NEURAL NETWORK

Convolutional neural network → Good for image recognition

Long short-term memory network → Good for speech recognition

NEURAL NETWORK

What are neurons?

Thing that holds a number (mostly b/t 0 & 1)

All based on its grayscale representation ranging b/t 0 (Black) & 1 (white) pixels.

(x) → no. inside the neuron is called "ACTIVATION"

They have values b/t 0 to 1 as whichever digit is represented by the network

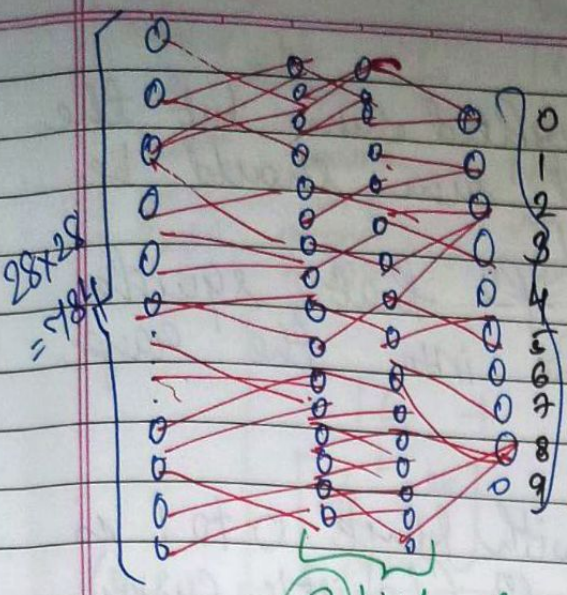
How are they connected?

(1) If we have 28 by 28 pixels grids then $28 \times 28 = 784$. becomes the 1st layer of our network.

(2) The last layer is the 4th layer consisting of 10 neurons each representing digits from 0 to 9.

(3) The middle 2 layers in between are called as the hidden layers.

The brightest of all neurons (value close to 1) is the value of digit that the neural network represent



(2) Hidden layers selected of (16) neurons each

assumed & can be taken as anything

Representations of (1) Digits

1) 9 = 0 + 1

2) 8 + 0 + 0

3) 4 + 1 + 1 + —

all this becomes the part of 3rd layer

This all loops/lines are divided into smaller sections to recognise those loops and they become the 2nd layer.

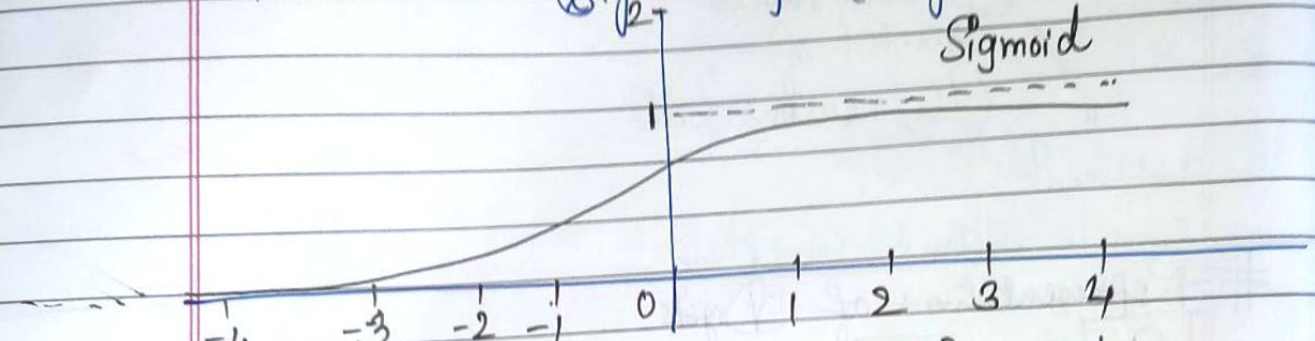
0 = (+) + (+)

1 = | + |

We calculate the weighted sum but the range of weighted sum should be in between 0 to 1

