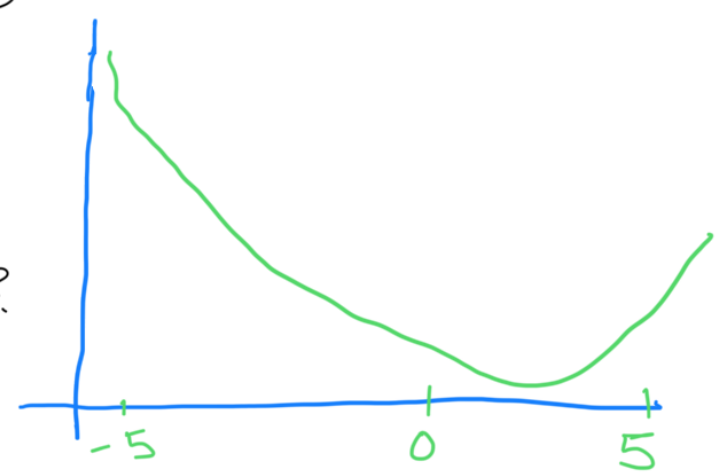
Spelled out intro to neural nets and back propagation : building micrograd - Andrej karpathy

---

$x = np.arange(-5, 5, 0.25)$

$y = 3x^2 - 4x + 5$



What is dy/dx at every point?

usual way -> find $\frac{dy}{dx}$ using calculus and put in the value of x.

Derivative = $\lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$

$f(x) = 3x^2 - 4x + 5$

$d = \frac{f(x+h) - f(x)}{h}$

$h = 0.0001$, $d \approx 20$

$x = 3$

Now,

Consider,

$d = a * b + c$,

$a = 2.0$

$b = -3.0$ $\Rightarrow d = 4$

$c = 10.0$

How, to calculate derivative of d w.r.t a?

$a_1 = a$, $a_2 = a + h$

$d1 = a_1 * b + c$, $d2 = a_2 * b + c$

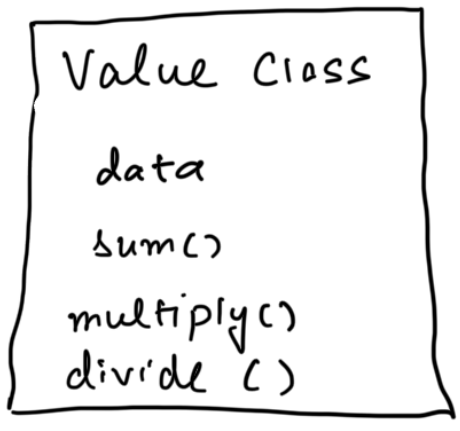
$\frac{\nabla d}{\nabla a} = \frac{d2 - d1}{h} \Rightarrow \frac{((a+h)b + c) - (a \cdot b + c)}{h}$

$\Rightarrow \frac{ab + hb + c - ab - c}{h} \Rightarrow \frac{hb}{h} = b$

Similarly, we do it for other variables.

Wrapping data into objects.

A



In order to overload +, −, × in python, you have special class of function __add__(self, other) where other is another object that you went to add with.

Other operations are :

Addition	__add__	+
Subtraction	__sub__	−
Multiplication	__mul__	*
Division (float)	__truediv__	/

We also have reverse methods which gets called when the main object is on the right hand side of the operand, just add r before the name, __rsub()__

In place methods, when we dont want to return a new object. just add i before the name, __isub__().

