→ "Training an MLP (Multilayer Perceptron) Chain Rule" (1)

The most important thing in "neural networks" is MLP (Multilayer Perceptron)

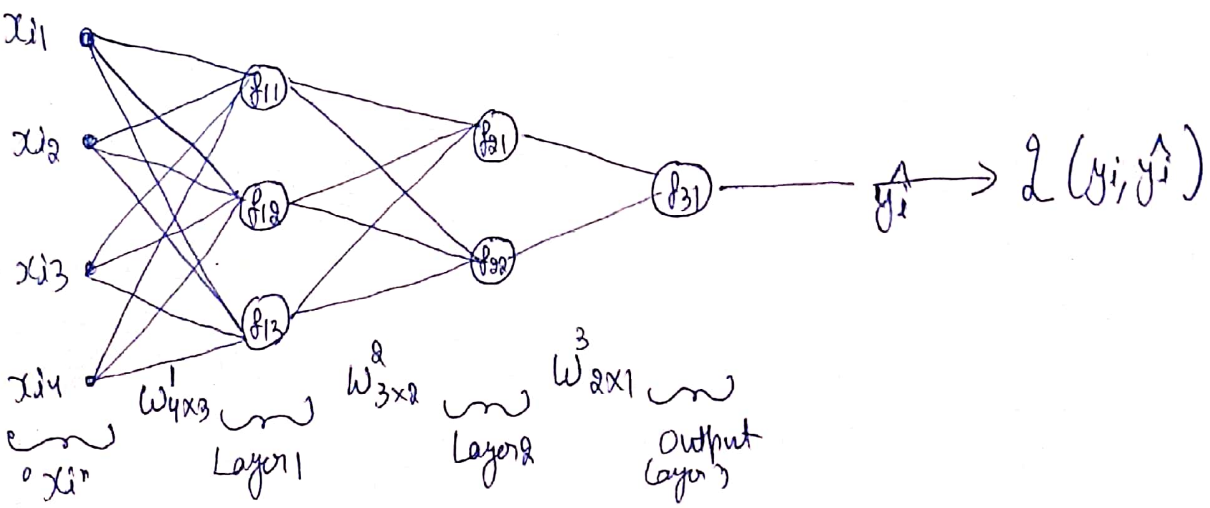
Let's say we have a dataset "D" comprising of x_i's & y_i's

$$D = \{x_i, y_i\}$$

$x_i \in \mathbb{R}^4$
$y_i \in \mathbb{R}$
} Here we want to train an MLP and this problem is a standard Regression problem

In standard regression problem, the simple loss function that we have is $\boxed{\text{"squared loss"}}$

Let's draw an MLP with the notations we saw earlier :-



Output is $\hat{y_i}$

We need a loss function, just like we saw in a single neuron model. So, $\hat{y_i}$ goes into a loss function represented by "L"

y_i's are from the dataset itself & $\hat{y_i}$ is the predicted values

PTO

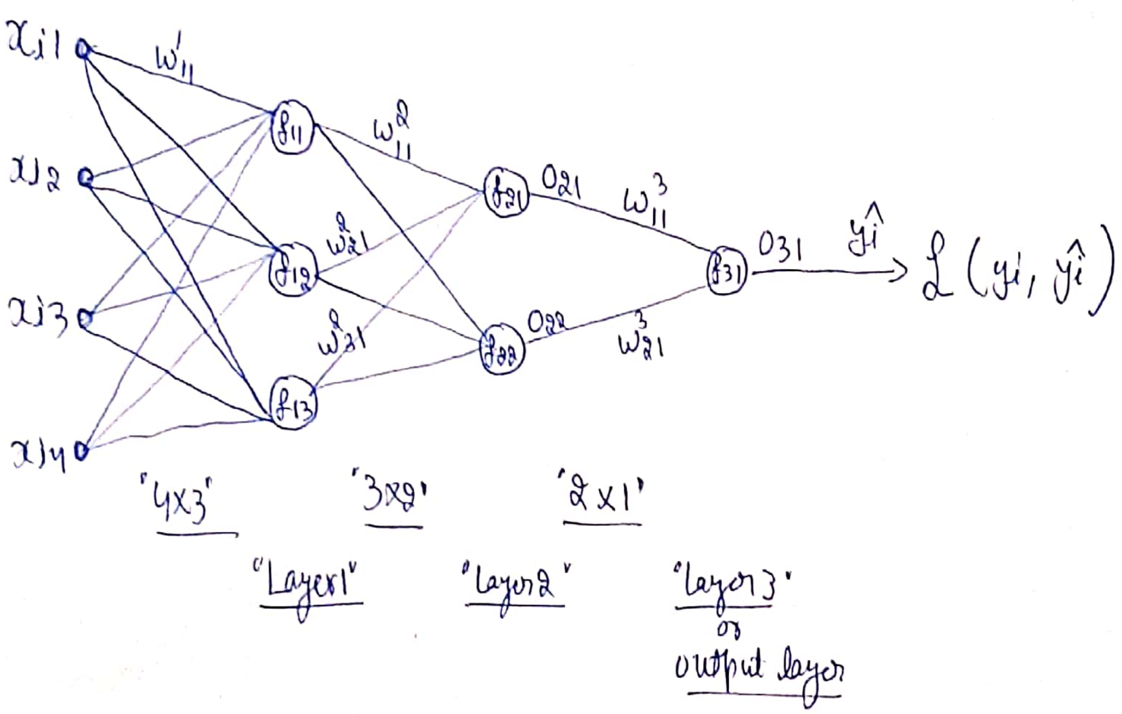
Weights in the first layer are represented using a 4×3 matrix