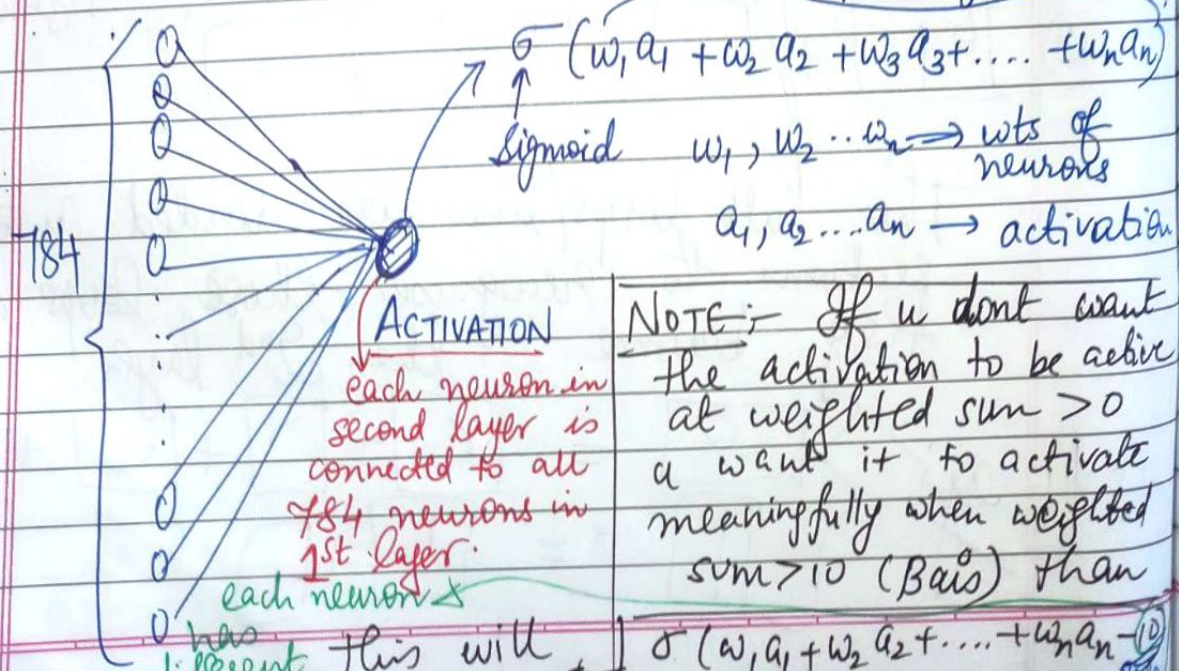
eg: we include a fn that squishes the real number line into the range between 0 & 1

The common function with range 0 to 1 is known as Sigmoid fn (Logistic curve).



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

-ve inputs → close to 0
+ve inputs → close to 1



NOTE: If u dont want the activation to be active at weighted sum > 0 u want it to activate meaningfully when weighted sum > 10 (Bais) than

$$\sigma(w_1 a_1 + w_2 a_2 + \dots + w_n a_n - 10)$$

WEIGHTS.

$784 \times 16 + 16 \times 16 + 16 \times 10$

- 784: neurons in 1st layer
- 16: neurons in second layer
- 16: Neurons in 2nd layer
- 16: Neurons in 3rd layer
- 16: Neurons in 3rd layer
- 10: Neurons in 4th layer

BAISES.

$16 + 16 + 10$

- 16: b/t 1st & 2nd
- 16: b/t 2nd & 3rd
- 10: b/t 3rd and 4th

Total 13002 baises & weights

Learning :— Finding the right weights and baises

REFERENCE

1st layer (28×28) → 2nd layer → 3rd layer → 4th layer (9)

$$a_0^{(1)} = \sigma(w_{0,0}a_0^{(0)} + w_{0,1}a_1^{(0)} + w_{0,2}a_2^{(0)} + \dots + w_{0,n}a_n^{(0)} + b_0)$$

Bais

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ \vdots & \vdots & & \vdots \\ w_{k,0} & w_{k,1} & & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} = \begin{bmatrix} \quad \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Sigmoid

① ORGANISE ALL THE ACTIVATIONS FROM ONE LAYER INTO A COLUMN AS A VECTOR.

