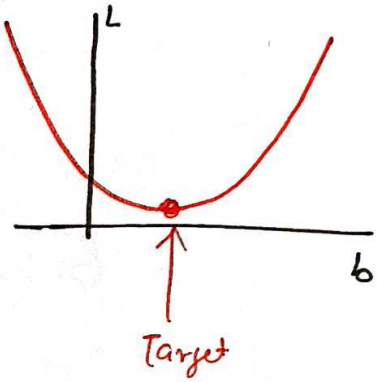
Gradient Descent

Gradient Descent: Gradient Descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks.

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t to the parameters. The learning rate α determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

$J(\theta)$ = Loss func
Loss (w, b)



Where we use gradient descent?

→ Backpropagation Algorithm

epochs = 5

for i in range (epochs):

    for j in range (X.Shape[0]):

        → Select 1 row (random)

        → Predict (using forward prop)

        → Calculate loss (using loss function → mse)

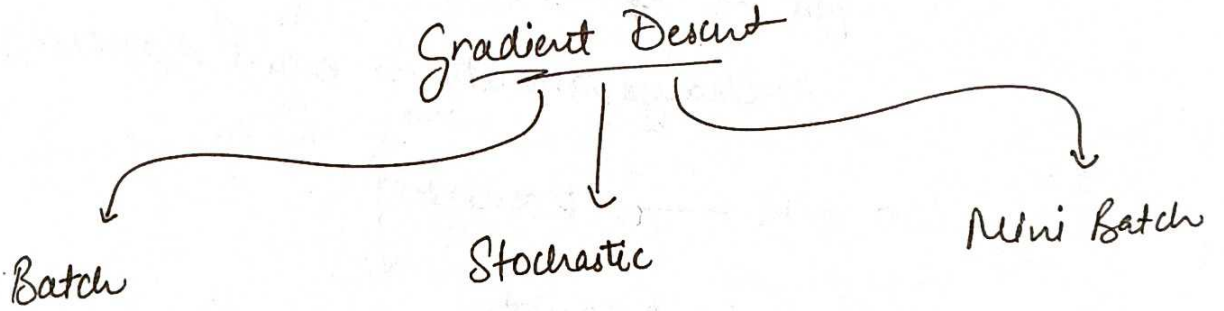
        → Update weight and bias using GD

$$w_n = w_0 - \eta \frac{\partial L}{\partial w}$$

    → Calculate avg loss for the epoch

    ↳ avg.

→ gradient descent



There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

① $\left\{\frac{dL}{\partial w}\right\}$ → derivation → type depend on how much data