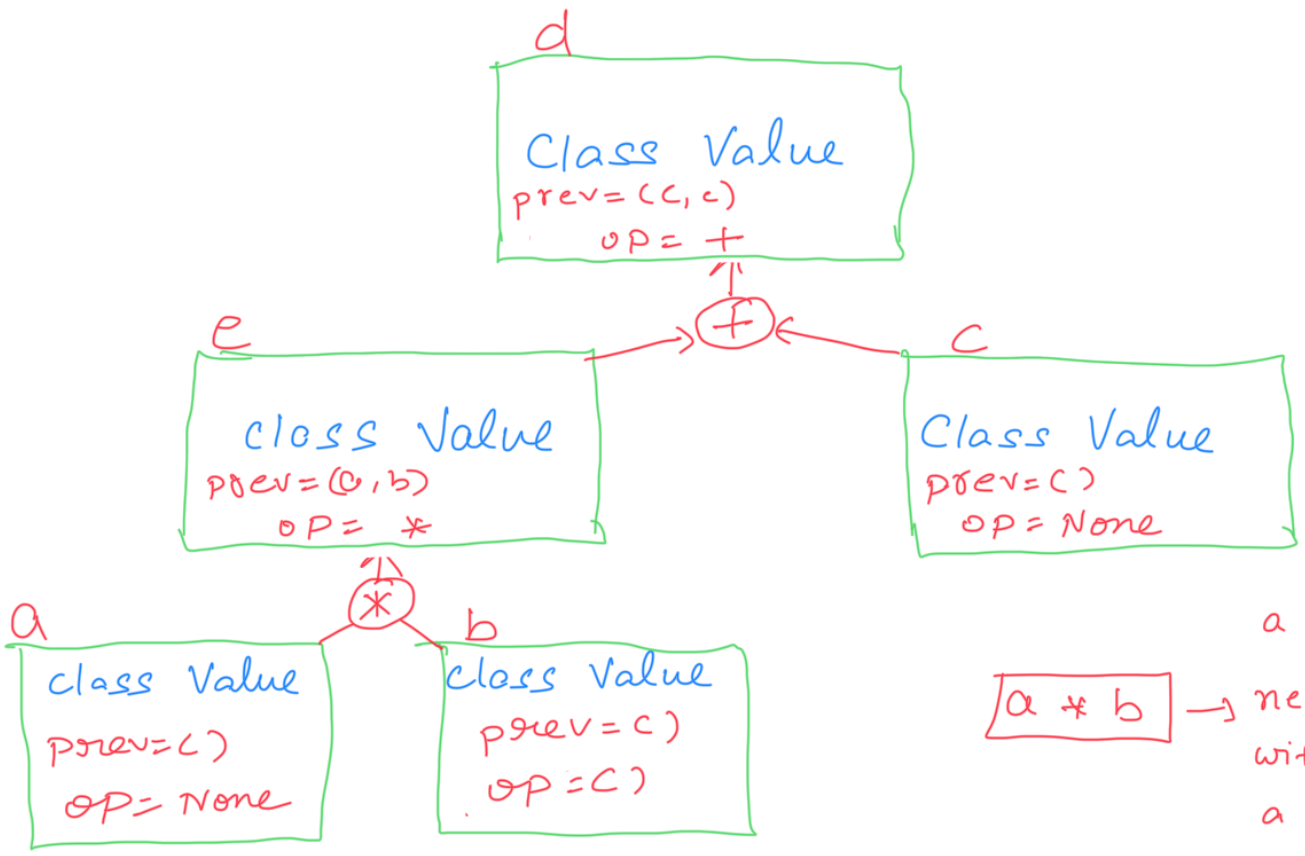
If the operation is not defined by special function class,
① Piggybank on existing operation not used much.
② Use functions instead of operation.

Now, that we have object representation of values

d = Value(a) * Value(b) + Value(c) } Same

a = (a.__mul__(b)).__add__(c)

