

Edge → Func^n arguments
data dependency

→ Variables: Parameters of operation
Eg: $Wx + b$ x = Input
W = Variable

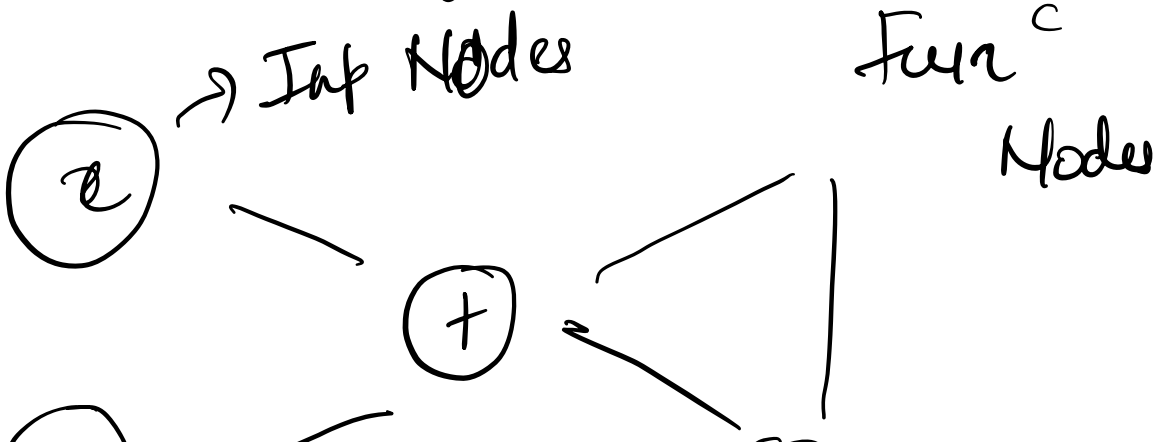
→ operations – Compute = Output given cput