$$\left[W^{1}_{4\times3} \right]$$

In layer second, they are represented using $W^{2}_{3\times2}$

In layer third, they are represented using $W^{3}_{2\times1}$

The weights & output associated with the MLP as per the notation we saw earlier is as given below :-



Let's now try to pose this problem as an optimization problem:-

Training the perceptron (MLP) given a dataset $D = \{x_i, y_i\}$ basically means we need to determine the weights associated with the edges.

So, during training we need to determine the weights & they are represented as $\left[W^{1}_{4\times3},\ W^{2}_{3\times2}\ \&\ W^{3}_{2\times1} \right]$

So in total we have to determine these 20 weights & training basically means, computing these 20 weights.

lets now go step by step

Step no. 1 :- Define the loss function & the loss function here is the squared loss.

$$\mathcal{L} = \underbrace{\sum_{i=1}^{n} (y_i - \hat{y_i})^2}_{\text{squared loss}} + \text{regularizer}$$

(Note :- Regularizer is on the weights)

Note :- Any weight can be represented as $\boxed{W_{ij}^{K}}$

L_2 regularizer is defined on the weights as :-

$$\sum_{i,j,K} (W_{ij}^{K})^2$$

L_1 regularizer is defined on the weights as :-

$$\sum_{i,j,k} |W_{ij}^{k}| \rightarrow \text{absolute value}$$

just ignore regularizer for now, to make optimization problem simpler.

let us define $\mathcal{L}_i$ which is basically the squared error for a single training point.

$$\mathcal{L}_i = (y_i - \hat{y_i})^2$$ Now $\mathcal{L}$ can be written as :-

$$\left[\mathcal{L} = \sum_{i=1}^{n} \mathcal{L}_i + \text{regularizer}\right]$$

$$\mathcal{L} = \sum_{i=1}^{n} L_i + \text{regularizer.}$$

Step 2 : form an optimization problem.
Now the optimization problem can be written as :

$$\left[\min_{W^1, W^2, W^3} \mathcal{L} \quad \begin{matrix} \to \text{Squared loss} \\ \to \text{regularizer} \end{matrix} \right]$$

Minimize the squared distance between the "predicted values" and the "actual values", for all the weight matrices
Here "L" has two parts (squared loss & regularization)

In other words we can write this optimization problem as :-

Minimize Loss, for all the possible weights (W^k_{ij})

[for all possible values of $i, j + k$]

$$\downarrow$$

$$\boxed{\min_{W^k_{ij}} \mathcal{L}}$$

Where '$\mathcal{L}$' is nothing but the sum of "square loss" & "regularizer"

Step 3 ⇒ Apply "Gradient-Descent" to solve the optimization problem.

a) Initialization of weights W^k_{ij} randomly.

