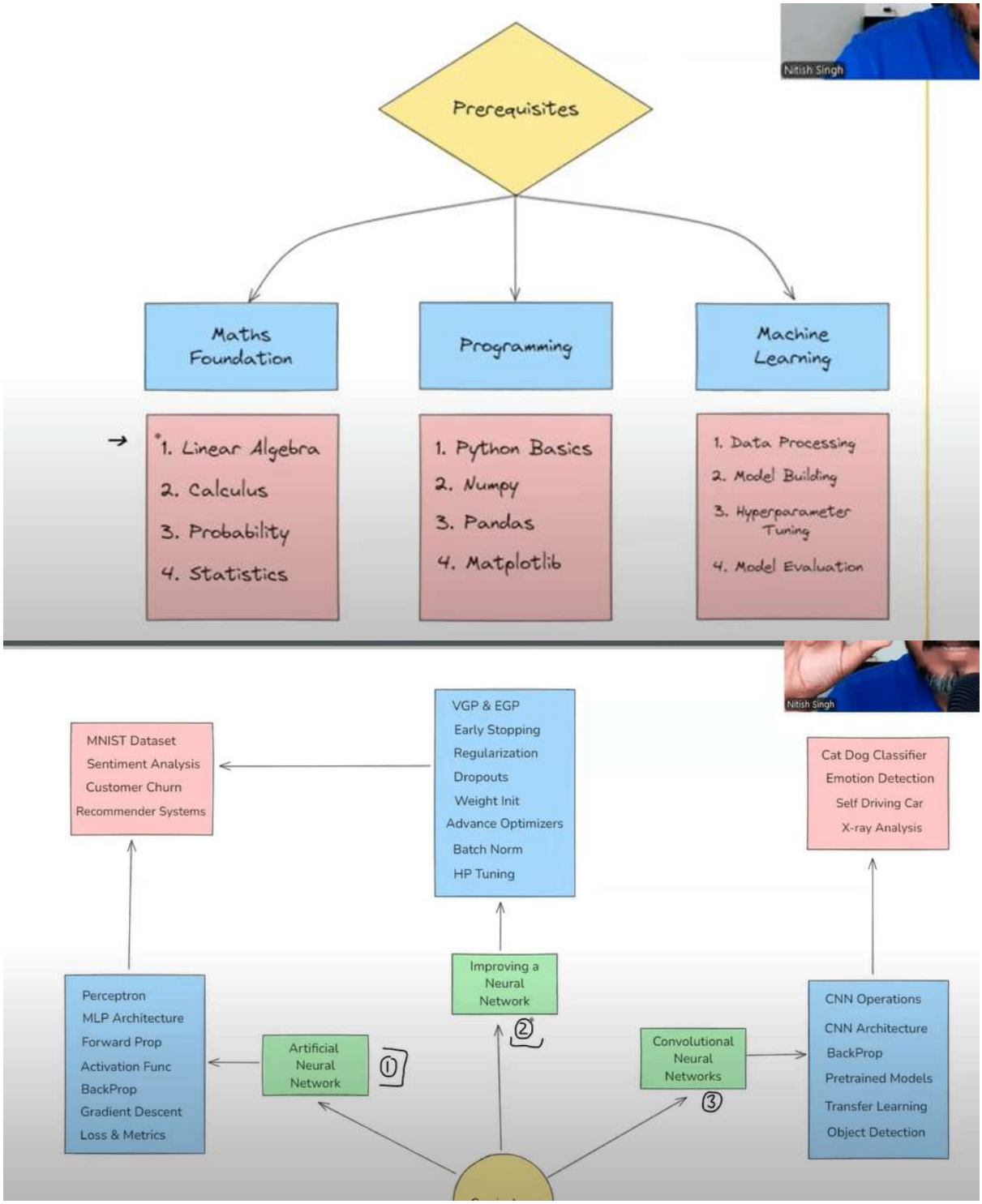
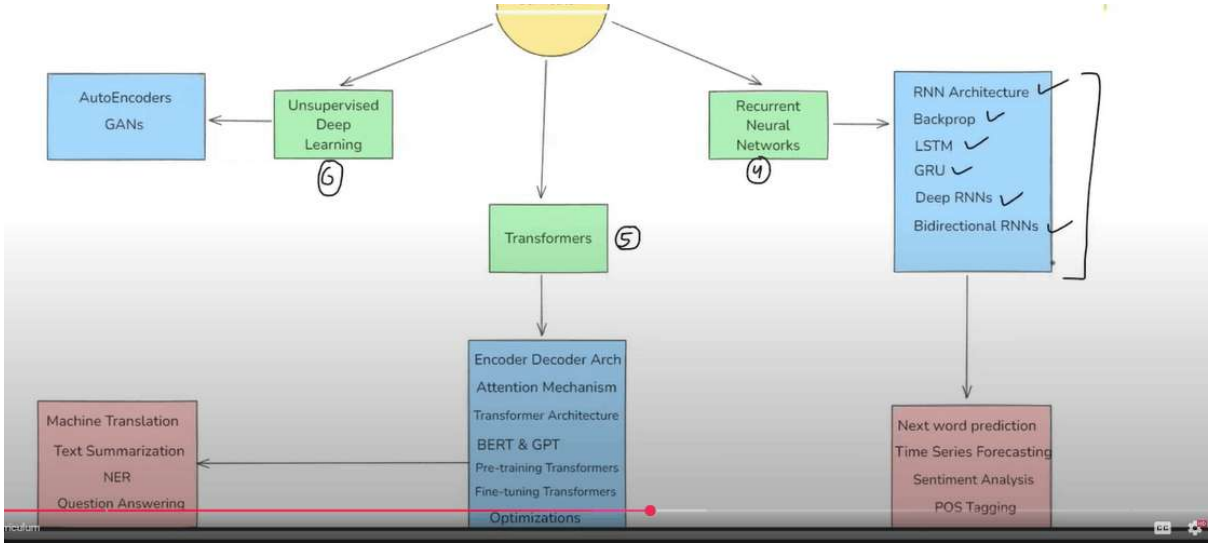