(ii) accuracy ↔ time
↳ trade off

Batch Gradient Descent

```
for i in range (nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

In previous Gradient Descent code, we change the param for every row. for eg:- epochs → 50, 100 → data point

for → epochs:
for → data: → run 100 times and change parameter 100 times

But, in Batch Gradient we use entire data → 100
In a first epoch and change parameter in single time using 100 data.

for → epochs: (50)
change parameter → entire data (100)

* Only 50 time weights and bias update

100 data → predict

dot product

$\hat{y}$_hat = np.dot(X, w) + b

→ 50 points

Y = 50 actual point

$Y - \hat{Y}$ = loss

$\sum_{i=1}^{50}$

* Sare rows ke liye ek sath predict krege do 50 points ka hoga fir 50 point ka ek sath loss find krege

no of epoch = no. of update

Stochastic Gradient Descent

↳ 10 x 50 = 500 times parameter change

epoch = 10

for i in range (10):

→ shuffle data

* frequency of weight update is high

for j in range (X.shape[0]):
↳ 50

↳ 1 random point

↳ y-hat → forward

↳ Loss

↳ w, b update → $w_n = w_0 - \eta \frac{\partial L}{\partial w}$

↳ Avg loss print → for the epoch

Where, in Batch Gradient Descent only 10 times change parameter.
↳ epoch

which is faster?

↳ Batch GD is faster

code:

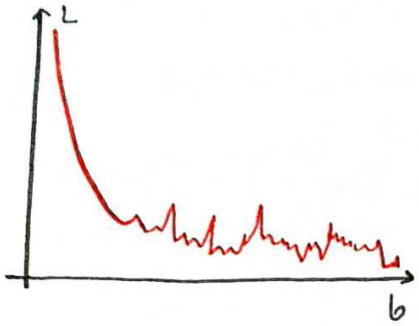
if Batch_size is equal to X.shape[0] then it is Batch GD.

if Batch_size is equal to 1. then it is Stochastic GD.

which is faster to reach answer?

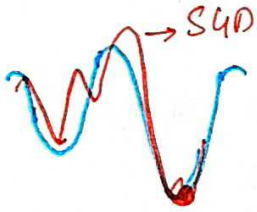
↳ Stochastic GD.

Stochastic GD



↳ Spikey SGD — Advantage

SGD → help the algo to move out of local minima cz of random spikey.

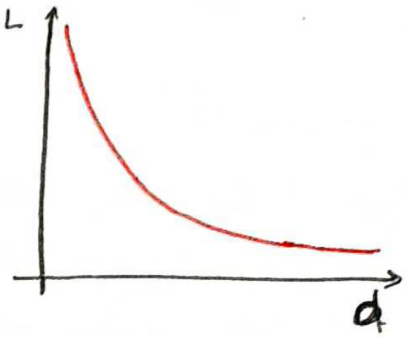


Disadvantage

↳ Not find exact solution. Find approx solution cz of spikey nature.

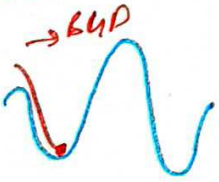
Difference

Batch GD



↳ Smooth BGD

BGD May be stuck in local minima