list of inp = which can be nodes variable or other ops

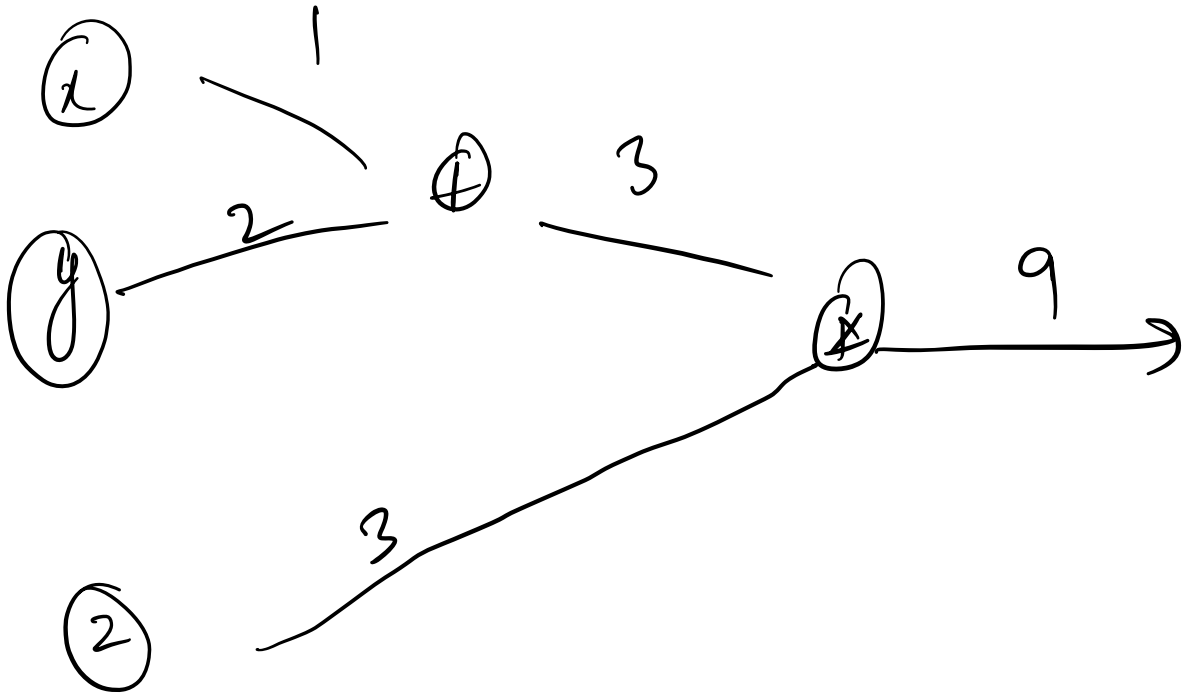
list of consumers =

Eg: $f(x) = (x+y)*2$

Eg: $(x+y)*2$



⇒ Values flow from edges,

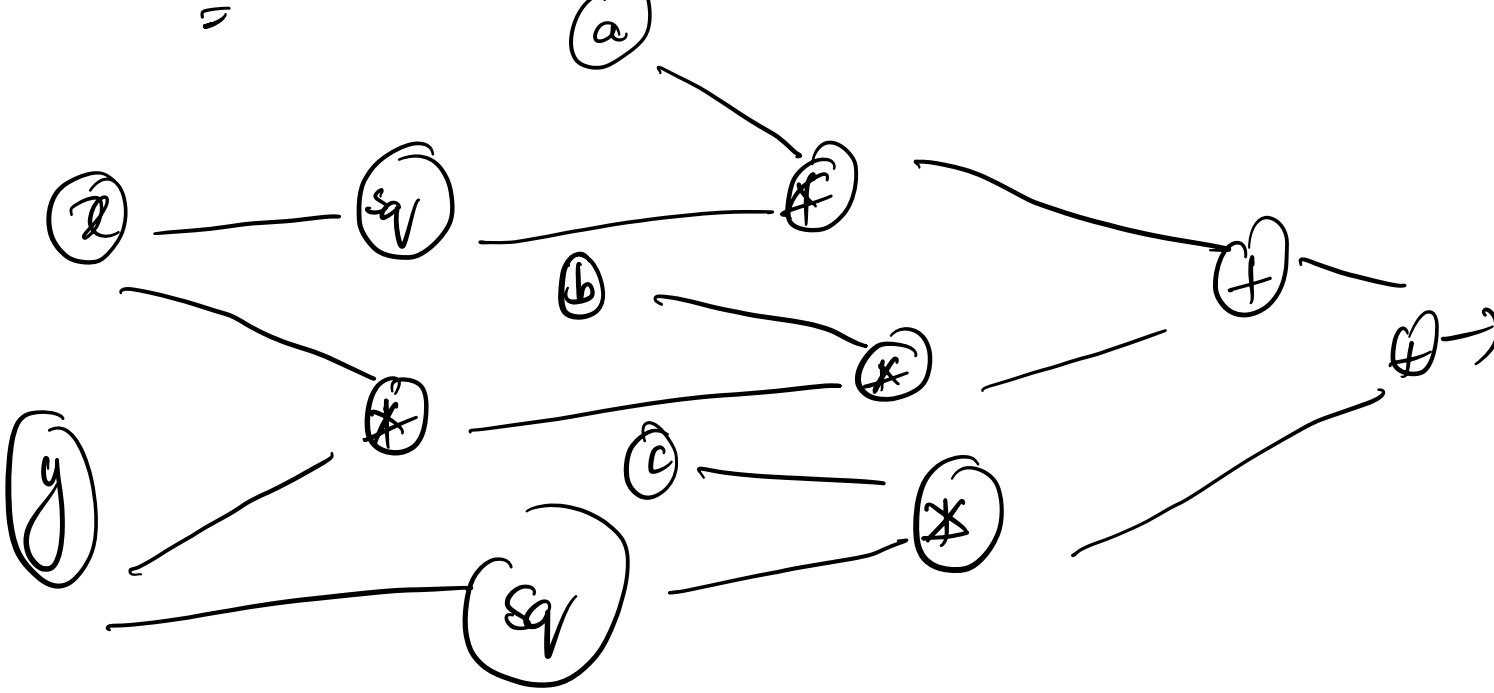


Neural Nets as Computation graph

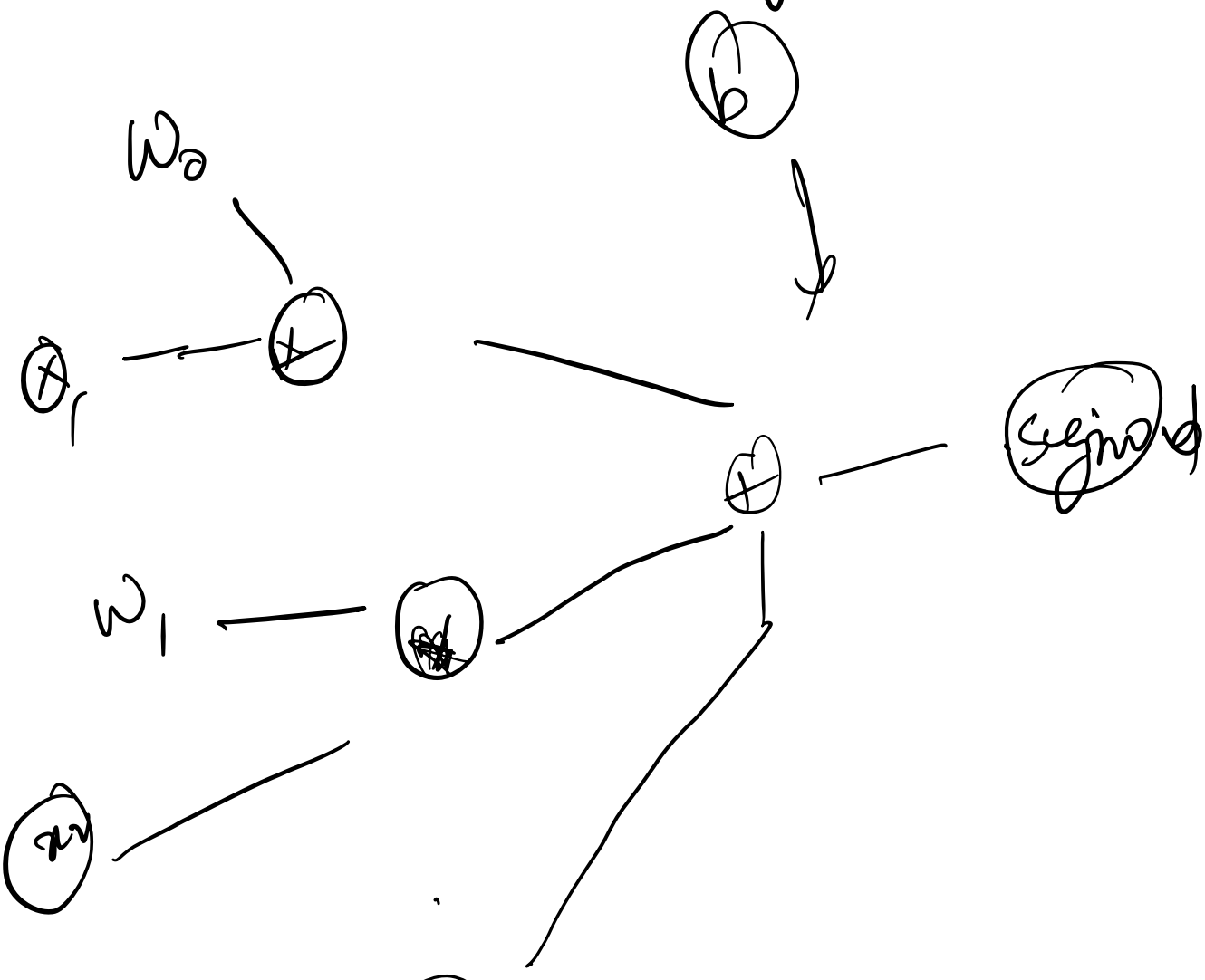
Eg: $ax^2 + bxy + cx^2$

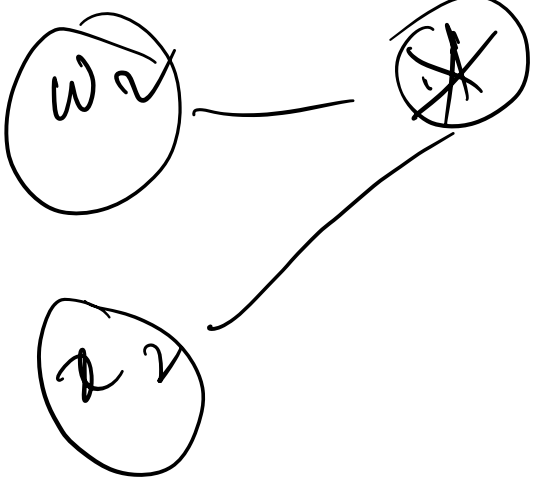
a, b, c = weights

x, y = Inputs



Preptrons Graph





→ Why we need Con Graph?