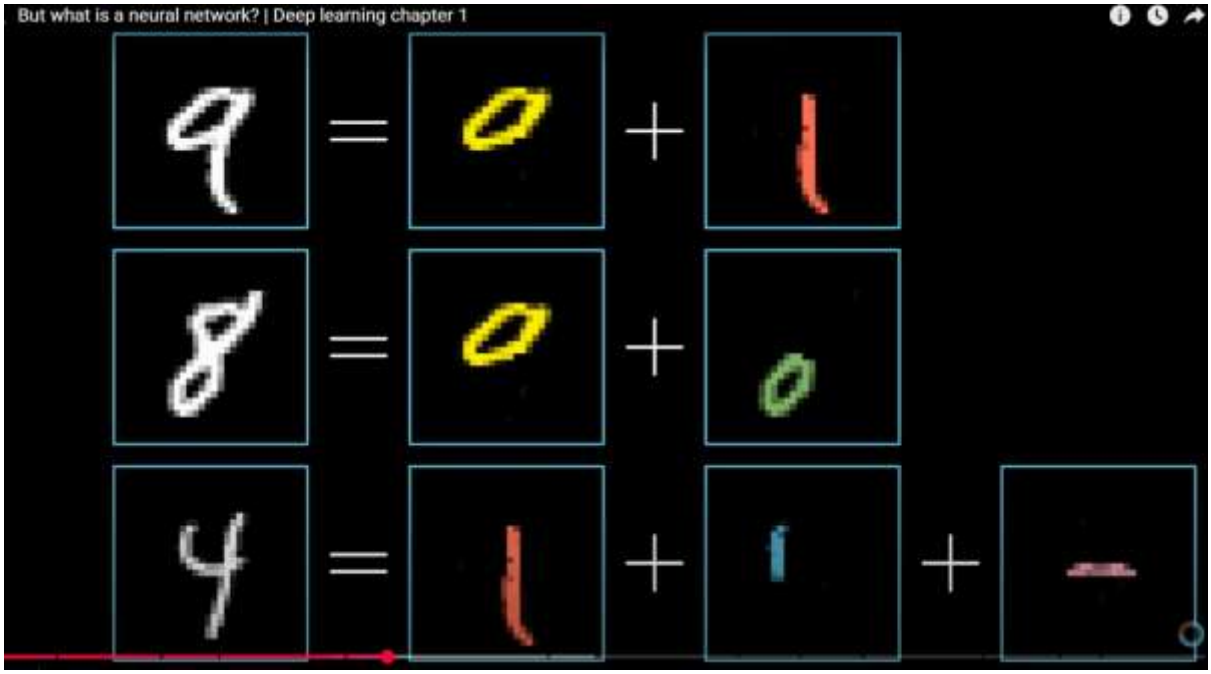
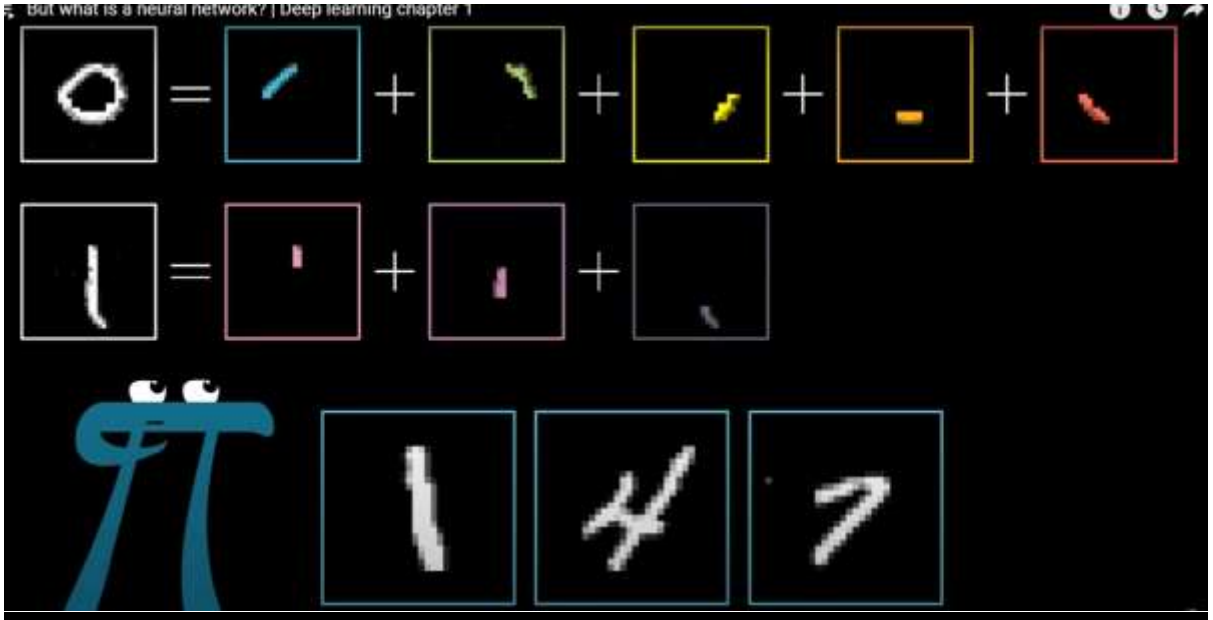
www.educosys.com/course/genai