```
epoch = 10
for i in range(10):
    y = np.dot(x, w) + b
```

Vectorization technique

Batch GD use only epoch loop and not use another loop → and use dot product fo find the prediction.

↳ Smartest replacement to loop ↳ faster than loop.

This is called vectorization

Vectorization technique disadvantage

↳ if dataset → 10 crore

So we have to load dataset in RAM (2

BGD → find prediction → entire dataset

↳ higher memory usage.

Mini Batch Gradient Descent

↳ Best of Both

↙

BGD ↔ SGD

Example →

320 Rows → batch value → 32

↓

In every epoch

10 batches

update 10 times

for i in epochs: → shuffle data

for j in num of batch →

1 batch

↳ y_pred (Vectrization)

↳ loss

↳ update

batch = X

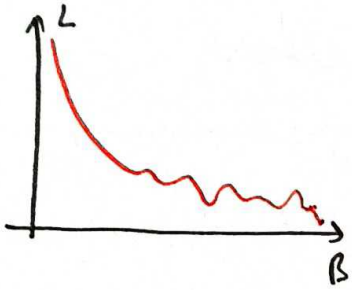
Total no. of row = n

batch-size = $\frac{n}{X}$ =

Make batch size

fast → BGD > Mini-BGD > SGD

Convergence → BGD < Mini-BGD < SGD



→ Why batch-size is provided in multiple of(2)?

2, 4, 8, 32, 64 - - - -

↳ Bcz RAM→effective
↳ optimization

→ What if batch-size doesn't divide # rows properly

↳ eg: no. of rows = 400
batch size = 150

no. of batch = $\frac{400}{150}$ = 2.66

first batch → 150

Second batch → 150

Third batch → left 100

Vanishing Gradient Problem