→ Deriving grads for deep net works everytime is cumbersome, relationship is easy with graphs.

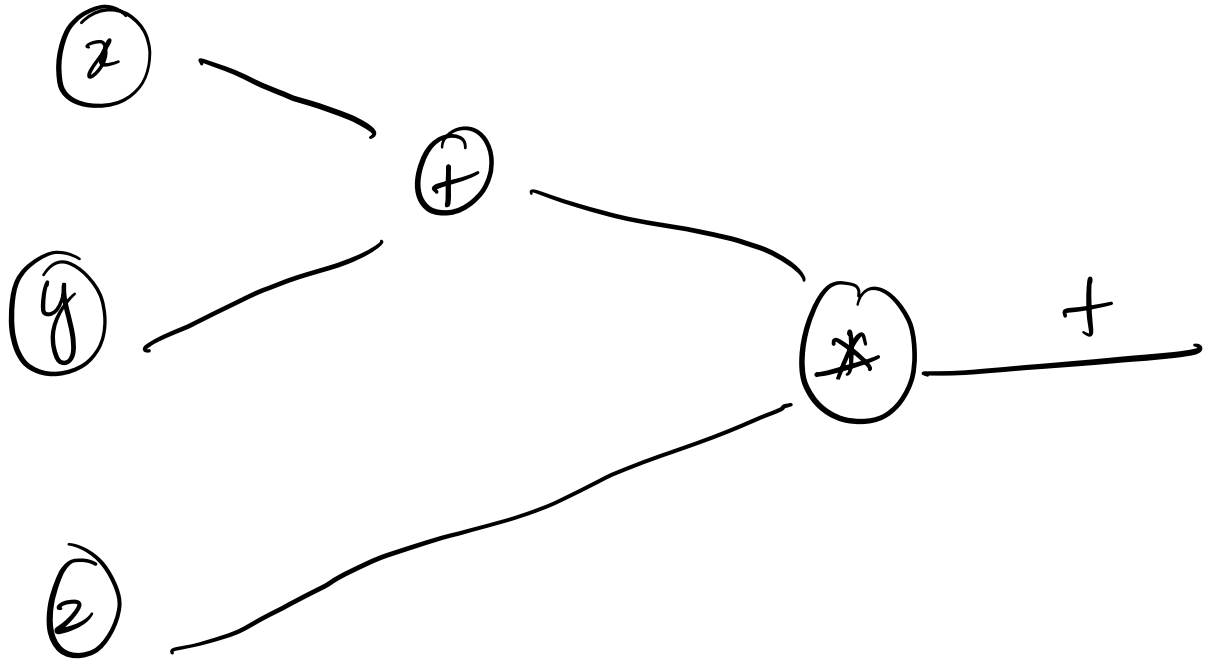
→ Data structure aligns well with back prop and provides modularity.

→ Graph traversal algorithms can help us in calculating grads easily

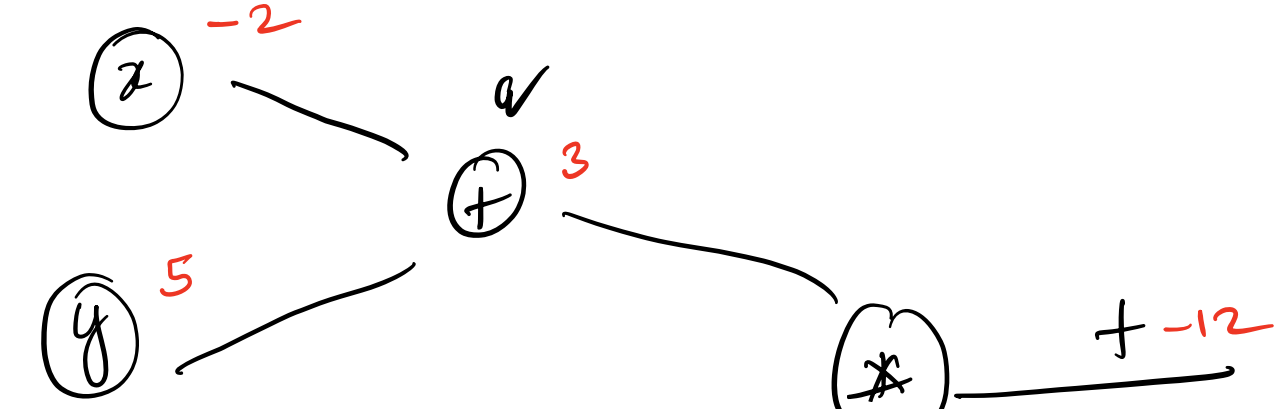
Backprop Simple Example

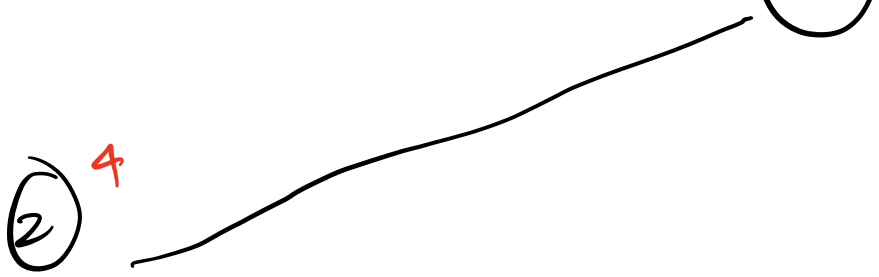
- Lets take

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



① Forward pass





⇒ left to right

⇒ Compute output values.