b) we need to iteratively keep on changing these weights

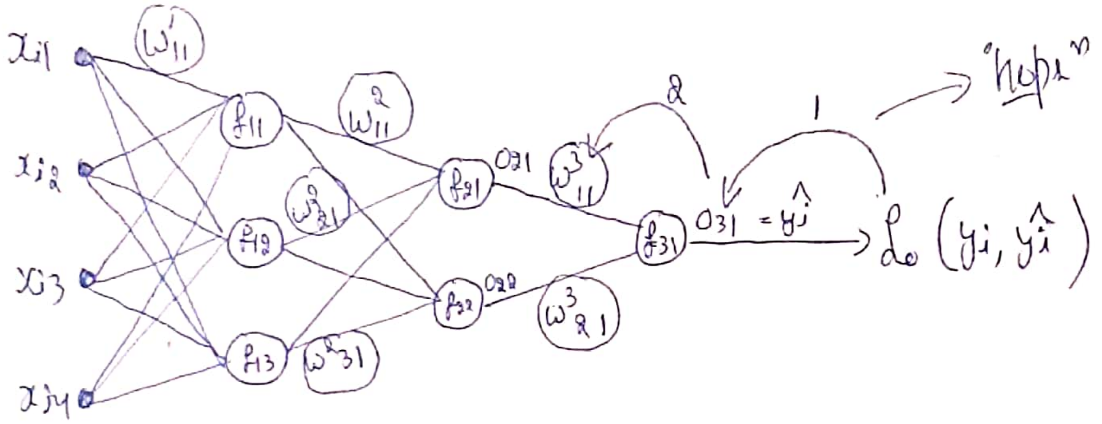
$$\left[(W^k_{ij})_{new} = (W^k_{ij})_{old} - \eta \frac{\partial \mathcal{L}}{\partial W^k_{ij}} \right] \to \text{update rule}$$

(c) Perform updates till convergence

Convergence means you keep changing w^k_{ij} till the time $(w^k_{ij})_{new}$ & $(w^k_{ij})_{old}$ are almost very very close.

Now the most important step in this is to compute the partial derivative. $\boxed{\frac{\partial L_o}{\partial w^k_{ij}}}$

Let's now compute these partial derivatives:



For simplicity let's just compute the derivatives of L_o with respect to the weights mentioned on these edges

$$\frac{\partial L}{\partial w^3_{11}}$$

Here w^3_{11} goes into f_{31} & it impacts O_{31} & O_{31} goes into L & it impacts L

$$\therefore \frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial w^3_{11}}$$

Similarly let's now compute $\frac{\partial L}{\partial w^3_{21}}$

$$\frac{\partial L}{\partial w^3_{21}} = \frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial w^3_{21}}$$

Here we are just following the path to compute these ["partial derivatives"]

So, we have computed the partial derivatives w.r.t to weights on third layer

Now let's compute partial derivatives w.r.t to the $W^2_{3\times 2}$ matrix (weight matrix)

$$\frac{\partial L}{\partial w^2_{11}} = \frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial O_{21}} * \frac{\partial O_{21}}{\partial w^2_{11}}$$

$$\frac{\partial L}{\partial w^2_{21}} = \frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial O_{21}} * \frac{\partial O_{21}}{\partial w^2_{21}}$$

$$\frac{\partial L}{\partial w^2_{31}} = \frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial O_{21}} * \frac{\partial O_{21}}{\partial w^2_{31}}$$

These are the weights in W_2

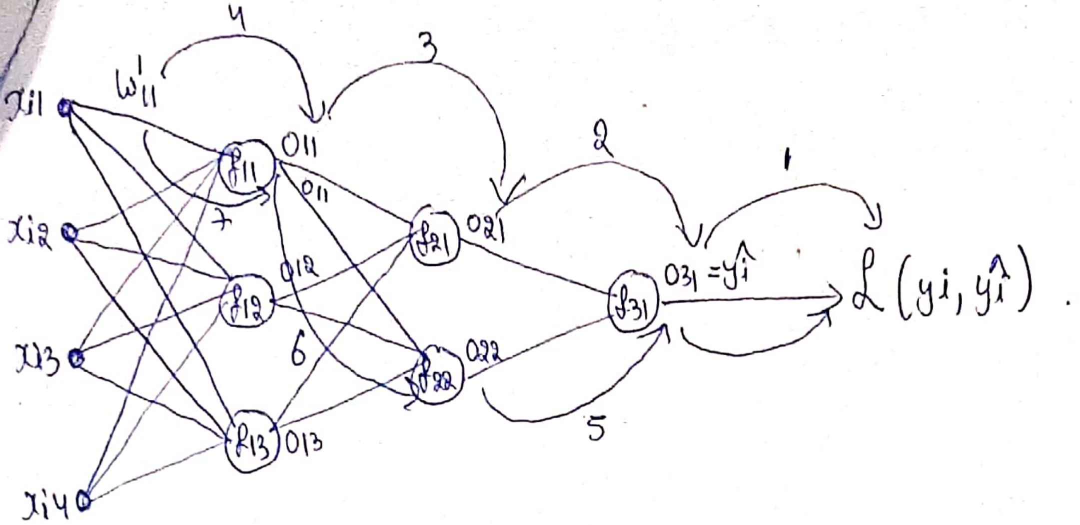
Although there are 6 weights in total, we have used only three weights