In machine learning, the vanishing gradient problem is encountered when training artificial neural networks with gradient-based learning methods and backpropagation. In such methods, during each iteration of training each of the neural network's weight recieves an update proportional to the partial derivative of the error function with respect to the current weight. The problem is that in some cases, the gradient will be vanishingly small, effective preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

2) $0.1 \times 0.1 \times 0.1 \times 0.1 = 0.0001$

2) Deep NN →  ← Vanishing Gradient Problem in Deep NN

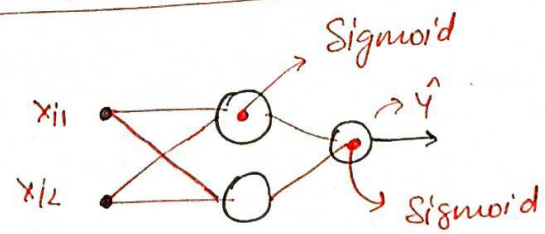
many hidden layer

3) Mostly Vanishing Gradient problem found in Sigmoid/tanh

derivative of sigmoid = 0 − 0.5

Problem

Backpropagation →



$$\frac{\partial L}{\partial w'_{11}} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial a_1} \times \frac{\partial a_{11}}{\partial w'_{11}}$$

Sigmoid

Sometime these no. b/c between 0-1

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w}$$

derivative of L w.r.t weight

0-1 0-1 0-1 0-1

Very small = multiply of these no. is very small ≈ 0.0001

eg:- $W_{old} = 1$ $\eta = 0.01$ and $\frac{\partial L}{\partial w} = 0.0001$

↳ calculate derivate

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial w}$$

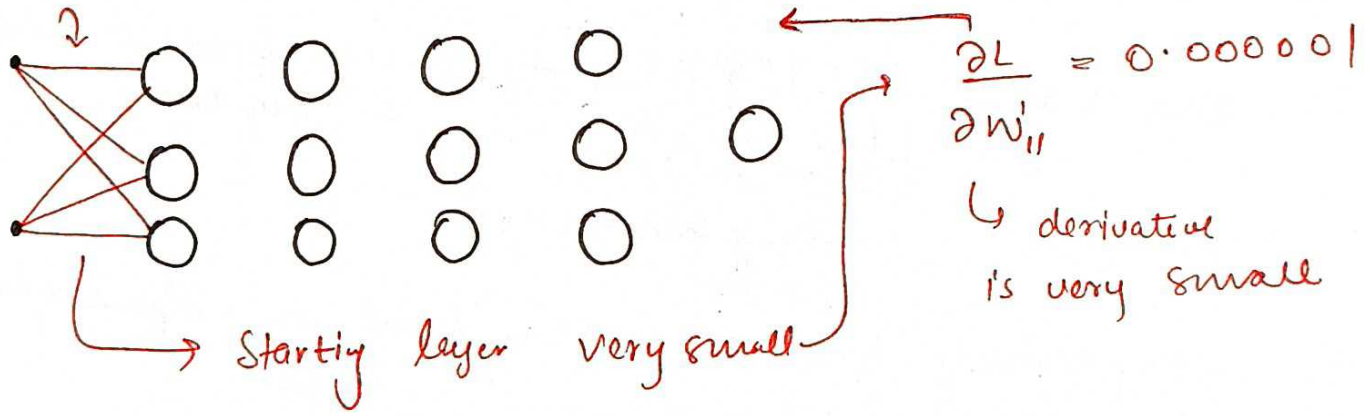
$$W_{new} = 1 - 0.001 \times 0.0001$$

$$\boxed{W_{new} = 0.99999}$$ → which means there is no weight change. old is 1. and new is 0.99999.

* If weights didn't change then Loss is same.

↳ $L = Y - \hat{Y}$

eg: Previous example → only 1 hidden layer
what if 4 hidden layer.