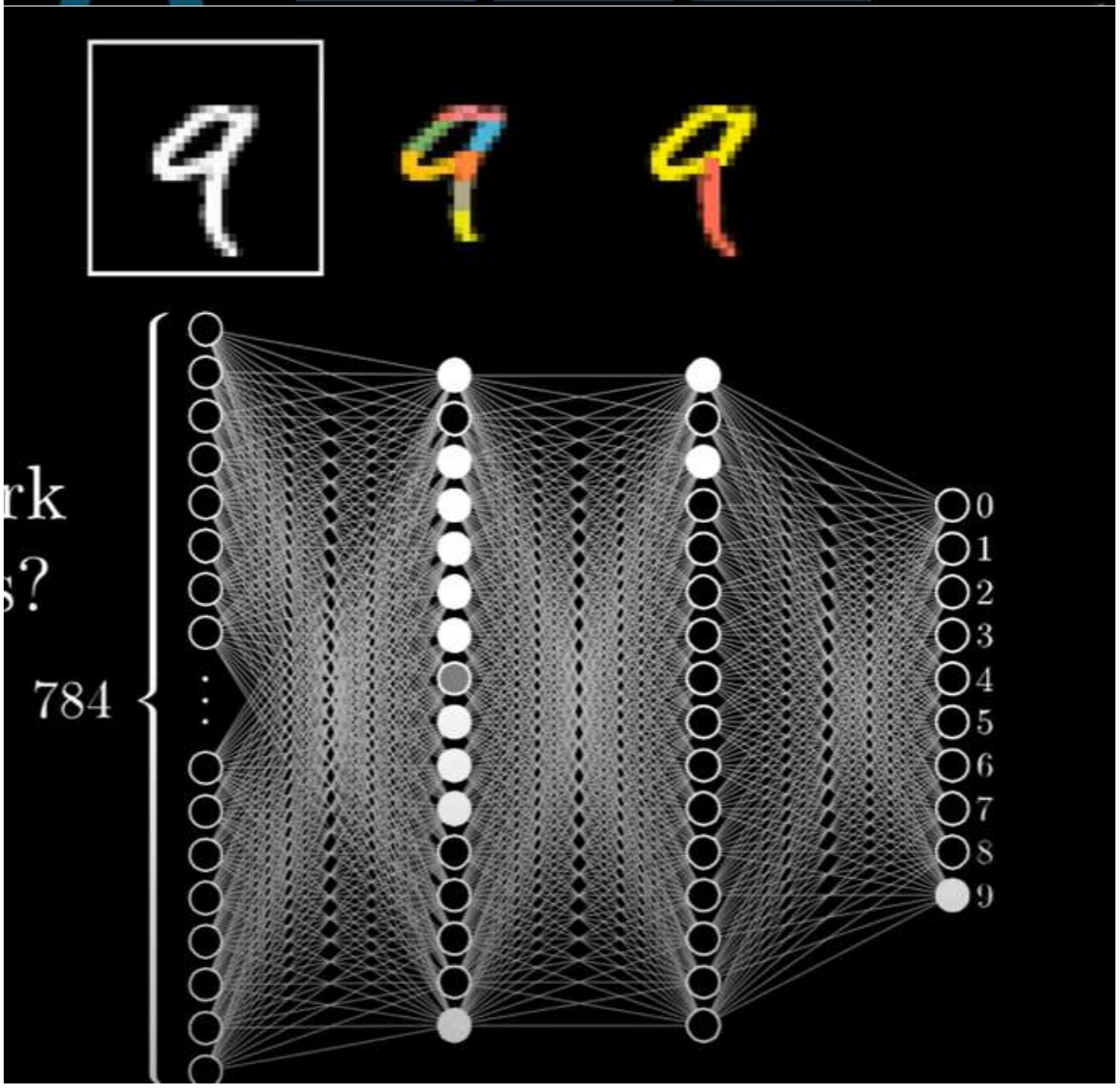


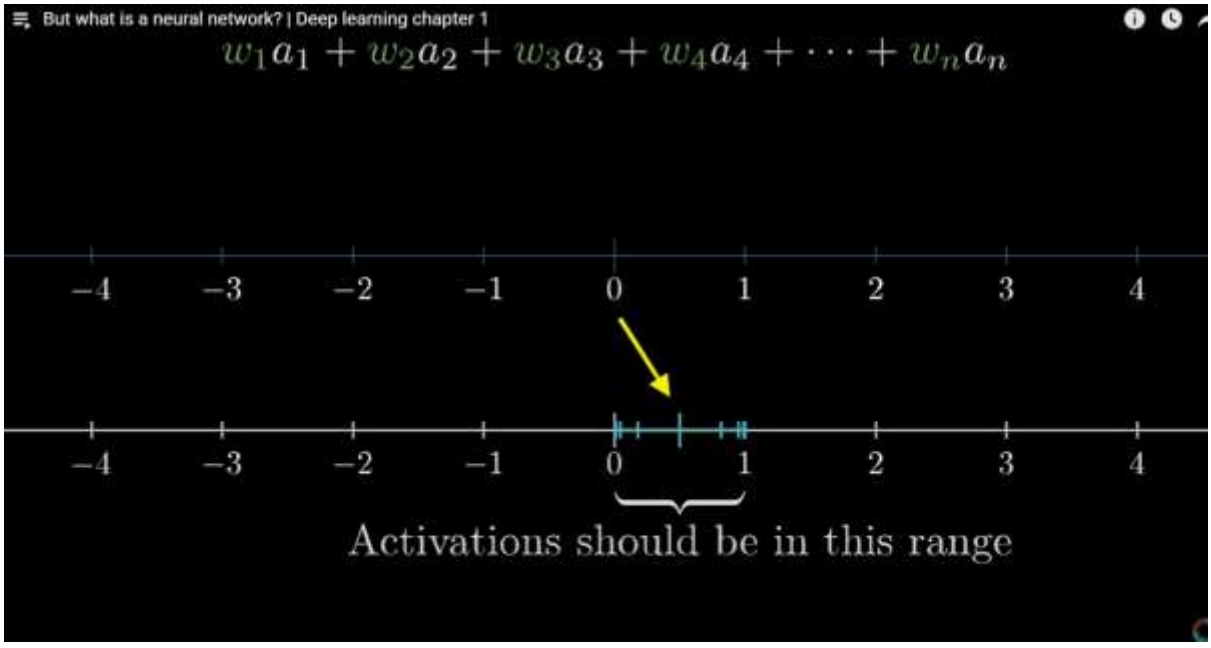


number inside neuron is called activation




But what is a neural network? | Deep learning chapter 1