Similarly we can compute other weights also

Now let's compute partial derivatives w.r.t to the first weight matrix.

Let's compute partial derivative ∂L w.r.t w^1_{11}, This is little bit tricky

P.T.O



"x_i" input

we are computing $\frac{\partial L}{\partial w'_{11}}$.

Here we can see w'_{11} goes into 'f_{11}' & impacts O11 but O11 instead of going to a single neuron, goes as an input to two neurons. f_{21} & f_{22} & impacts both 'O_{21}' & 'O_{22}'

So, there are basically two paths from w'_{11} to $\mathcal{L}$.
∴ we have to choose another rule of chain rule process.
This can be computed using a simple rule in calculus.

Let's say we have a variable 'x' which goes & impacts two functions $f(x)$ & $g(x)$ & now let's assume we have a function 'h' which is impacted by both $f(x)$ & $g(x)$ as shown below:

$$\begin{tikzcd} & f(x) \arrow[dr] & \\ x \arrow[ur] \arrow[dr] & & h(f(x), g(x)) \arrow[r] & {} \\ & g(x) \arrow[ur] & \end{tikzcd}$$

'x' is an input to $f(x)$ which generates some output & this output goes into function 'h', similarly 'x' is an input to $g(x)$ which generates some output & this output again goes into function 'h'.

So, here we have two paths from ["x"] to ["h"].

$$\left[\frac{\partial h}{\partial x} = \boxed{\frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x}} + \boxed{\frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x}} \right]$$

"second path"

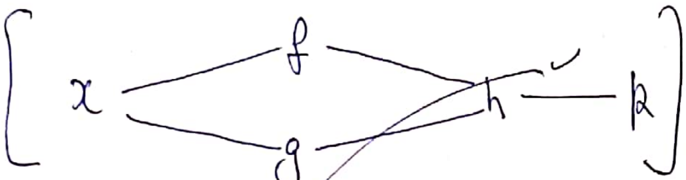
"One path"

This is becoz "h" is a function of "f" & "g"

But here in our case, the path from W^1_{11} is diverging at ["O_{11}"] & later these two paths are converging at ["O_{31}"].

Here we can see step 1 is common in both these paths.

These paths can be analyzed as



This is the structure that we actually have in the MLP.

$$\frac{\partial k}{\partial x} = \frac{\partial k}{\partial h} \cdot \frac{\partial h}{\partial x}$$

Now we have to figure out how to compute $\frac{\partial h}{\partial x}$ → for that we have two paths

$$\left[\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x} \right]$$

$\therefore$

$$\frac{\partial k}{\partial x} = \frac{\partial k}{\partial h} * \left[\frac{\partial h}{\partial f} * \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} * \frac{\partial g}{\partial x} \right]$$

Now let's use this similar concept to find $\frac{\partial L}{\partial w^1_{11}}$

PTO

Scanned by CamScanner

$$\frac{\partial L}{\partial w'_{11}} = \frac{\partial L}{\partial o_{31}} * \frac{\partial o_{31}}{\partial w'_{11}}$$ Now we need to compute this

$$\frac{d o_{31}}{\partial w_{11}} = \frac{\partial o_{31}}{\partial o_{21}} * \frac{\partial o_{21}}{\partial o_{11}} * \frac{\partial o_{11}}{\partial w'_{11}} + \frac{\partial o_{31}}{\partial o_{22}} * \frac{\partial o_{22}}{\partial o_{11}} * \frac{\partial o_{11}}{\partial w'_{11}}$$

$$\therefore \frac{\partial L}{\partial w'_{11}} = \frac{\partial L}{\partial o_{31}} * \left\{ \frac{\partial o_{31}}{\partial o_{21}} \cdot \frac{\partial o_{21}}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w'_{11}} + \frac{\partial o_{31}}{\partial o_{22}} \cdot \frac{\partial o_{22}}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w'_{11}} \right\}$$