Backpropagation → cannot converge

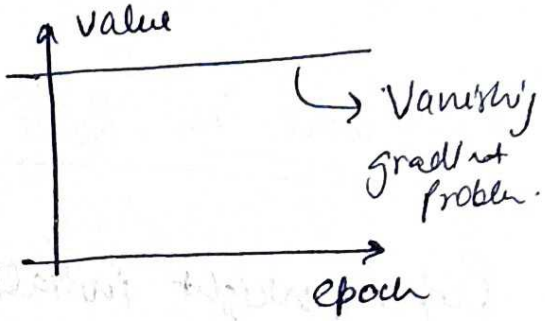
↳ model traiyd ✗

Sigmoid fun' → kitna bhi bada input ho ya kitna bhi chhota input ho usse 0-1 ke beech mai leke ata hai. Isse Vanishig Problem hoti hai.

How to recognize Vanishing Gradient Problem?

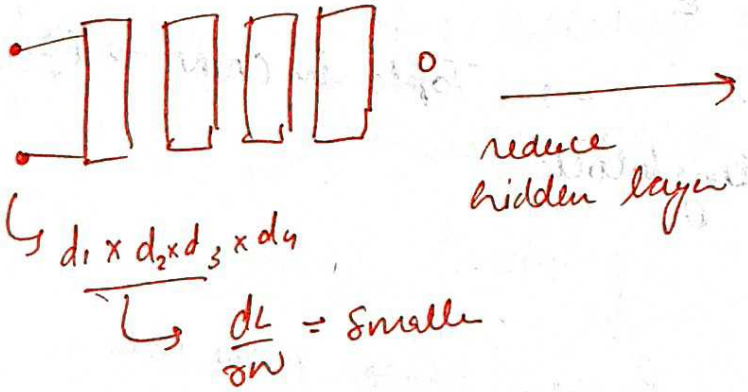
1) Loss focus → epoch → no changes in Loss
   ↳ Cz Vanishing gradient Problem.

2) Weights → Graph



How to reduce or handle Vanishing Gradient Problem →

1) Reduce model complexity — but this is not applicable



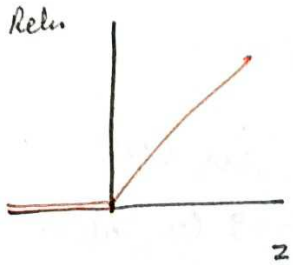
Sometimes this method is good but we increase hidden layer for complex pattern and find more accurate prediction. So, this method is not more useful.

2) Using ReLU Activation function

ReLU → (0-z) → any value. 1000, 2000

$R(0, z)$ → if value is negative then convert change into 0
if value is positive do value don't change

In sigmoid → every value convert into (0-1)



→ derivative → if value is -ve = 0
→ if value is +ve = 1

disadvantage → dying Relu → if all derivative is -ve = 0
then W_n is equal to W_0

↳ Discussed in ReLU Topic

3) Proper Weight initialize → Glorot
↳ Xavier
↳ future topic

4) Batch Normalization → future topic

5) Residual Network → Topic in CNN → RESNET
↳ building block

Exploading Gradient Problem → RNN Problem.
↳ small introduction

derivative > 1.
↳ (10), (10), (10), (10) → 10000