rk
s?
784
0
1
2
3
4
5
6
7
8
9



we want to recognize 8 suppose

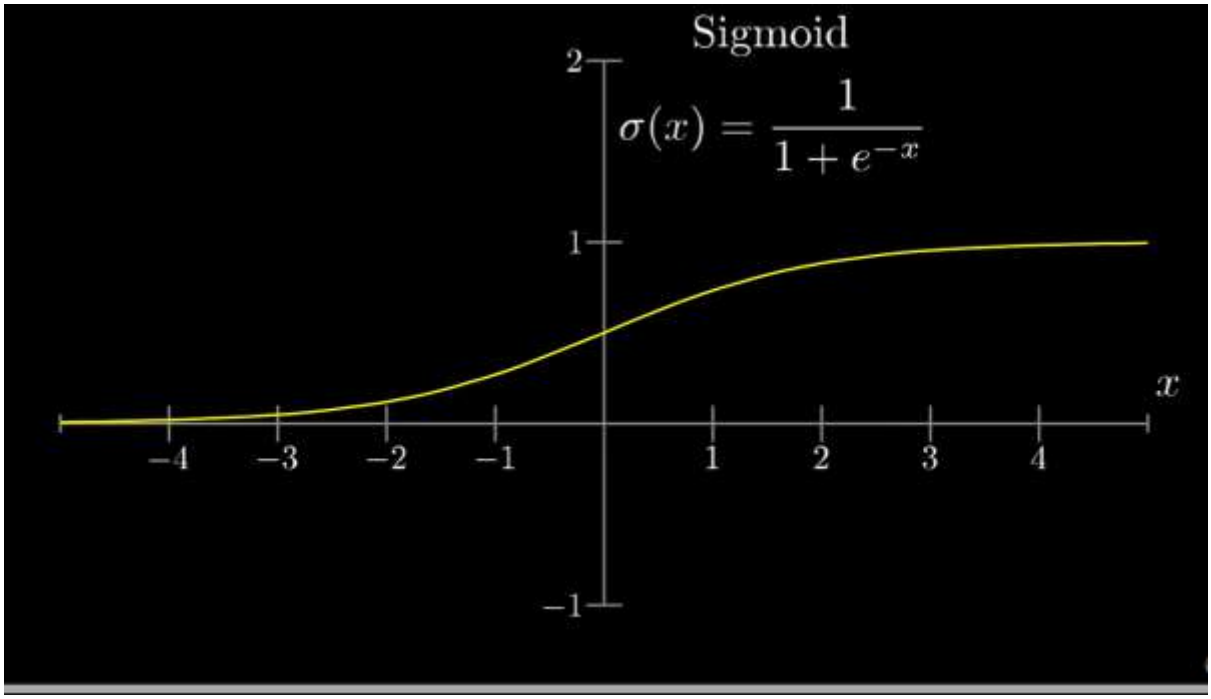
so we multiple the input image pattern with 8's pattern