② ORGANISE ALL THE WEIGHTS AS MATRIX where each row of that matrix corresponds to the connections b/t one layer and a particular neuron in the

③ Bais Vector to add all bais to the previous vector product of weighted sum.
This can be represented as

$$a^{(1)} = \sigma(Wa^{(0)} + b)$$

$a^{(1)}$ → 1st layer; $a^{(0)}$ → Initial layer; b → bais vector.

→ Nowadays, RELU function is used instead of sigmoid fn

We define a cost function which is the squares of the differences between 2 values which we get and the actual value

→ The actual value is generally 1.0 for a single pixel & 0 for others

Consider avg cost of all datas.

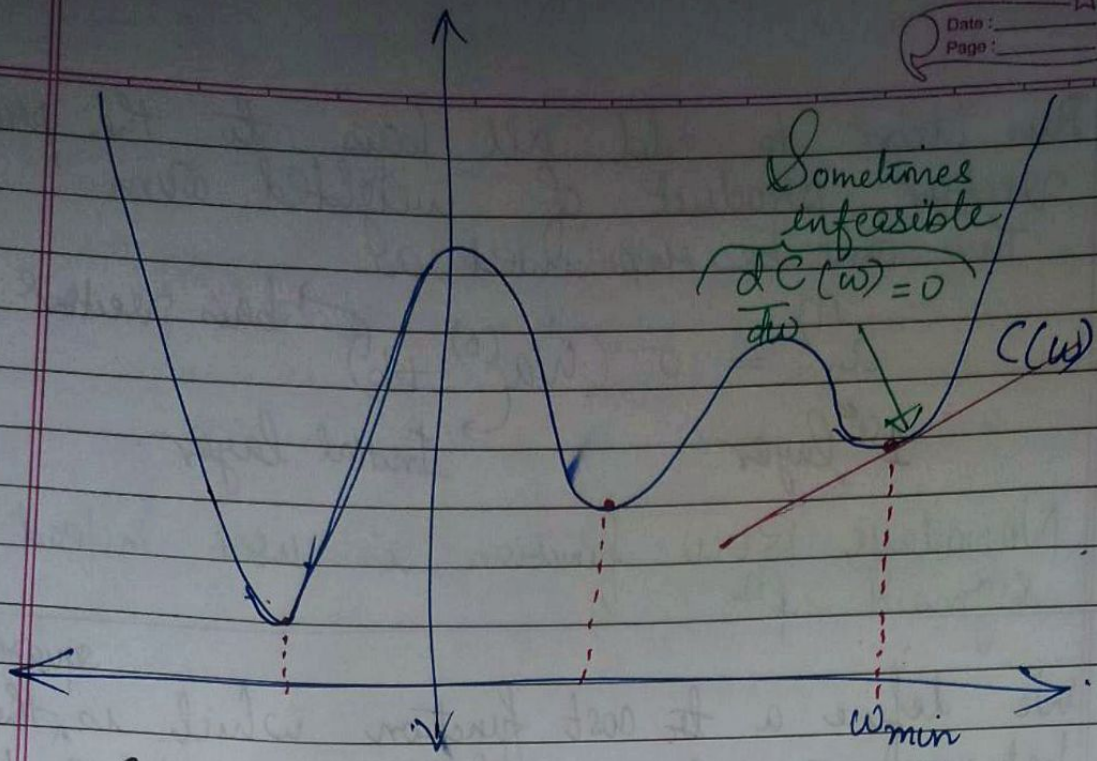
$$\text{Cost} = (0.43 - 0.0)^2 + (0.28 - 0.0)^2 + (0.8 - 1.0)^2 + \dots$$

(1.0 → one value which we expected as ans.)

Should be as small as possible

wrong random values which function have given

Neural Network function	Cost function
Input: 784 numbers (pixels)	Input: 13,002 weights/baises
Output: 10 numbers.	Output: 1 number (the cost)
Parameters: 13002 weights/baises	Parameters: Many many training examples



There are many minimas for a complicated fn but all minimas don't have the minimum value of the fn, only one minima has the minimum value of the fn

↓

The minima is determined by calculating the slope of fn at that point

If slope is +ve → shift to left.
If slope is −ve → shift to right.

↳ by this way we get a point where slope is zero and we find a minimum.