$\frac{\partial L}{\partial w_{11}} = d_1 \times d_2 \times d_3 \times d_3$ (big number)

* our model weight increasing highly.
At some point, our model behave randomly. And Loss ✗ not reduce

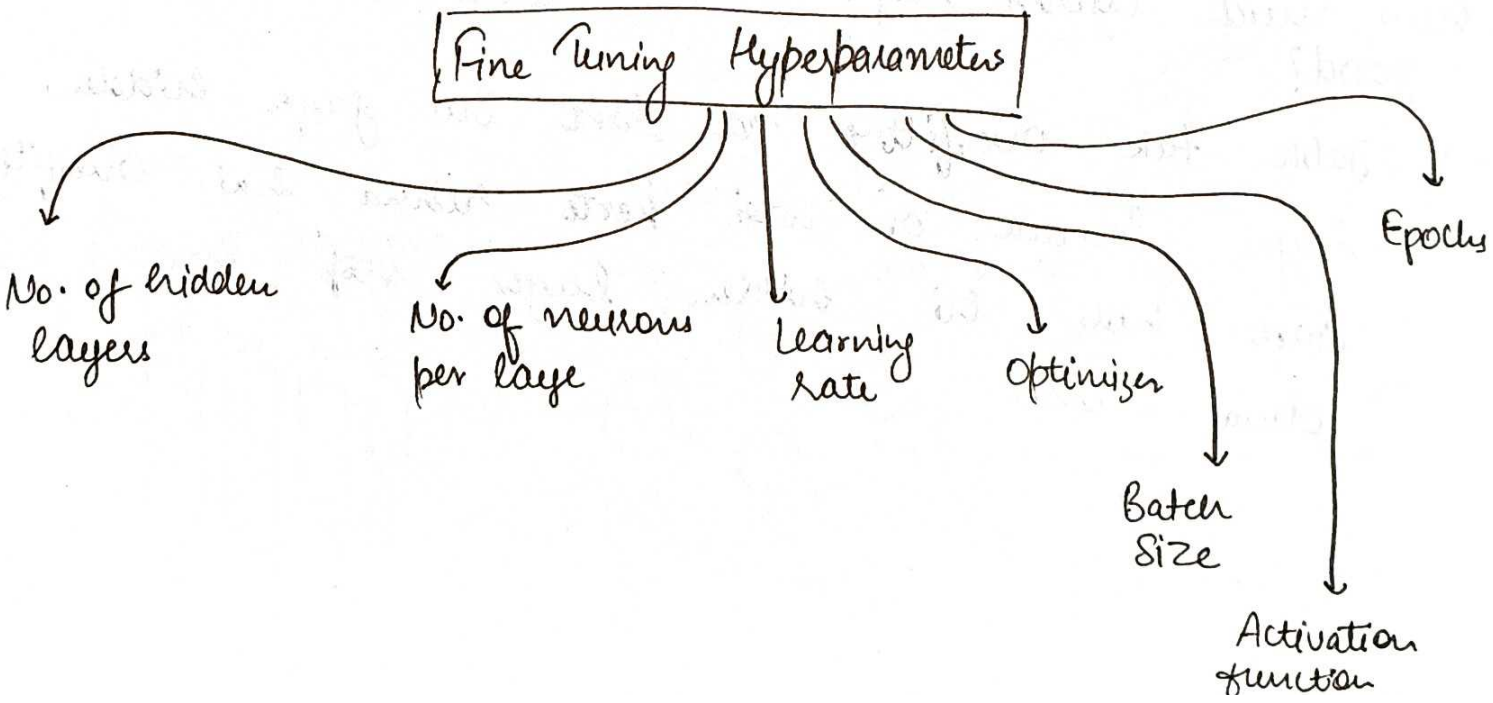
$W_n = W_0 - \eta \frac{\partial L}{\partial w}$ (1000, 0.1, 1)

$W_n = 99$

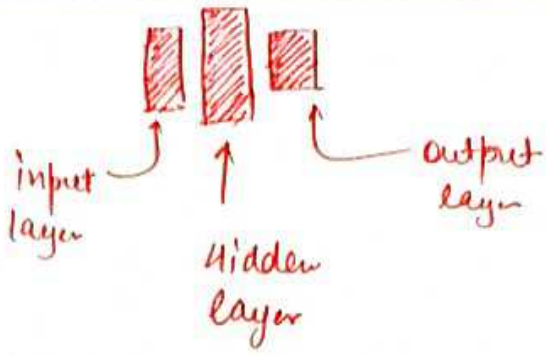
How to Improve a Neural Network? → Kind of Roadmap

1. Fine tuning NN hyperparameter
   - ↳ eg:- 1. How many hidden layers?
     2. No. of Neurons per layer
     3. Learning Rate → GP
     4. Optimizer
     5. Batch Size
     6. Activation
     7. Epochs

2. By Solving Problems:
   - → Vanishing / Exploding gradient
   - → Not Enough Data
   - → Slow Training
   - → Overfitting

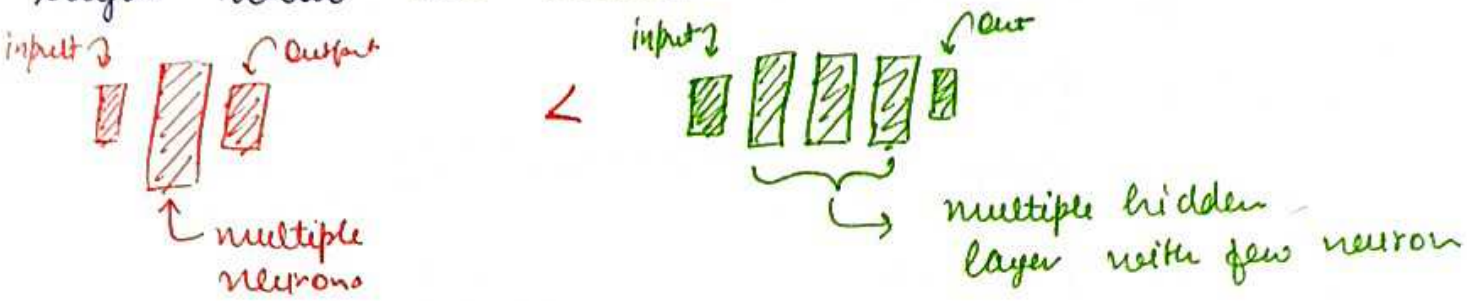


1) No. of hidden layers



1 hidden layer with 512 neurons ⟶ complex

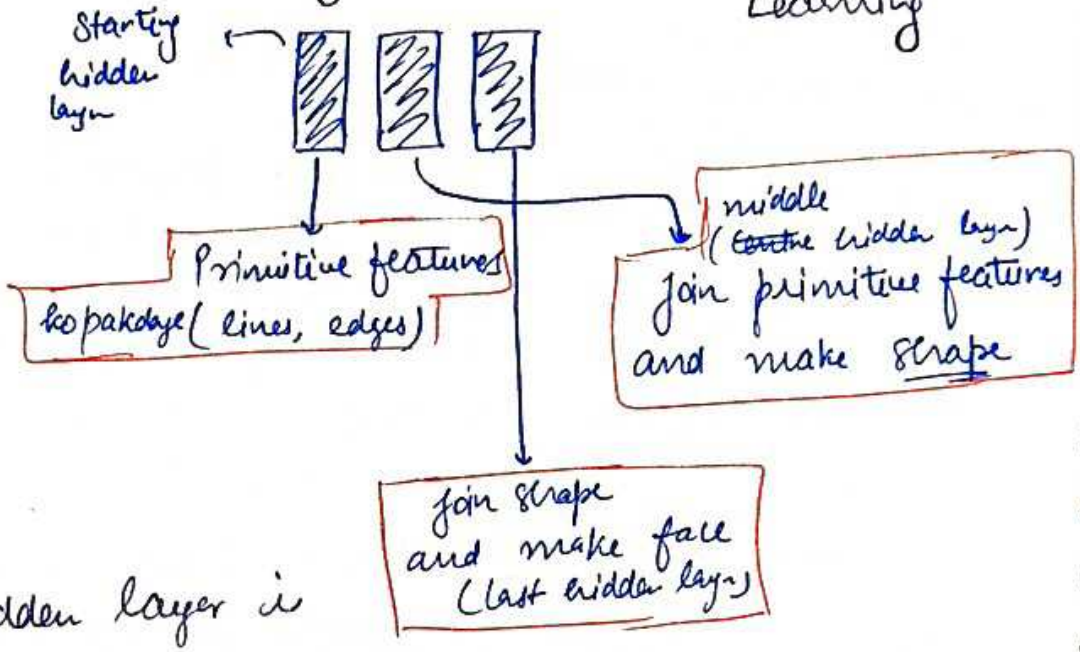
less better than multiple layer with less neuron.



This is because Deep Learning uses ⟶ Representation Learning

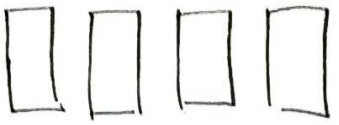
eg:-

face ↑



how much hidden layer is good?

→ Jabh tak overfitting na start ho jaye hidden layer increase or add karte rehna hai. Overfitting start hote hi hidden layer stop kar dena hai.



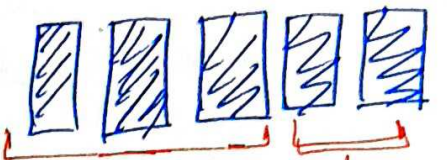
→ hidden layer

hidden layer mai jaise aage jaate rahe hai complex pattern samjh rahe hai. And this is a advantage

This advantage is called Transfer Learning.

Model → Human face → detection