and it will light up only neurons that match in both grid

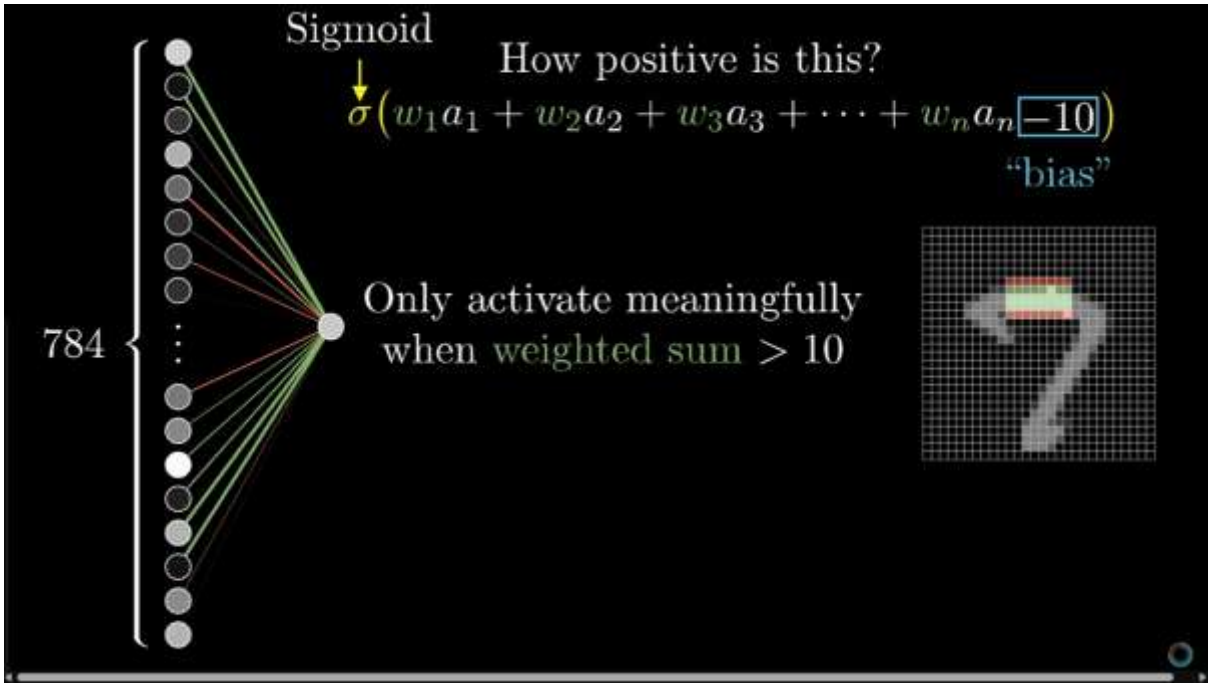
so only those will have higer number