② Backward pass

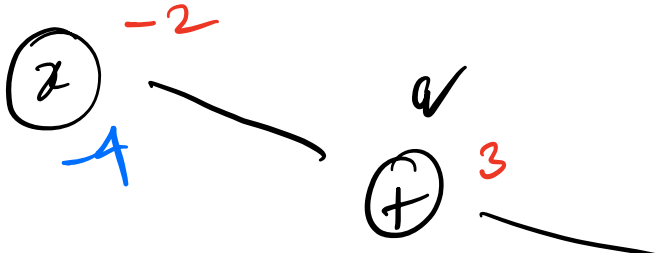
→ Want to compute grads w.r.t each of input & output

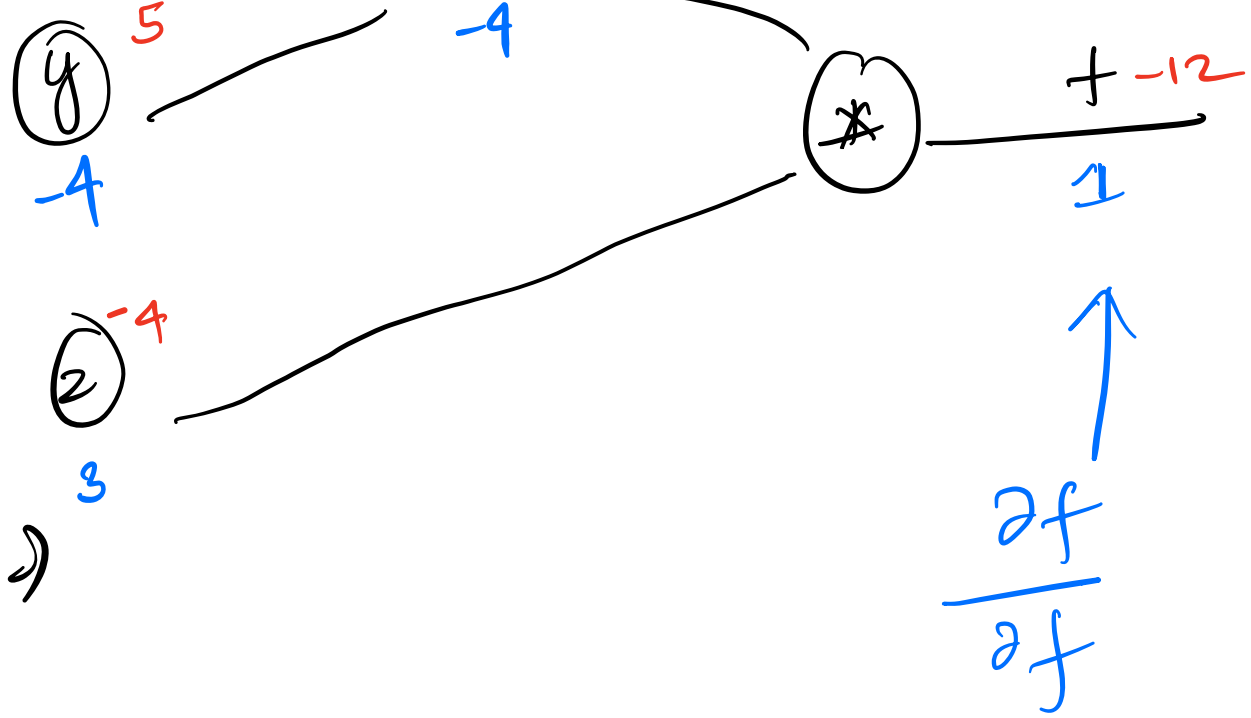
i.e

$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

→ Right to left

Up → Values
Down → grads





$q = x + y \; ; \; f = qz$

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

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

How

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

since y is not directly connected

to f

Downstream gradient → this is value of derivative to be computed

Local gradient → local effect of y affecting intermediate value

Upstream gradient → how much output of this piece of graph, affects final output

$$q = x + y$$

$$\frac{\partial f}{\partial q} = 1 = \text{local derivative}$$