→ reuse → Monkey face



Line and Shape Same

Took last layer and Train on Monkey face.

2 Neuron

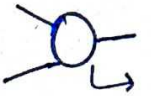
Input layer

eg:- cgpa / iq / placed

cgpa —O
iq —O
} depend on input

Output layer

(i) Regression and classification prob

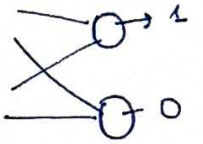


→ only one output neuron

(ii) Multiclass classification

→ depend on output multiclass

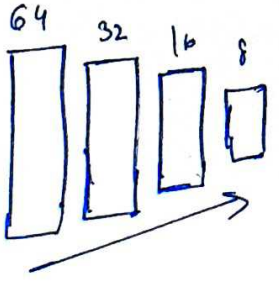
eg:



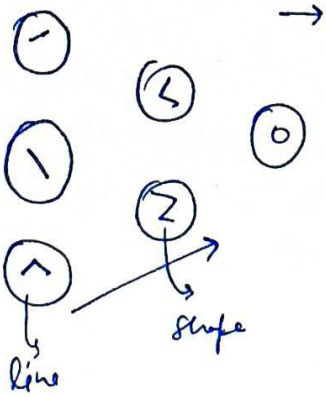
2 classification

Hidden layer

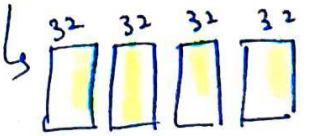
(i) Pyramid Structure



→

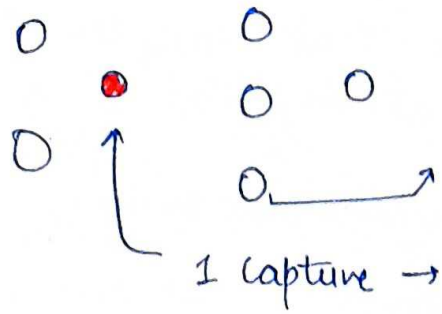


→ * If we don't make pyramid structure still our prediction is good.



* No. of must be sufficient

eg:-



In this layer we can't recover the data

1 capture → highly chance to loss the data

* No. of node should be more than what is require
  ↳ if facing overfitting with more neuron then reduce some neuron.

3. Batch size → Batch → all row ⇒ weight update
  ↳ Stochastic → 1 row ⇒ weight update
  ↳ Mini Batch → (32) row → 1 update
  ↳ hyperparameter.

Slower ← Smaller (8 to 32) ← Generalize result better ← New data

Large (8192) → fast
  ↳ we can also get better result
  → use small learning rate at starting epochs then increase the learning rate
  ↳ depend on GPU and RAM

4. Epochs → 100 → 500 → 1000
  ↳ early stopping → keras callback feature
    ↳ keras → Stable ~~accuracy~~ accuracy → Stop