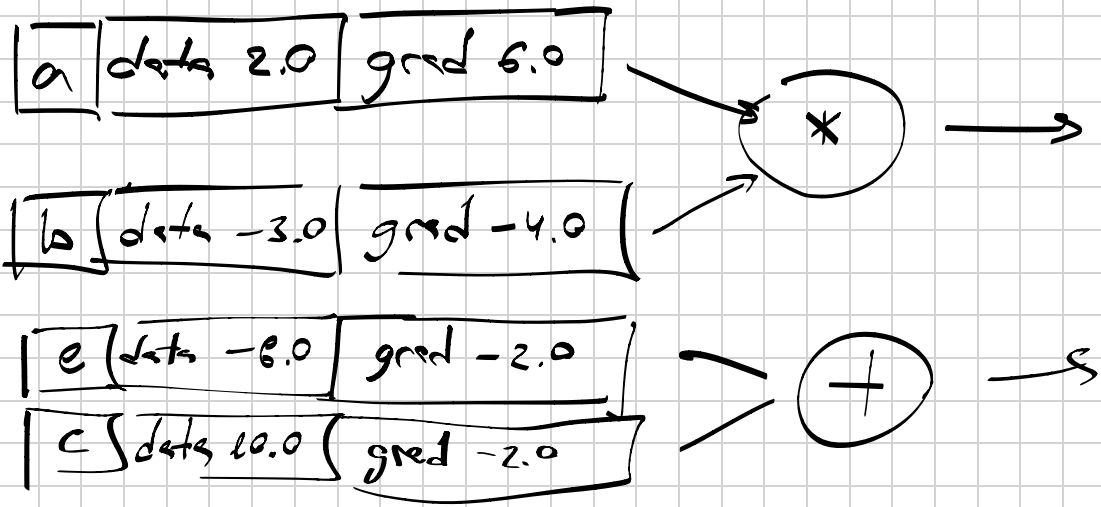
# Micro Grad




...

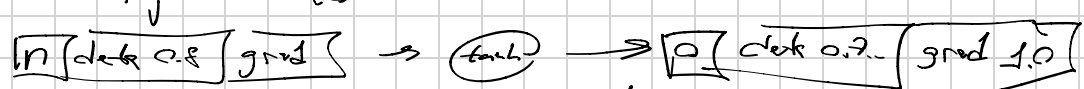
grad - gradient. Basically finding derivative

$\frac{dx}{dL}$ at each point.

So if ...

So: we have o - output

o.grad $= \frac{do}{do} = 1.0$

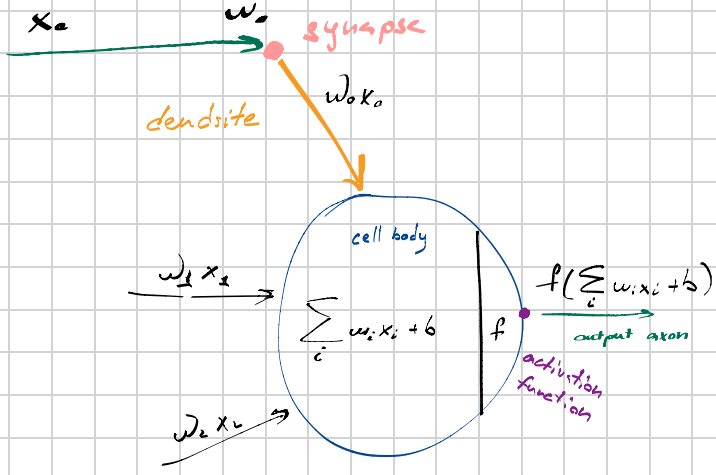



=> for backpropagation we need $\frac{do}{dn} = \frac{d(\tanh(n))}{dn} =$

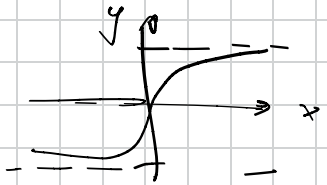
$= 1 - \tanh^2(n) \approx 0.5$

This process of finding local derivatives continues.

# Math model of neuron:




Usually some kind of squashing func like Sigmoid.