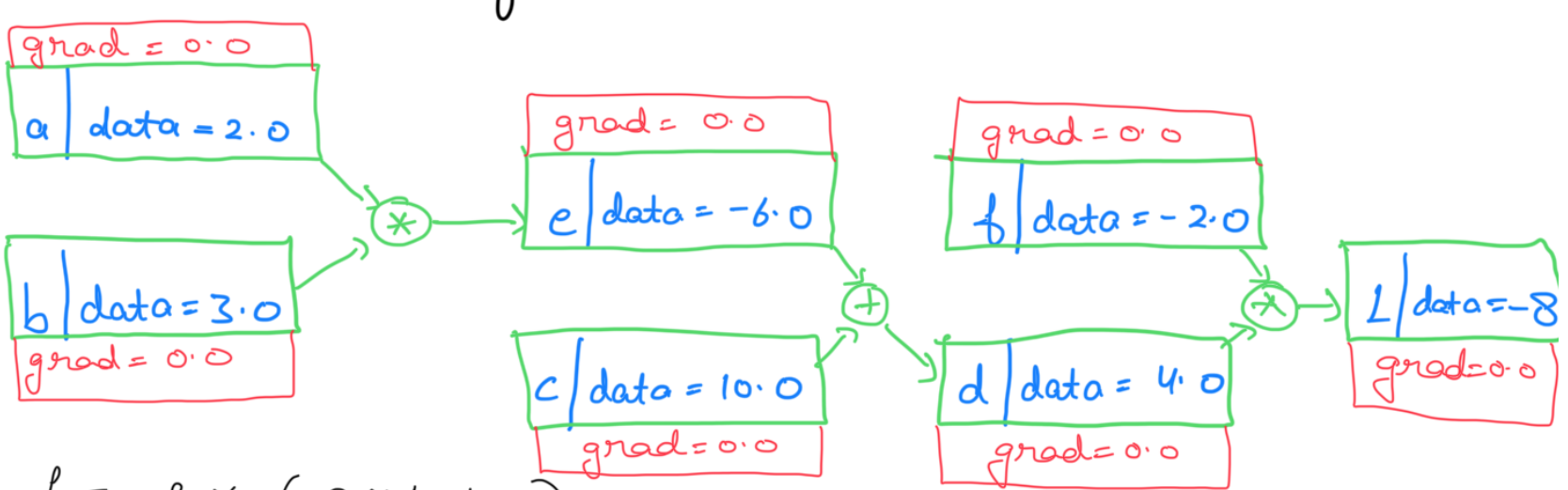
Now, we have the object notations of numbers. we now want to create a dependency graph to define a function and what values are dependent on what other values.



a + b → a returns new object e with prev as a and b

e + c → e returns new object d with prev as e and c.

Consider a loss function such as:



$L = f * (a * b + c)$

how do we find the derivative of L with all the variables ?

We define a new variable grad for each value which will represent the gradient of output wrt that value.

Gradient Check -

gradient checks

numerical way = calculate the change in output with a small change in input.

$$\frac{dL}{da} = \frac{L(a+h) - L(a)}{h}$$

$$= \frac{f*((a+h)*b+c) - f*(a*b+c)}{h}$$

by this method we can find the derivative of output with each dependent values.

problems →

for this method to implement, we have to calculate a new value $f(x+h)$ for every dependent value, hence will take $O(N^2)$ complexity for n variables as each calculation will take $O(N)$ which will be repeated N times.

This method is not suitable to be used in production but can be used for correctness of our algorithm.

Auto Differentiation

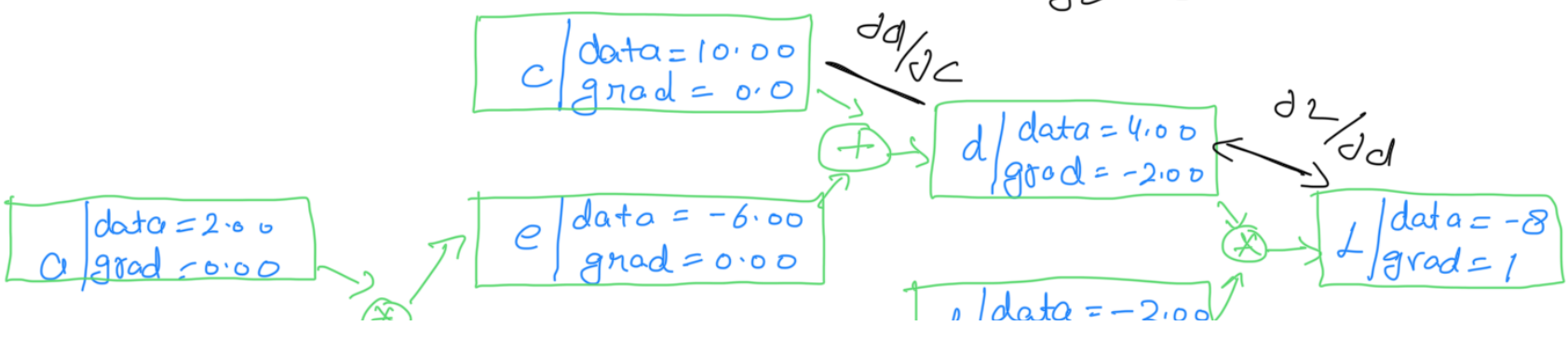
$L = d * f$, $\frac{dL}{dd} = f$, $\frac{dL}{df} = d$