GRADIENT :-

Gradient of a function gives you the direction of steepest ascent. basically which direction should you step to increase the function most quickly. Taking the negative of that gradient gives you direction to step that decreases the function most quickly. Length of the gradient vector is actually an indication for just how steep that steepest slope is.

"Gradient" → the direction of steepest increase $\nabla C(x, y)$.

$$\vec{w} = \begin{bmatrix} 2.23 \\ -1.57 \\ 1.98 \\ \vdots \\ -1.16 \\ 3.82 \\ 1.21 \end{bmatrix} \qquad -\nabla C(\vec{w}) = \begin{bmatrix} 0.18 \\ 0.45 \\ -0.51 \\ \vdots \\ 0.40 \\ -0.32 \\ 0.82 \end{bmatrix}$$

negative gradient of cost fn

This tells which changes to which weight matters most

13,002 weights & biases.

How to nudge all weights & biases

The negative gradient of cost function is just a vector that tells you which nudges to all those nos is going to cause the most rapid decrease to the cost function

We need to minimise cost funct as small as possible.

The process of repeatedly nudging an input of a function by some multiple of the negative gradient is called gradient descent.
Its a way to converge towards some local minimum of cost function

each component of the negative gradient $(-\nabla C(\vec{w}))$ tells us 2 things.

- The sign tells us whether the corresponding component of input vector should be nudged up or down
(~~relative magnitudes of all comp~~)
- the relative magnitudes of components states that which chages matter more.

eg

$$-\nabla C(\vec{w}) = \begin{bmatrix} 0.31 \\ 0.03 \\ -1.25 \\ \vdots \\ 0.78 \\ -0.37 \\ 0.16 \end{bmatrix} \quad (13002)$$

w_0 should increase somewhat
w_1 —"— a little
w_2 should decrease alot
w_{13000} should increase a lot.
w_{13001} should decrease somewhat
w_{13002} should increase a little.

If a function has 2 variables as an input, and u compute that its gradient at some particular point comes out as (3, 1)

$$C(x,y) = \frac{3}{2}x^2 + \frac{1}{2}y^2$$

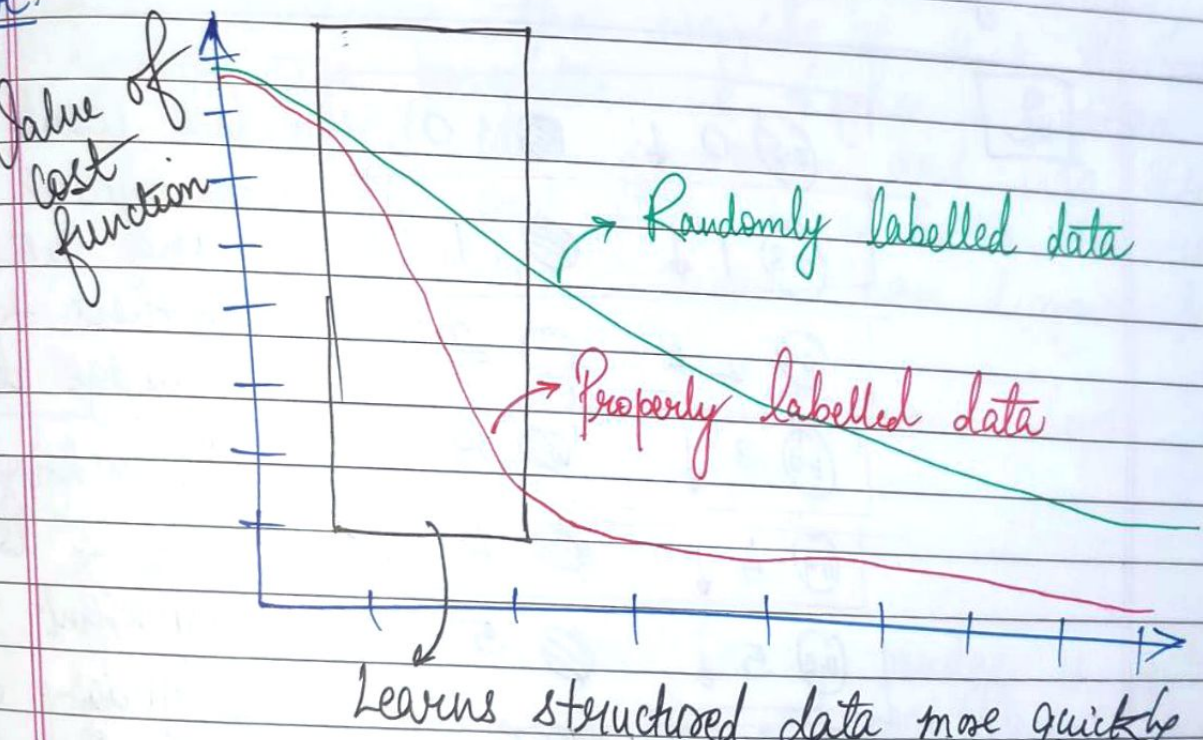
$$\nabla C(1,1) = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