$$\frac{\partial f}{\partial y} = 1 \times -4 = -4$$

→ See so easy

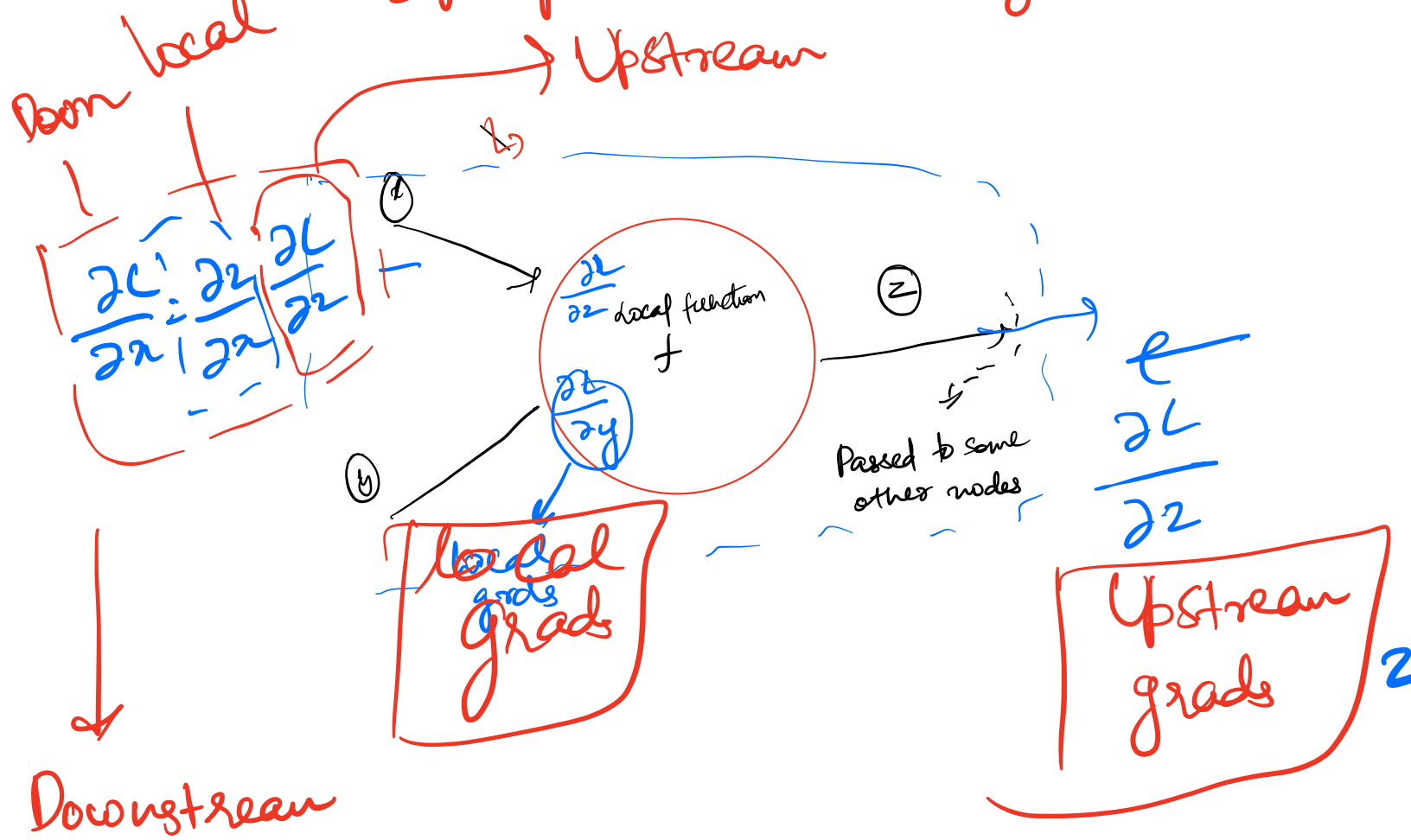
→ Just do forward
and
then do backward

→ This way of thinking about computation graph is very modular

ZOOM IN ON NODE

→ Each node doesn't need to know or care about rest of the graph.

→ Just do local computation within each node, and by aggregating calculate global der.



grads

→ Output passed to some other nodes

→ Eventually loss is computed is this

→ Eventually backprop passes some grad w.r.t this output to this node

→ Local grads: How much each output gets affected by each input of node

→ Again downstream are passed along to other nodes in graph

→ At the end we would have traversed graph entirely by

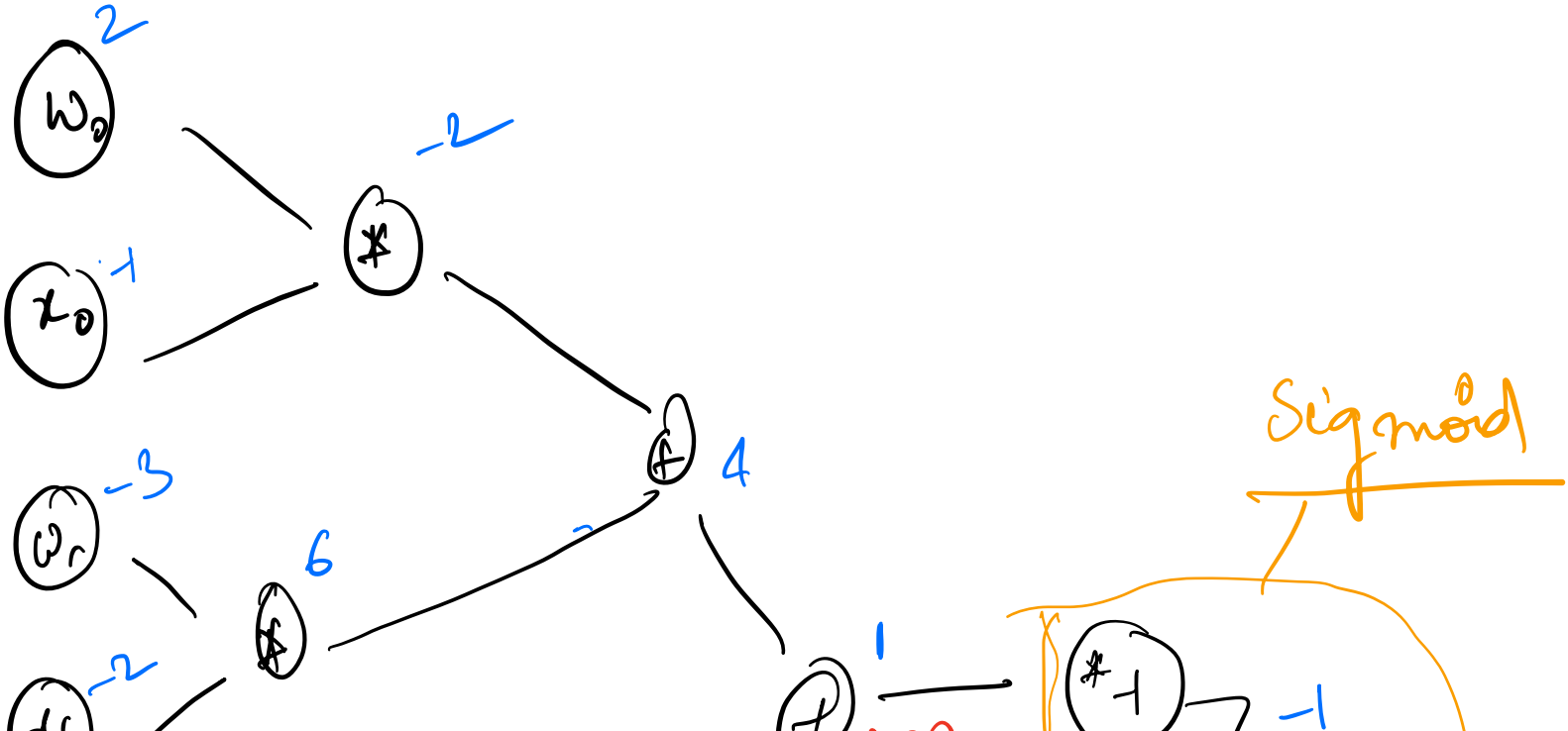
backwards, computing the grads entirely.