$L = f*(a*b+c)$.

how do we calculate $\frac{dL}{dc}$ without symbolic differentiation and without numerical differentiation?

let's draw the graph again.

$$\frac{dL}{dc} = \frac{dL}{dd} * \frac{\partial d}{\partial c}$$



b | data = 3.00
grad = 0.00

f | grad = 4.00

in order to calculate $\frac{dL}{}$, we can just calculate $\frac{dL}{dd}$ and $\frac{dd}{dc}$ and apply the chain rule.

Chain rule = $\frac{dL}{da} = \frac{dL}{db} \times \frac{db}{dc} \times \frac{dc}{dd} \times \frac{dd}{da}$ ....

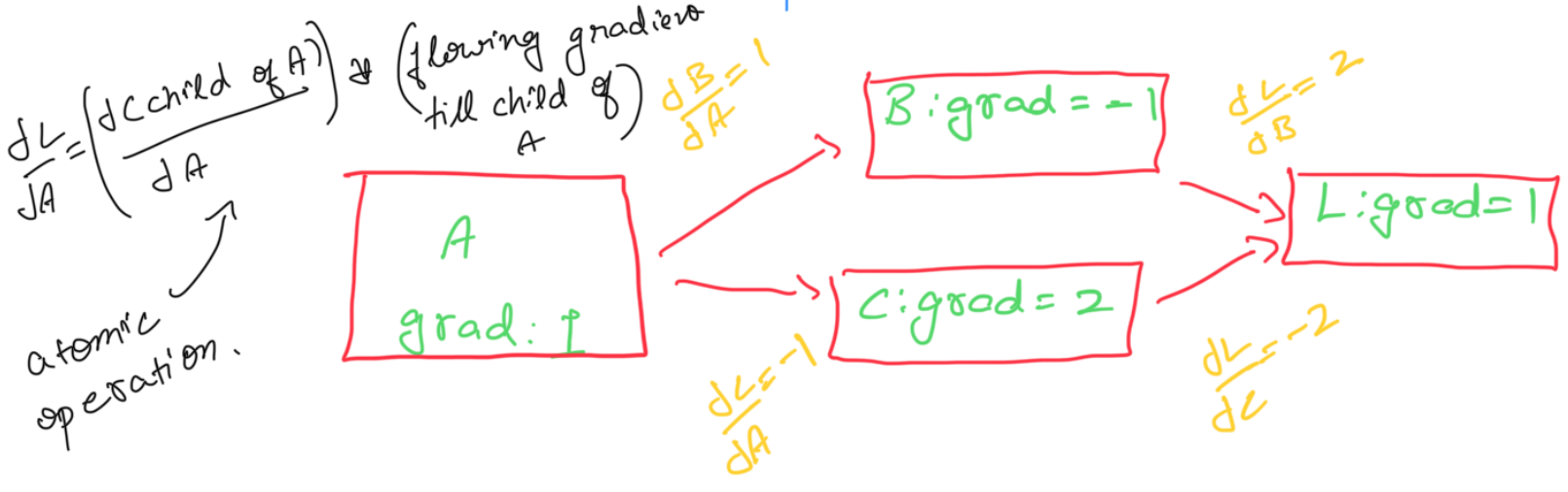
So,

in order to calculate the derivate of output with any variable, we move along the edges to that input and accumulate the gradients to that node via multiplication.

Hence,

1) The graph will always be a DAG.
2) The edge will store the derivative of child w.r.t parent.

The value at edge which is a derivative of child wrt parent can be stored in parent class as a grad that can be summed up across all the childs.