Similarly we can compute other weights also belonging to ["first weight matrix"]

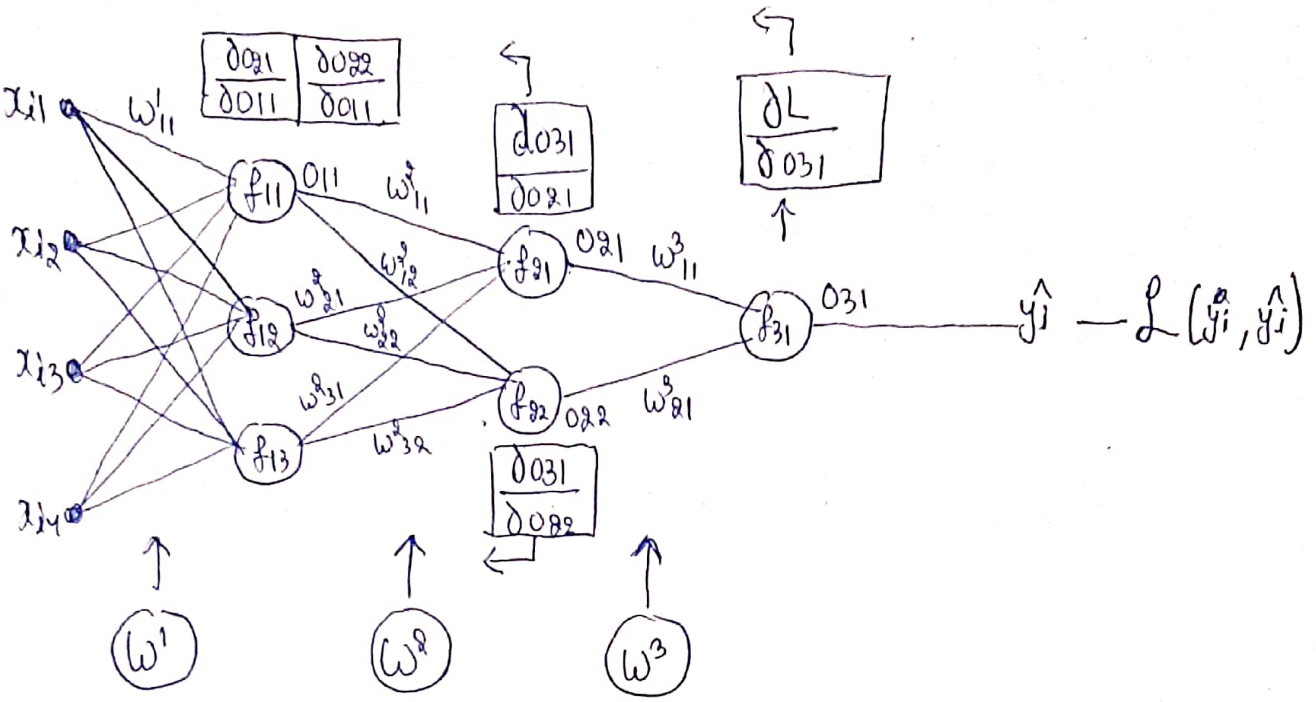
Now once we have computed all these derivatives we can quickly update the weights associated with the edges as:

$$\boxed{(w^k_{ij})_{new} = (w^k_{ij})_{old} - \eta \frac{\partial L}{\partial w^k_{ij\,old}}}$$

↳ This is how we train an MLP

P.T.O

imp ⇒ "Backpropagation"



Note :- Backpropagation is one of the most important algorithm in whole of deep learning.

let's now see how it works :-

Imagine we have a dataset D of points $\{x_i, y_i\}$

Now we will input every datapoint 'x_i' to the network just like any optimization problem, let's go step by step

① Initialize W^k_{ij} (weights) all the weights randomly.

Note :- There are various ways of initializing the weights.

② for each 'x_i' in D.

[Send each "x_i" through the network & here each "x_i" is having four features as it is a four dimensional data]

(a) pass x_i forward through the network

↳ This is called forward propagation.
(becoz we are taking the data & sending it forward in the network)

So, we input one data point & we get the loss using (y_i & $\hat{y}_i$)

Now using this loss we can compute all the derivatives to update the weights.