Direction of steepest ascent.

x has 3 times the impact on C as y.

x values carries more importance

Note:-



BACK PROPAGATION

↓

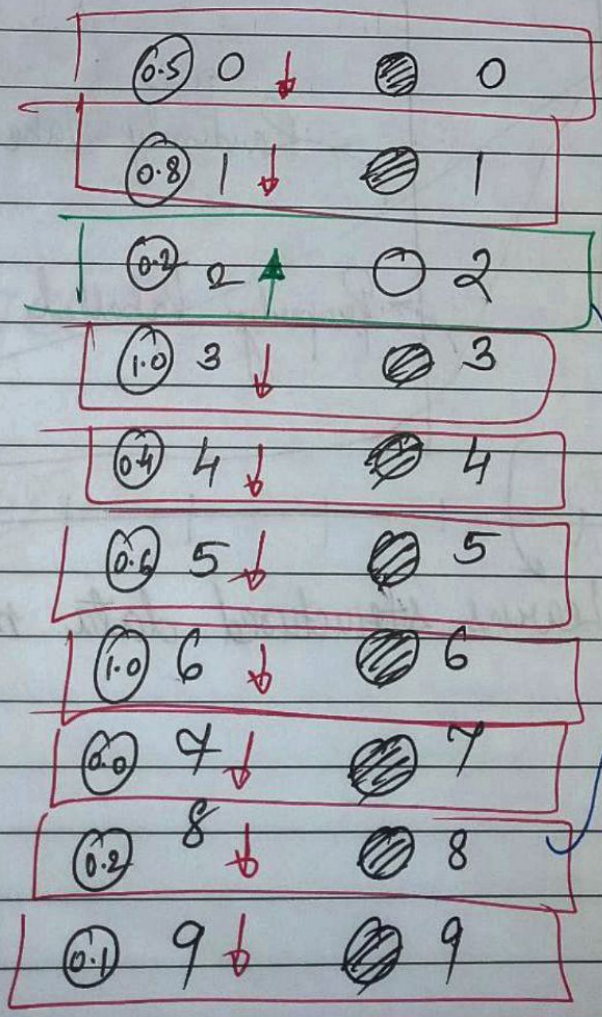
Algorithm for ~~up~~ computing the complicated gradient. An algorithm for determining how a single training example would like to nudge the weights and baises,

$$-\Delta C(\ldots) = \begin{bmatrix} 0.16 \\ 0.72 \\ -0.93 \\ \vdots \\ 0.04 \\ 1.64 \\ 1.52 \end{bmatrix}$$

↑ All weights and baises

The magnitude of each component ~~does~~ how sensitive the cost function is to each weight and bais.

2



We want 2 to nudge up and all others to nudge down

→ The nudging up of 2 is more important than nudging down of 8 as 0.2 is pretty close to 0 but not to 1.0

Focusing on neuron whose activation is to be increased (2).

Activation → weighted sum of all activation in previous layer + bais in the sigmoid squification fn.

→ (0.2) neuron of no 2. $= \sigma(w_0 a_0 + w_1 a_1 + \dots + w_{n-1} a_{n-1} + b)$

To increase the activation, 3 methods.

1] Increase bais (b).

2] Increase w_i (weights) in proportion to a_i

Note :- the weighted sum is of previous layer, so if u increase the weights of that neurons which are brighter i.e higher activation than the weighted sum increases. and has stronger influence to cost fn ; ∴ it is better to increase weights of brighter neurons than dimmer neurons.