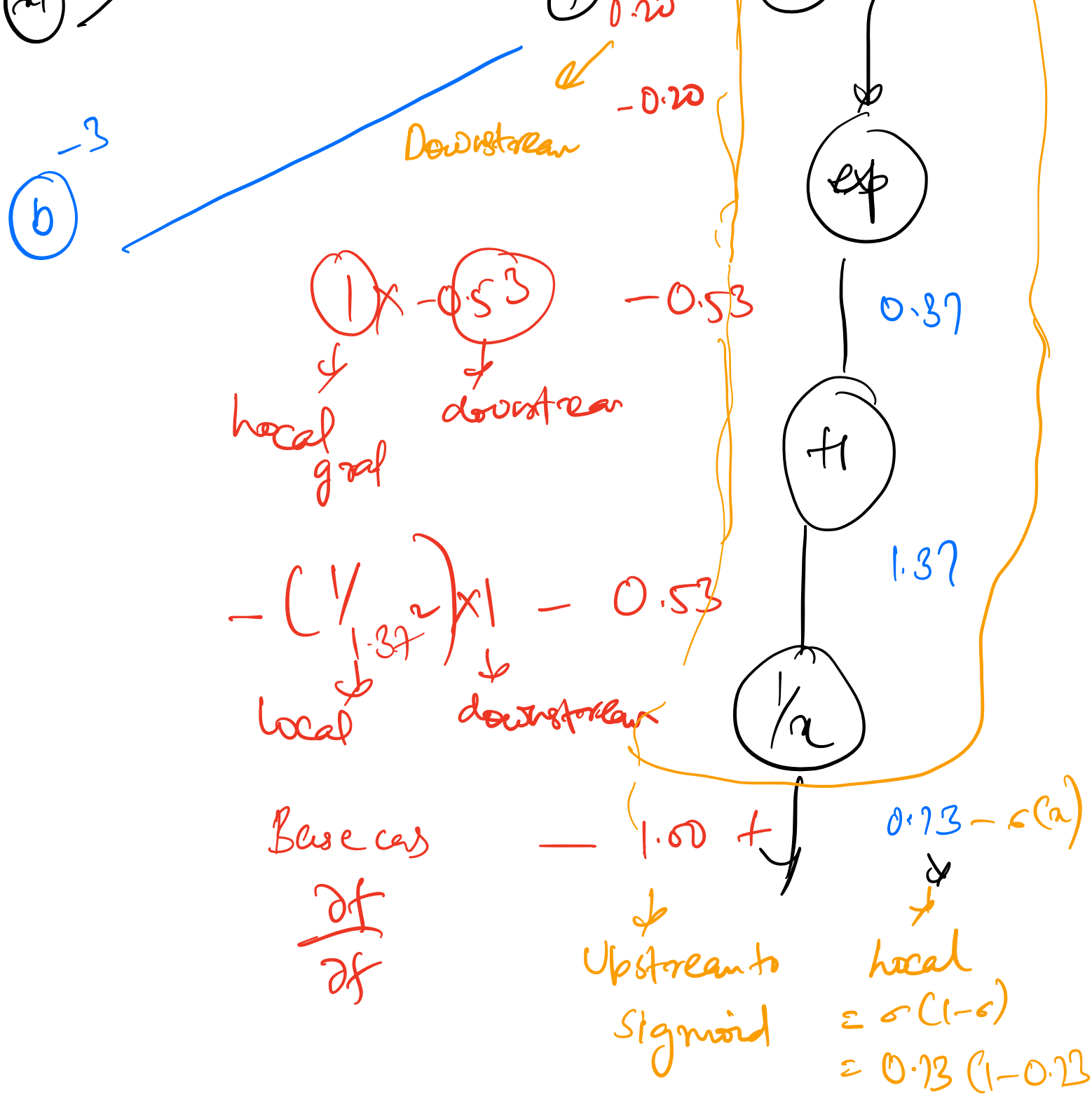
→ So GRAPH is good data structure

ANOTHER EXAMPLE

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

↑
looks familiar??





→ There are multiple ways to pick the pieces of comp uk graph.

→ Type of primitives can be defined on our own to be more complicated

→ Eg: Sigmoid shows up most of the time in NN

→ So picking this as primitive would be easy to have as building block

Local grad of σ

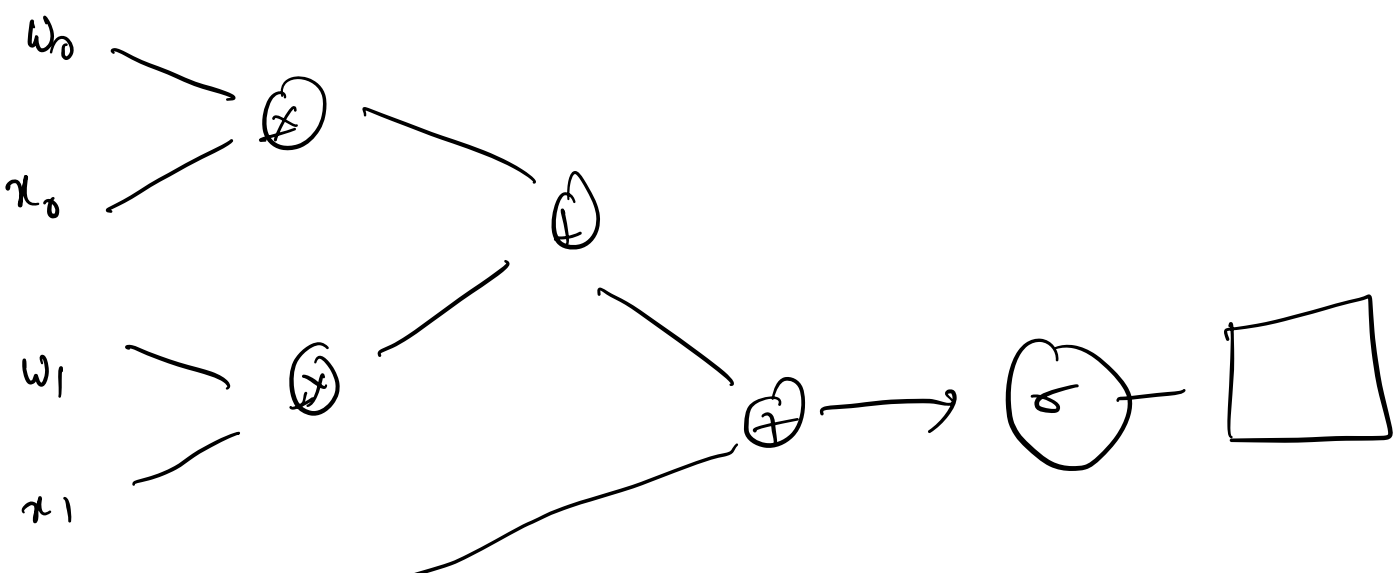
$$\frac{\partial f(x)}{\partial x} = \left(1 - \sigma(x)\right)\left(\sigma(x)\right)$$