after multiplication value can be anything

but what we want is vlaue between 0 and 1 so for this we use sigmoid

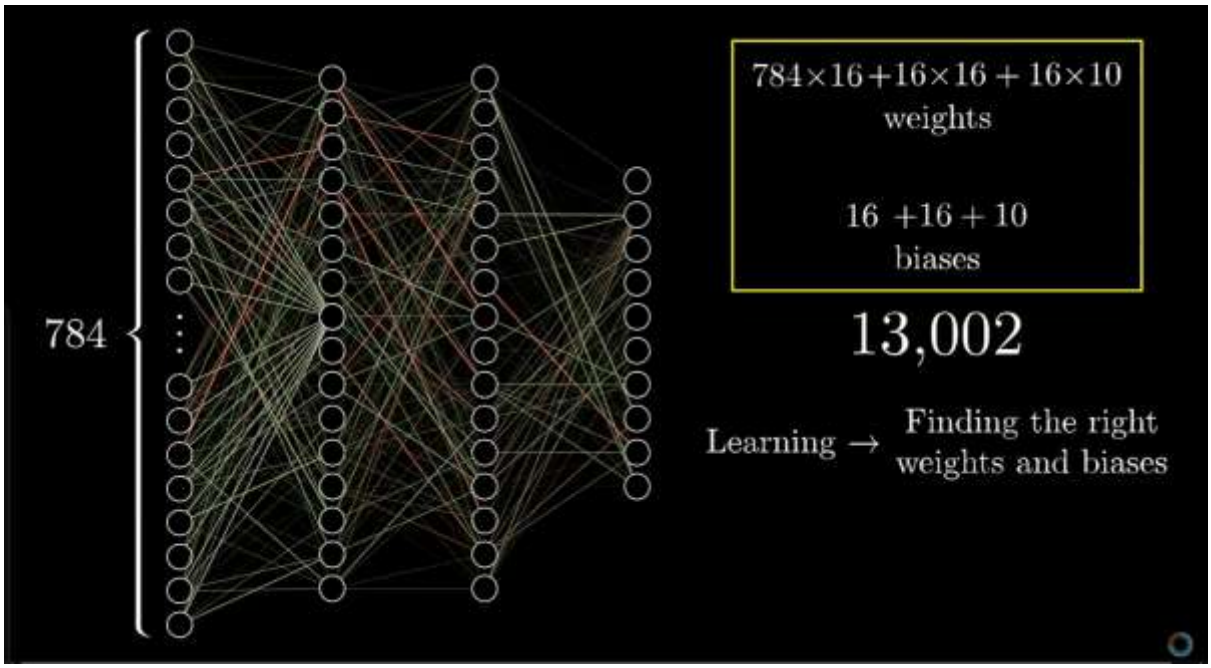