3] Change a_i (activation) of previous layer in proportion to w_i

Back Propagation

Whatever upper or lower nudge u get for the neuron in the 2nd last layer by considering the nudges of all the neurons in the last layer, we can also find the nudge of its previous (i.e 3rd last layer). This is Back Propagation

→ For all datas, we do the following and take the average to generalise the negative gradient of cost fn.

We adjust the weights so that the output in last layer is 2 by making the weights of brighter neurons more. Similarly for all.

	2	5	0	4	1	9	Avg
w_0	−0.08	+0.02	−0.02	+0.11	−0.05	−0.14	→ −0.02
w_1	−0.11	+0.11	+0.07	+0.02	+0.09	+0.05	→ +0.12
w_2	−0.07	−0.04	−0.01	+0.02	+0.13	−0.15	→ −0.06
⋮	⋮	⋮	⋮	⋮	⋮	⋮	
w_{13002}	+0.13	+0.08	−0.06	−0.09	−0.02	+0.04	→ +0.04

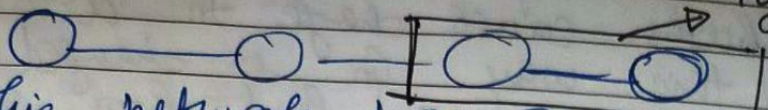
This is the negative gradient of cost function: $-\eta \Delta C(w_1, w_2, \ldots, w_{13002})$ → Average over all training data

The above process is little slow, so we randomly divide the input data of nos. into some minibatches, and repeatedly going through all of the mini batches and making these adjustments, we get a local minimum of cost fn

↓

Stochastic gradient descent

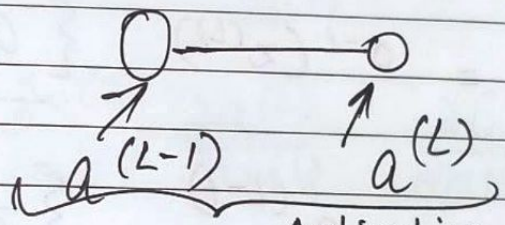
Consider a simple network, where each layer has one neuron each.

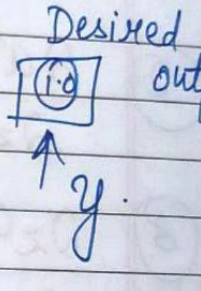


This network has 3 weights & three bais (w_1, w_2, w_3) & (b_1, b_2, b_3)

Cost fn → $C(w_1, b_1, w_2, b_2, w_3, b_3)$

Our goal is to understand how sensitive is the cost function to these variables.
Also, we need to know which adjustments to these terms, cause most efficient decrease in cost function.