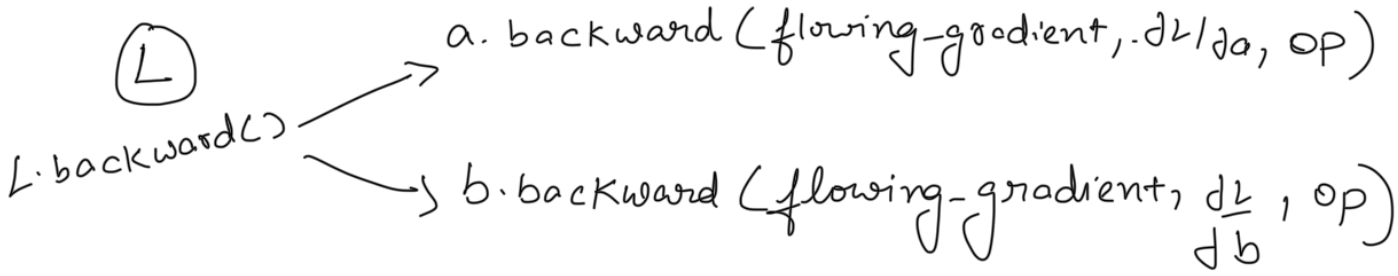


$B.grad += L.grad * \frac{dL}{dB}$, $C.grad = L.grad * \frac{dL}{dC}$

$A.grad += B.grad * \frac{dB}{dA} + C.grad * \frac{dC}{dA}$.

thought:

pseudocode:



a.backward(flowing_gradient, $\frac{d\,child}{d\,parent}$, operation):

a.grad += flowing gradient $\times$ $\frac{d\,child}{d\,parent}$.

flowing_gradient * = $\frac{d\,child}{d\,parent}$

if op = '*':

child1.backward(flowing_gradient, child2.data, '*')

child2.backward(flowing_gradient, child1.data, '*')

Similarly for other atomic operations.

In summary, Auto differentiation is

① Splitting the output expression into atomic operations of +, −, ÷, * etc.

② Creating a DAG of our function using these atomic expressions.

③ Doing a forward pass on our DAG using our atomic expressions.

④ Recursively applying chain rule and calculating the gradient of parent w.r.t child node.

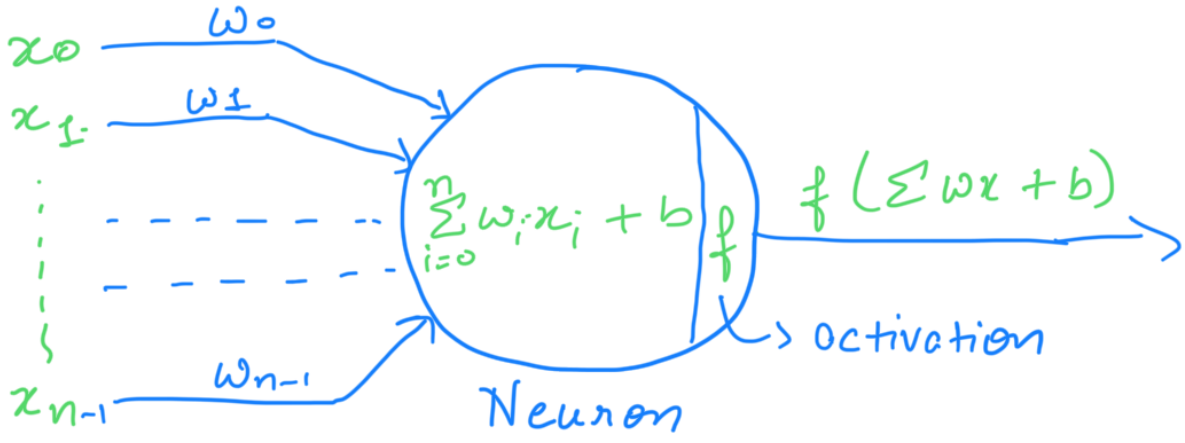
⑤ after gradient accumulation, changing the value of variables according to gradients.