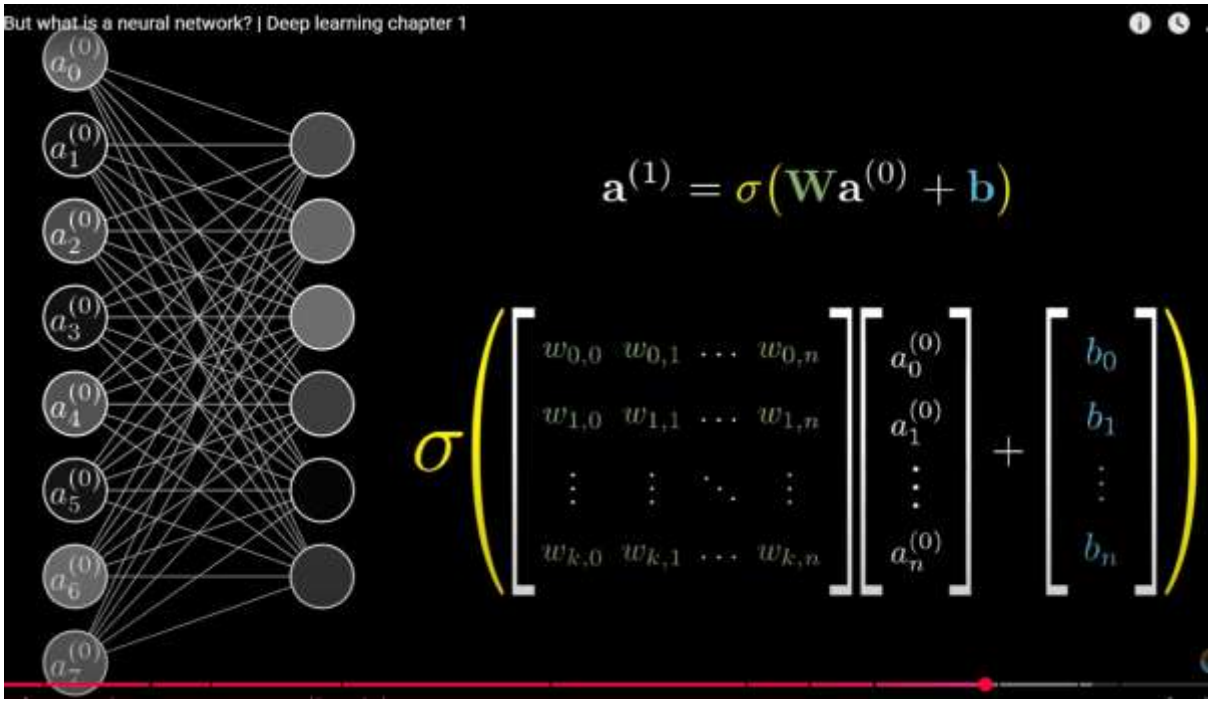
logistic curve



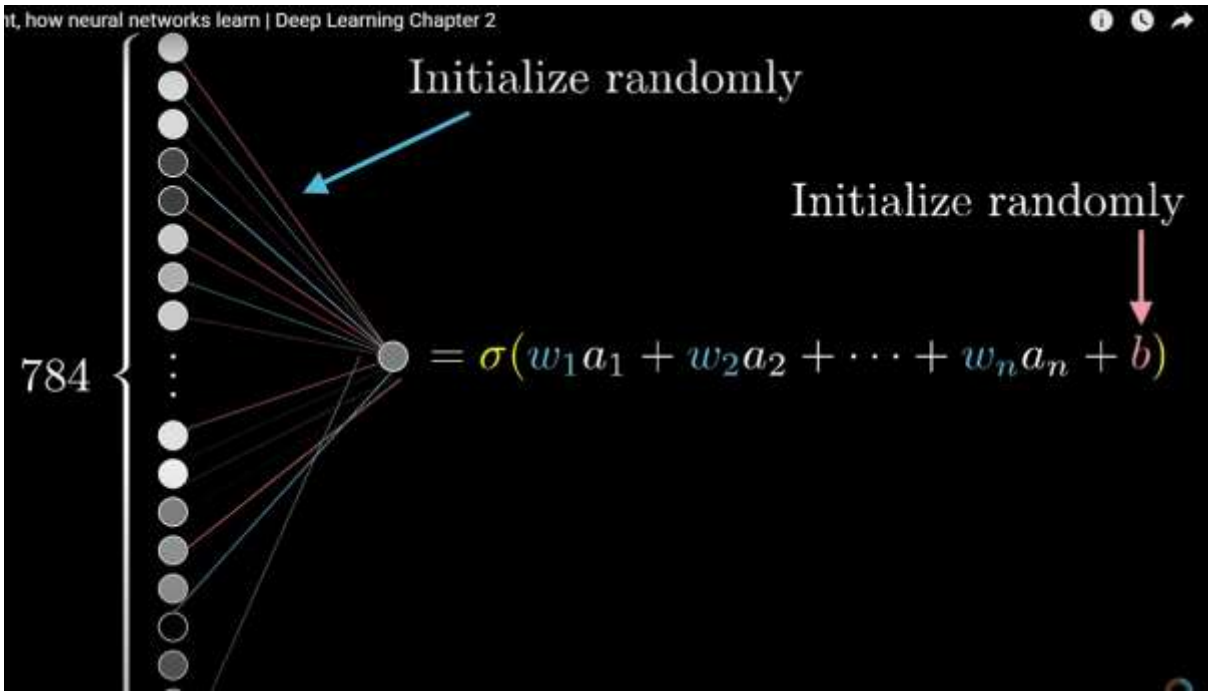
10 mean it is meaningfully active