Cost of one training example → $C_0(\ldots) = (a^{(L)} - y)^2$

$$a^{(L)} = \sigma(w^{(L)} a^{(L-1)} + b^{(L)})$$

$$z^{(L)}$$

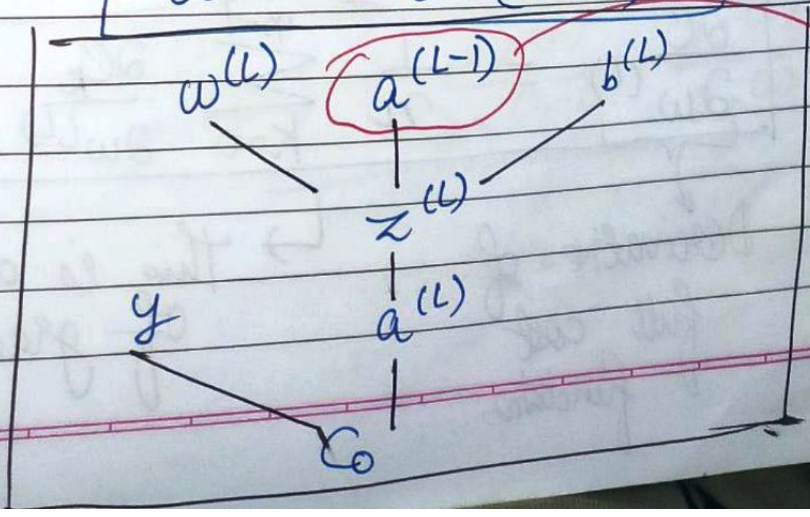
$$a^{(L)} = \sigma(z^{(L)})$$

$w^{(L-1)}$ $a^{(L-2)}$ $b^{(L-1)}$

$z^{(L-1)}$

$a^{(L-1)}$

FLOWCHART



A small change in weight ($w^{(L)}$) causes the small change in $z^{(L)}$ and that further cause chage to $a^{(L)}$ and by this way C_0 is changed.
Our goal is to find how cost function changes by a small chage in weight
So, by Chain RULE

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

① $\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$ $\{C_0 = (a^{(L)} - y)^2\}$

② $\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$ $\{a^{(L)} = \sigma(z^{(L)})\}$

③ $\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$ $\{z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}\}$

So, $\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$

→ Average of all training examples is

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$$

↳ this is one component of gradient vector

Derivative of full cost function

Gradient vector is formed by the partial derivates of the cost fn w.r.t all those weights and baises.

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

for bais term $\partial C / \partial b^{(L)}$

$$\frac{\partial C}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

(4) $\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1 \quad \{ z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)} \}$

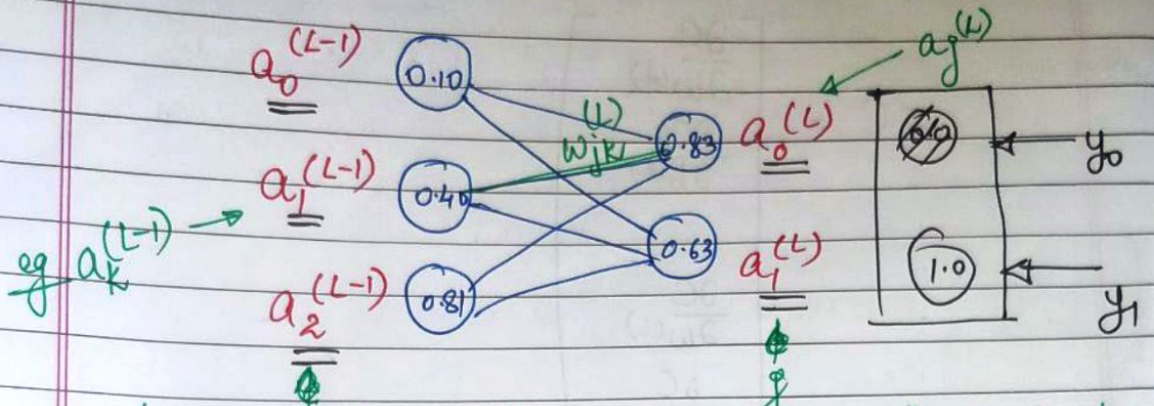
$$\therefore \frac{\partial C}{\partial b^{(L)}} = 1 \cdot \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

Now, we can use the same chain rule idea backwards, this is called back propagation

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

w_L

For more than one neuron in a layer there will also be subscript to denote those neurons



$$C_0 = \sum_{j=0}^{n_{L-1}} \left(a_j^{(L)} - y_j\right)^2$$

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + b_j^{(L)}$$

$$a_j^{(L)} = \sigma\left(z_j^{(L)}\right)$$

$$\frac{\partial C_0}{\partial w_{jk}^{(L-1)}} = \sum_{j=0}^{n_{L-1}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L-1)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}}$$

Sum over layer L

$$\frac{\partial C_0}{\partial a_k^{(L-1)}}$$

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}}$$

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = a_k^{(L-1)} \cdot \sigma'(z_j^{(L)}) \cdot \boxed{\frac{\partial C}{\partial a_j^{(L)}}}$$

$$\boxed{\sum_{j=0}^{n_{L+1}-1} w_{jk}^{(L+1)} \sigma'(z_j^{(L+1)}) \frac{\partial C}{\partial a_j^{(L+1)}}}$$

OR

$$\boxed{2(a_j^{(L)} - y_j)}$$

VC

This chain rule expression gives you the derivative that determine each component in the gradient that minimize the cost of the network by repeatedly stepping downhill.