Total complexity = O(N), N = number of variables.

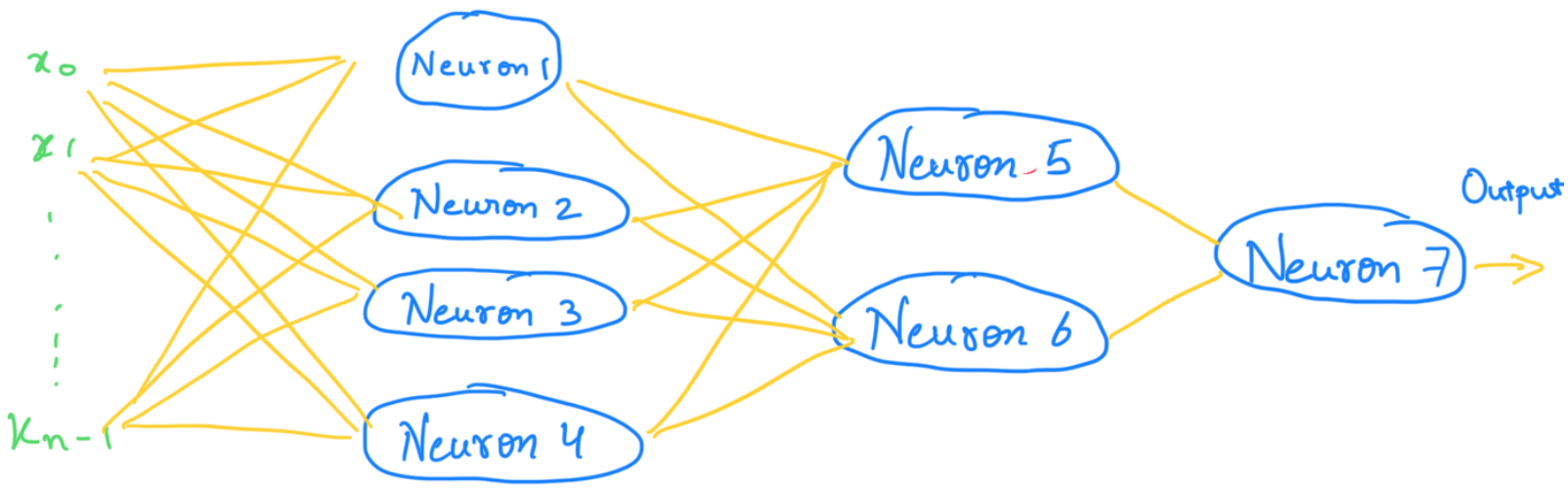
backpropagation = Recursive application of chain rule backwards in computational graph

Now, consider a more realistic and complex scenario.

How to design a neuron?



2 Layer Neural Network.



given input $x = [x_0, x_1, x_2, x_3 \cdots\cdots x_{n-1}]$

how to prepare a neuron which can do a forward pass as well as backward pass?

$$\text{Neuron } n = \sum_{i=0}^{n} w_i x_i + b$$

activation functions:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

designing a neuron:

x_1 = Value (2.0)

x_2 = Value (0.0)

w_1 = Value (-3.0)

w_2 = Value (1.0)

b = Value (6.7)

$x_1 w_1 = x_1 * w_1$

$x_2 w_2 = x_2 * w_2$

$x_1 w_1 x_2 w_2 =$

$x_1 w_1 + x_2 * w_2$

$n = x_1 w_1 x_2 w_2 + b$

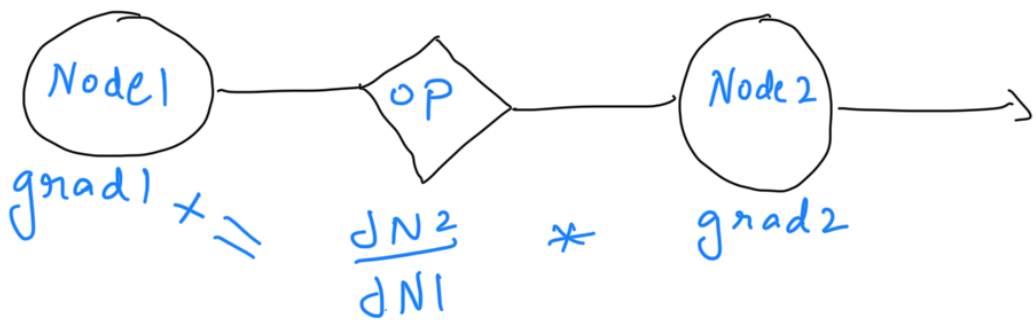
$O = n.\tanh()$

You dont have to break the local derivative into atomic pieces, like the case of tanh.