→ So

$$0.2 = (1 - 0.73) \times 0.73 \times 1 = 0.2$$

→ Defining more complex primitive gives more efficiency and semantic meaning

⇒ Implementation "flat" code



b

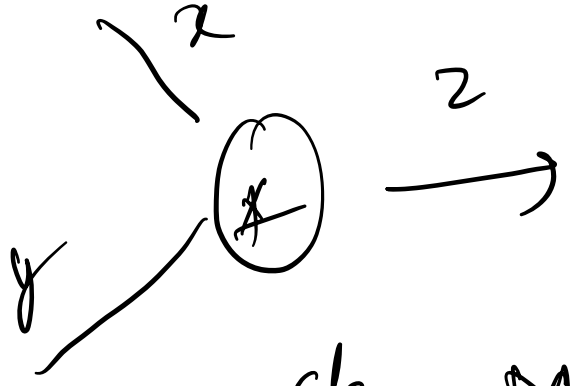
```
def f(w0, x0, w1, x1, w2)
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)

Back
    grad_L = 1.0
    grads3 = grad_L * (1-L) * L
    grad w2 = grad s3
    grad s2 = grad s3
    grad s0 = grad s2
    grad s1 = grad s2
```

2nd Way Modular API

```
class ComputGraph
    def forward():
    def backward():
```

Eg:




```
class Multiply
    def forward(self, x, y)
```

ctx.save(x, y) — Need the in backward

z = x × y

return z

Upstream grad

def backward(ctx, grad_z):

x, y = ctx.get(x, y)

Local grads

grad_x = y × grad_z

grad_y = x × grad_z

return grad_x, grad_y

⇒ What about Vectors, Sir??

→ Vector valued functions

→ Derivative is Jacobian

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

$x \in R^N$

$y \in R^M$

$$\left(\frac{\partial y}{\partial x}\right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

→ Backprop with Vectors

→ We know, we can just focus on one node

→ And then everything else is same.

∂y

$\frac{\partial L}{\partial z}$ Upstream grads

D_z

y

D_y

$[D_y \times D_z]$

for each element of z how much L changes

(Jacobian matrices)

for each output elem, how much change to y each input elem

Eg:

(RELU) → elem wise max

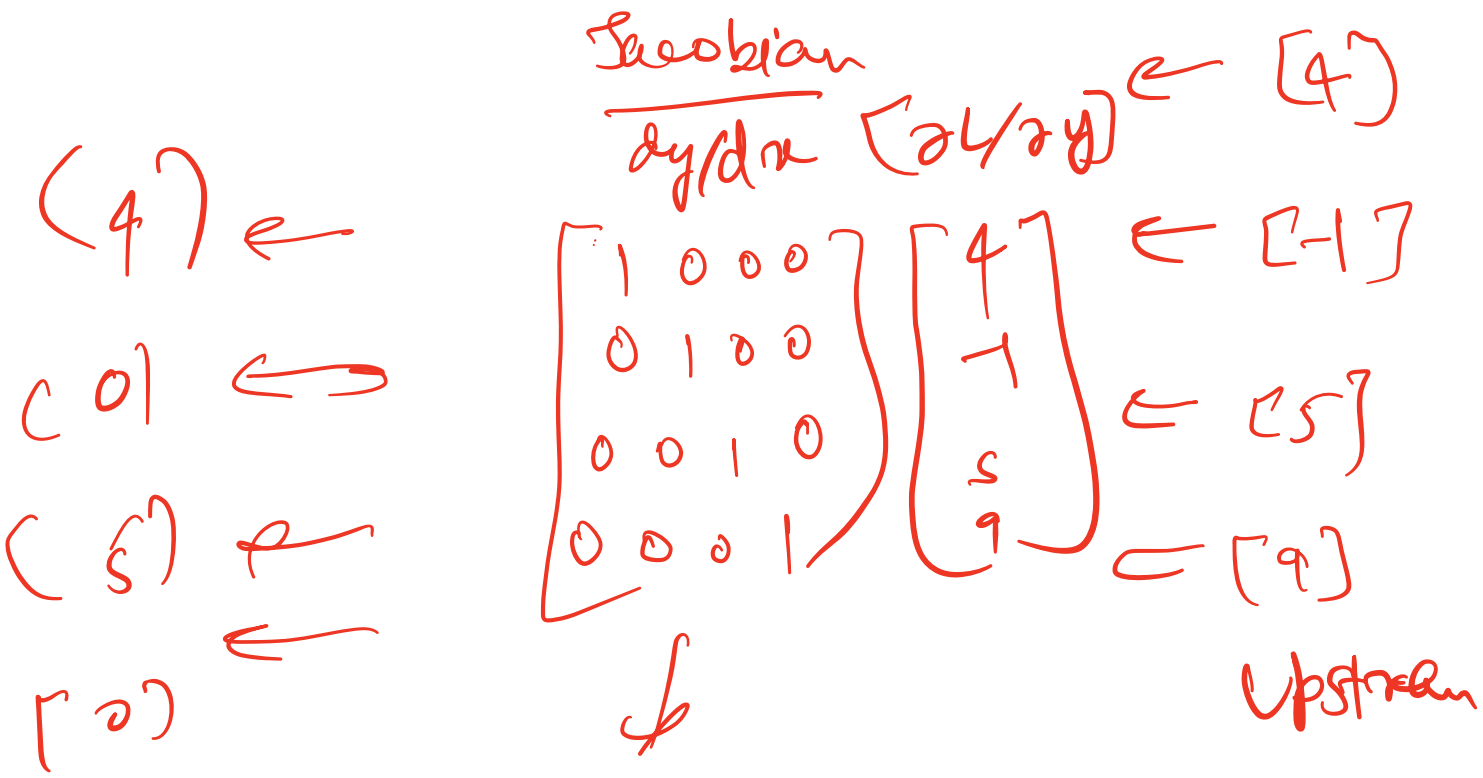
$[1] \rightarrow [1]$

$[-2] \rightarrow [0]$

$[3] \rightarrow [3]$

$[-1] \rightarrow [0]$

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



→ very sparse
usually
Jacobian for
most functions

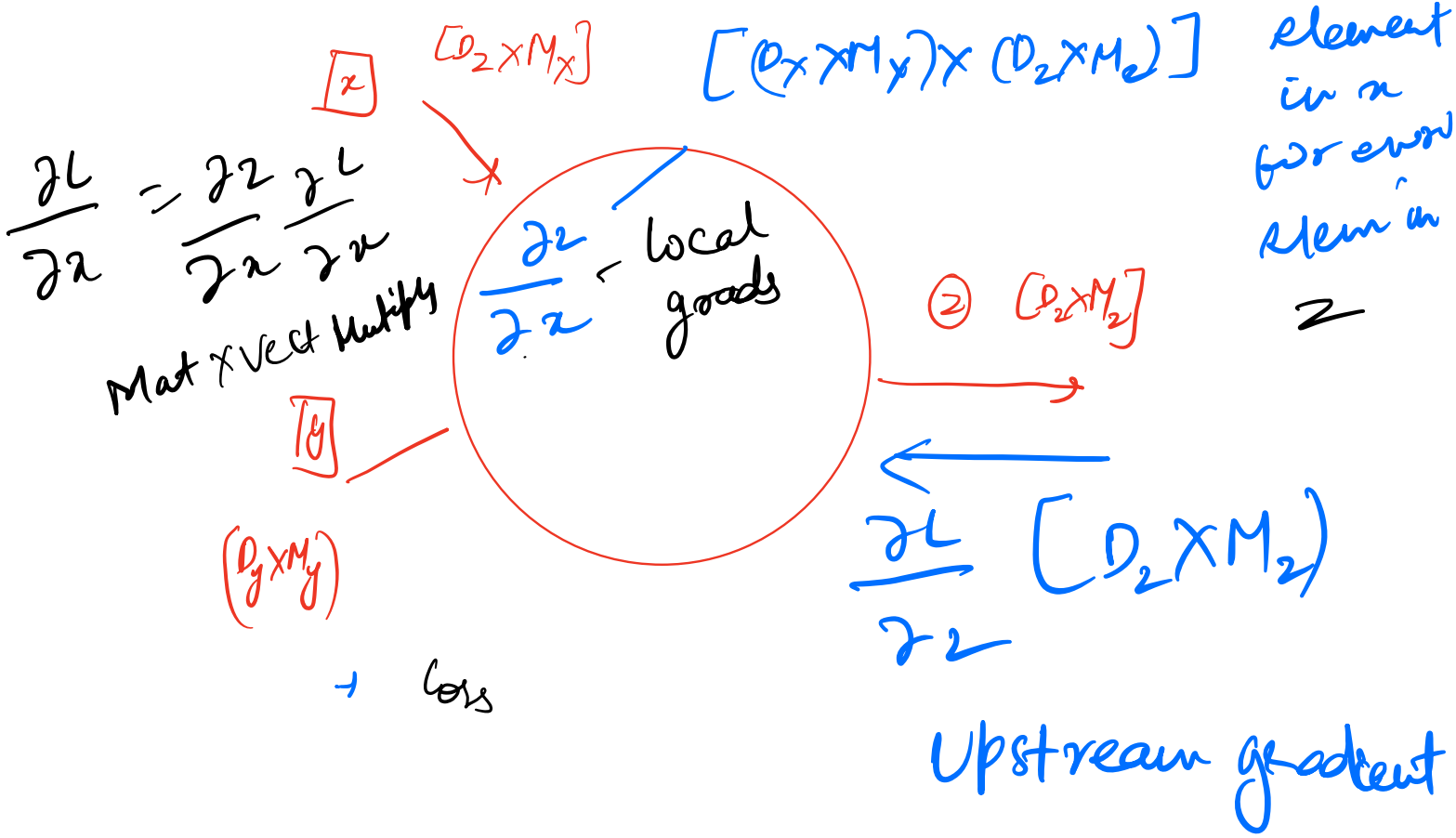
→ Hence the sparse
trick here is
to identify
efficient way to
get
same output

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

$$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow \begin{bmatrix} 4 \\ 1 \\ 5 \\ 9 \end{bmatrix} \leftarrow \text{Upstream grad}$$

① backprop with Matrices

, shape it for each



still a scaler

→ $\partial L / \partial x$ will always have same shape as x

→ Easy to Understand:

① Think of in element and element derivatives

Lets take mc example

x [N x D]

[2 1 -3]

[-3 4 2]

w [D x M]

[3 2 1 -1]

[2 1 3 2]

[3 2 1 -2]

↓

Matrix mult

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

↓

y

[-1 -1 2 6]

[5 2 11 7]

dL/dy

[2 3 -3 9]

[-8 1 4 6]

↓

Upstream grad

↓

Jacobian shape

$\partial y / \partial x = [(N \times D) \times (N \times M)]$

$\partial y / \partial w = [(D \times M) \times (N \times M)]$

Eg: $N = 64$ $D = M = 4096$

$= 256$ GB of mem

(Trick is to find a way to do this implicitly)

- Lets look at just a slice of local grad

$$\partial L / \partial x_{1,1}$$

$$= \left(dy / dx_{1,1} \right) \left(\partial L / \partial y \right)$$

↓ local grad slice

$$\begin{bmatrix} \boxed{2} & 2 & 2 & 2 \\ & & & \end{bmatrix}$$

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

$y_{1,1} = x_{1,1} \times w_{1\times 1} + x_{1,2}\, w_{2,1}$

$+ x_{1,3}\, w_{3,1}$

$dy_{1,1} / dx_{1,1} = \boxed{w_{1,1}}$

So, we know what this is

→ So you see a pattern

$\boxed{\partial y_{1,2} / dx_{1,1}}$ Just copying

$= [3 \;\; 2 \;\; 1 \;\; -1]$

$[0, 0, 0, 0]$

local gradient slice

↓

So now

$\partial L / \partial x_{1,1}$

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

$= (w_1, \dots) * (dL/dy_1)$

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

↓

→ Now keep on doing this

→ So think of it as one element at a time

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

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

Another View

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

Chain Rule

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

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

→ We do right to left,

so we always do vector matrix multiply

→ This is called reverse mode auto diff

→ Using same understanding we can use to calculate high order derivatives as well