(b) Compute loss, $\mathcal{L}(\hat{y}_i, y_i)$

(c) Compute all the derivatives using Chain rule

(d) Update weights from end of the network to the start → This is called "back propagation"

So what we did, we send our input & passes it forward through the network & computed the loss, this is called forward propagation. Now we will do a backward propagation to update the weights from end of the network to the start.

The weights can be updated as:

$$(w^3_{11})_{new} = (w^3_{11})_{old} - \eta \left[\frac{\partial L}{\partial w^3_{11}}\right]_{(w^3_{11})_{old}}$$

This can be computed using the Chain rule.

$$\left[\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial w^3_{11}}\right]$$

Now once we have these derivatives, we can simply update the weights.

When we went from input to output, that is called.
["forward propagation".]

& when we are coming back, we are updating the weights using loss function & this is called ['Backward propagation".]

Note :- In forward propagation, we are sending the input & trying to compute the output which is $\hat{y_i}$ & In backward propagation, we are using the error or the loss $\mathcal{L}(y_i, \hat{y_i})$ to update the weights, so that next time, we send the input, these weights will be better tuned to give "lesser loss" or "lesser error".

③ Repeat step no. 2 till convergence.

Note :- At the end of step 2, all the data points have been seen once by the network.
There is a possibility that at the end of step no 2, still we have not converged so, we have to repeat step no 2 till convergence..

Convergence means $\{ (w^k_{ij})_{new} \approx (w^k_{ij})_{old} \}$

There is an important term called "epoch". So one "epoch" of training means ~~that~~ we have input all ~~the~~ points in our dataset once through the "neural network".

$\underbrace{D \rightarrow \{x_i ; y_i\} \longrightarrow \text{Network}}_{\text{This is called epoch}}$

[Passing all the datapoints through the network.

If we pass the datapoints five times through the network, it is called 5 epochs.

Note :- completing each of step 2 is basically completing one epoch.

In real world, for Neural Network training, we run the model for multiple epoch., which means we pass our dataset through the network multiple times.

Ideally each of the point "x_i" in our dataset be picked randomly in a uniform way. (we pick our points uniformly at random.)

Back propagation intuitively is as explained below :-

① Initialize the weights randomly

② epoch → "forward propagation"
→ "Compute loss"
→ "Compute derivatives (using chain rule)"
→ "Update weights using back propagation"

Repeat till convergence

Note :- Back propagation is a multiepoch training methodology where we leverage chain rule to update weights.

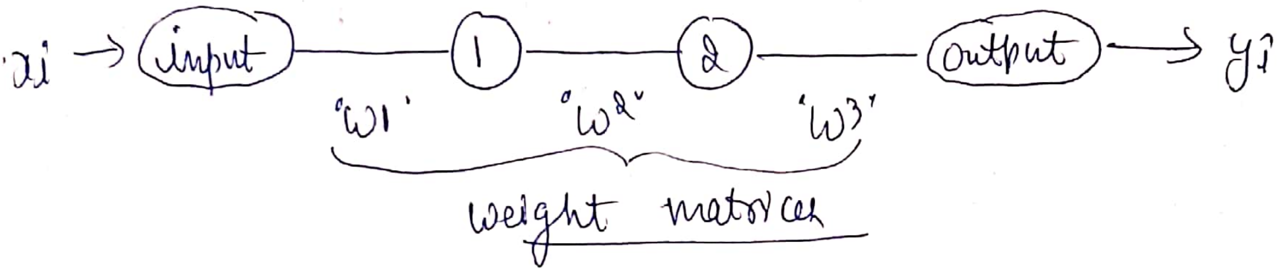
vvvimp
Note :- "Back propagation" is only applicable if the activation functions are differentiable.

P.T.O

Note:→ if the activation function is easily differentiable than we can speed up the training of the "Neural Network" using backpropagation.

Second important thing is the batch-size that can increase the training rate.

If we look at the MLP that we just have, it comprises of two hidden layers apart from input & output layers



In the steps we saw earlier, we are sending one point through the network at a time forward, computing the loss, computing the derivatives and updating the weights. This takes more time.

So, instead of sending one single point through the network why can't we send a batch of points through the network

Note:→ In 'SGD' we take one point at a time to compute the derivatives to update the weights.
Whereas in 'mini-batch' 'SGD' instead of taking one point at a time, we are taking a set of points to compute the derivatives

———— o ————