$x_1 * w_1 = x_1 w_1$ chain rule

$$\rightarrow \frac{d x_1}{d x_1 w_1} = w_1 * (x_1 w_1 . grad)$$

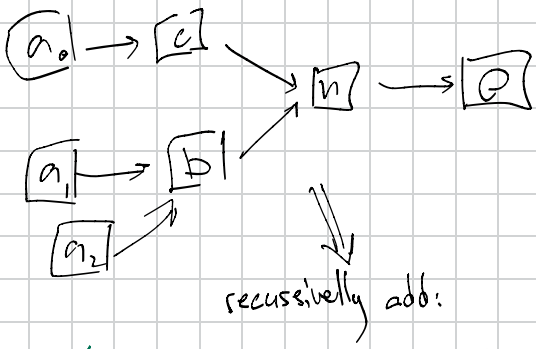
$$\frac{d w_1}{d x_1 w_1} = x_1$$

Now we defined _backwards functions for each of the operational functions so we can backpropagate automatically and calculate local derivatives.

We want to do a topological sort such that it doesn't call o until we worked with all of it's children.

The topo would be:

```
visited = set()
topolist = [ ]
# o = node (output node)
def topo(v):
    if v not in visited:
        visited.add(v)
        for ch in v._prev:
            topo(ch)
        topo_list.append(v)
topo(o)
topolist
```




$(a_0, c, a_1, a_2, b, n, o)$

this is the order we need.

Thus we can:

```
for node in reversed(topolist):
    node._backwards()
```

# this gives us all the .grad values

# Linear Regression (forward) Problem

Must implement linear regression:

1. implement get_model_prediction()

   Inputs:

   X - dataset used for predictions. len(X) = n & len(X[i]) = 3

   weights - the current $w_1, w_2, w_3$ for the model. len(weights) = 3

   Basically:

   $X = [[x_{01}, x_{02}, x_{03}], [x_{11}, x_{12}, x_{13}], \dots [x_{n1}, x_{n2}, x_{n3}]]$

   $weight = [w_1, w_2, w_3]$

2. implement get_losses()

   Inputs:

   model_prediction - prediction for each training ex. len(m_p) = n

   ground_truth - correct ans for each ex. len(g_t) = n

   Basically:

   $m\_p = [y\_p_1, y\_p_2, y\_p_3, \dots y\_p_n]$ , nx1 np.array

   $g\_t = [y\_c_1, y\_c_2, y\_c_3, \dots y\_c_n]$ , nx1 np.array

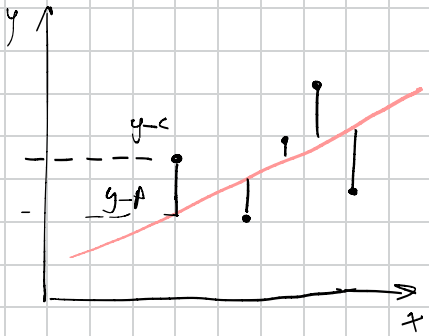
Building:

- some Qs right away. What loss func, what loss are looking for?

Standard loss would be the distance, or avg distance

⇒

So for 2-dimensional (we have the same but 3 dim, 3 Xs for each y input)




$\frac{\sum (y_{ci} - y_{pi})^2}{n}$ — ? is that what we want.

basically what I have is Mean Squared Error (MSE)

where $MSE = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

There is also RMSE, basically the same just for the additional root square of MSE

$RMSE = \sqrt{MSE}$

But in Statistics the standard would be SE (Standard Error of Regression)

It would be: $SE = \sqrt{\frac{SSE}{n-k}} = \sqrt{\frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n-k}}$

and k - number of parameters (3 in our case)

Now the get_model_predictions()

seems really simple, we just want to get array size $N \times 1$

from $N \times 3$ and $1 \times 3$

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

, need $[y_1, y_2, y_3, \ldots y_n]$

or

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$ (doesn't matter)

Solved!

I overcomplicated the problem, but learned/revisited some things and got practice so all good, loved it!!! ♡

# Linear Regression (Training) Problem

We have get_derivative() and get_model_prediction() need to implement train_model()

@inputs:

1. X - dataset ( $X = \begin{bmatrix} x_{01} & x_{02} & x_{03} \\ x_{11} & x_{12} & x_{13} \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}$ $N \times 3$
2. Y - correct answers ( $Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$ $N \times 1$
3. num_iterations - # of iterations to run
4. initial_weights - weights ( $w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$ $3 \times 1$

@return:

res = [ $w_1, w_2, w_3$ ] - final weights after training

Solution:

```
for _ in range(num_iter):
    current_prediction = self.get_model_prediction(X, init_weights)

    # get all grads & update w[i]s according
    for i in range(len(w)):
        grad = self.get_derivative(current_pred, Y, len(X), X, i)
        w[i] -= grad * self.learning_rate

return np.round(w, 5)
```