$$\tanh = \frac{e^{2x}-1}{e^{2x}+1}$$

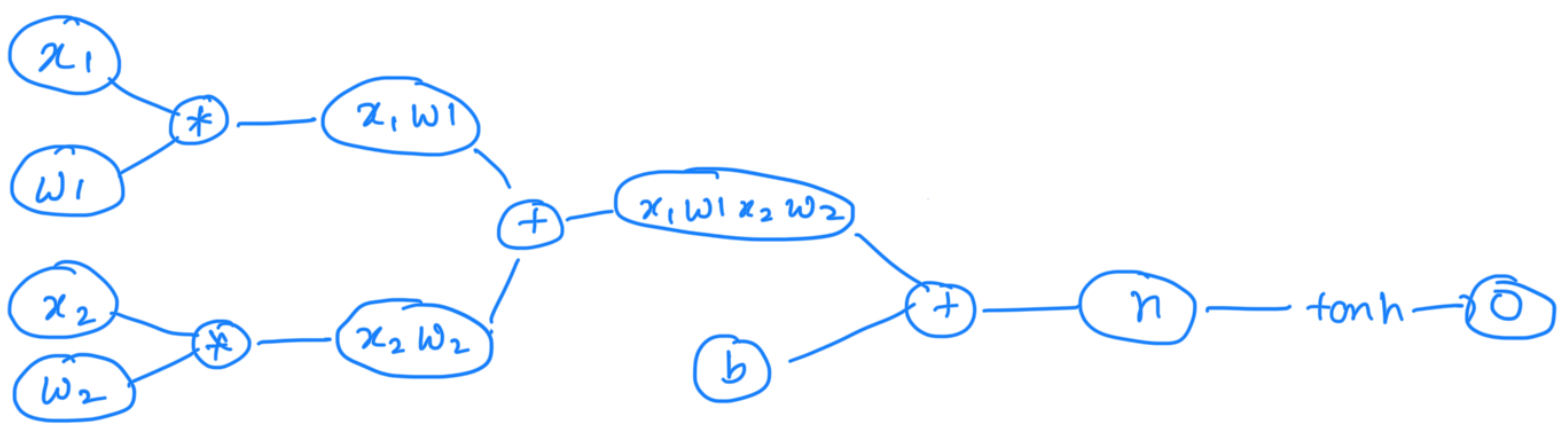
tanh can be broken down further into atomic operations of $2x$, e^{2x}, $e^{2x}-1$, $e^{2x}+1$ but we can choose to not do that and just compute the gradient of tanh directly as it is commonly used.

general rule of backprop:



$$\frac{d(\tanh)}{dx} = 1-(\tanh)^2$$

$$\frac{d(\text{sigmoid})}{dx} = (\text{sigmoid})(1-\text{sigmoid})$$



$x_1.grad = x_1w_1.grad * w_1.data$

$w_1.grad = x_1w_1.grad * x_1.data$

$x_2.grad = x_1w_1.grad * w_2.data$

$w_2.grad = x_2w_2.grad * x_2.data$

$x_1w_1.grad = x_1w_1x_2w_2.grad * 1$

$x_2w_2.grad = x_1w_1x_2w_2.grad * 1$

$x_1w_1x_2w_2.grad = n.grad * 1$

$b.grad = n.grad * 1$

$n.grad = (1 - (o.data)^2) \cdot o.grad$

The NN graph is a DAG so while doing backprop, we apply topological sort. so that the parents gradients are computed only after child's gradient has been computed.