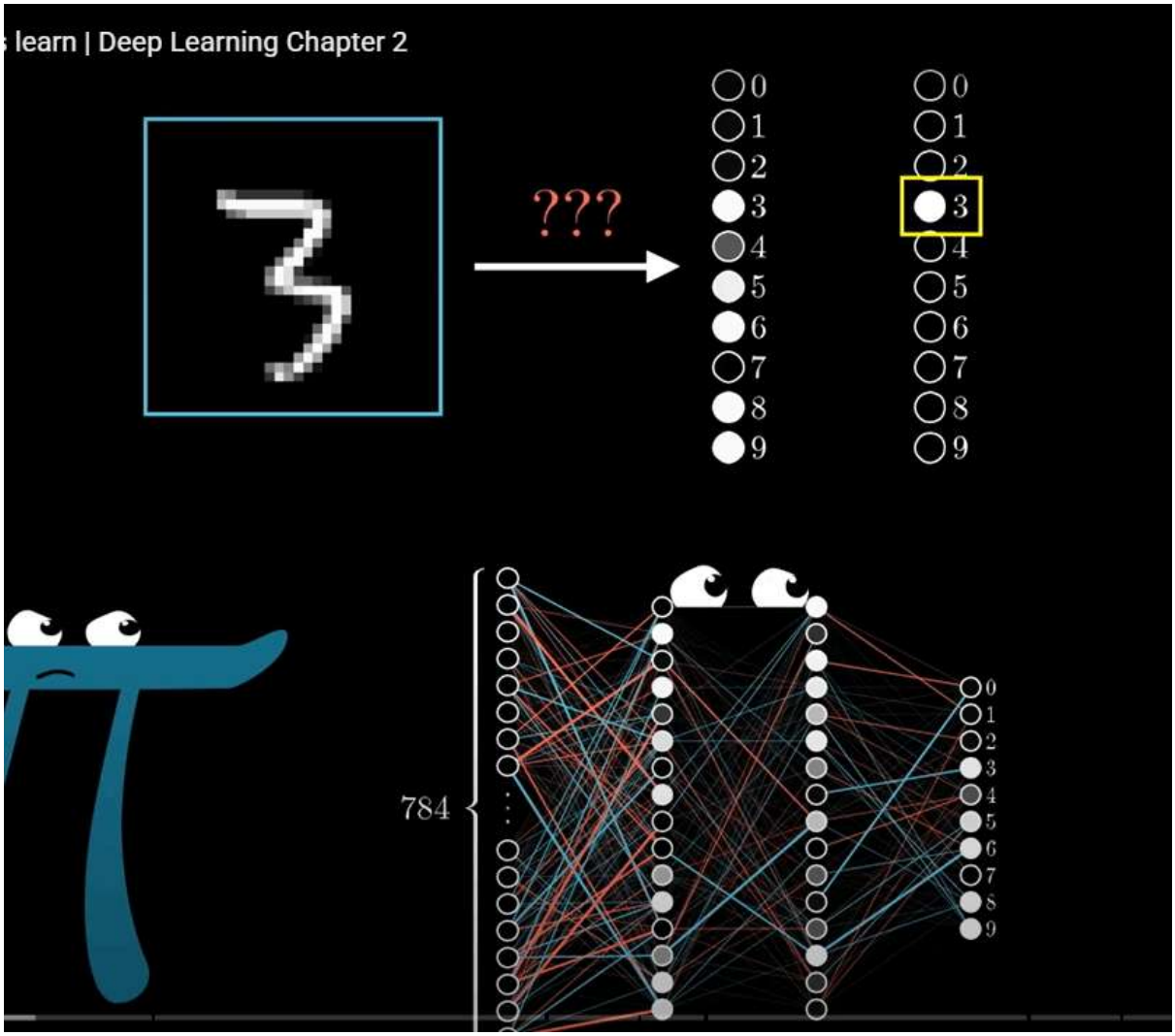
so we keep that bias





but how does neural network find pattern and learn it ??

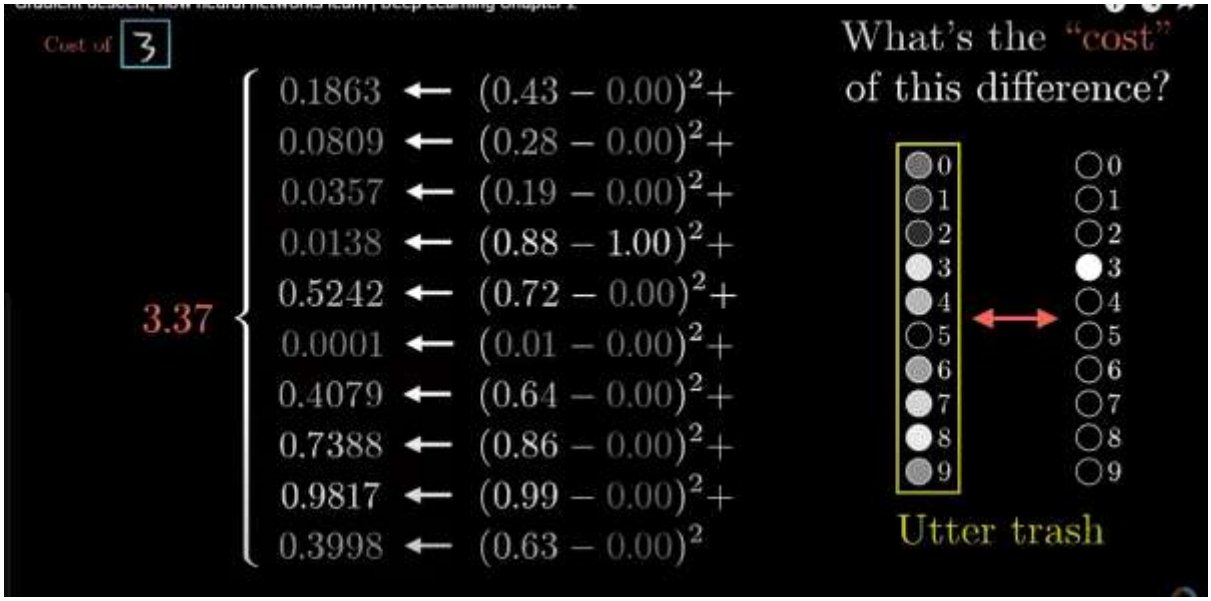




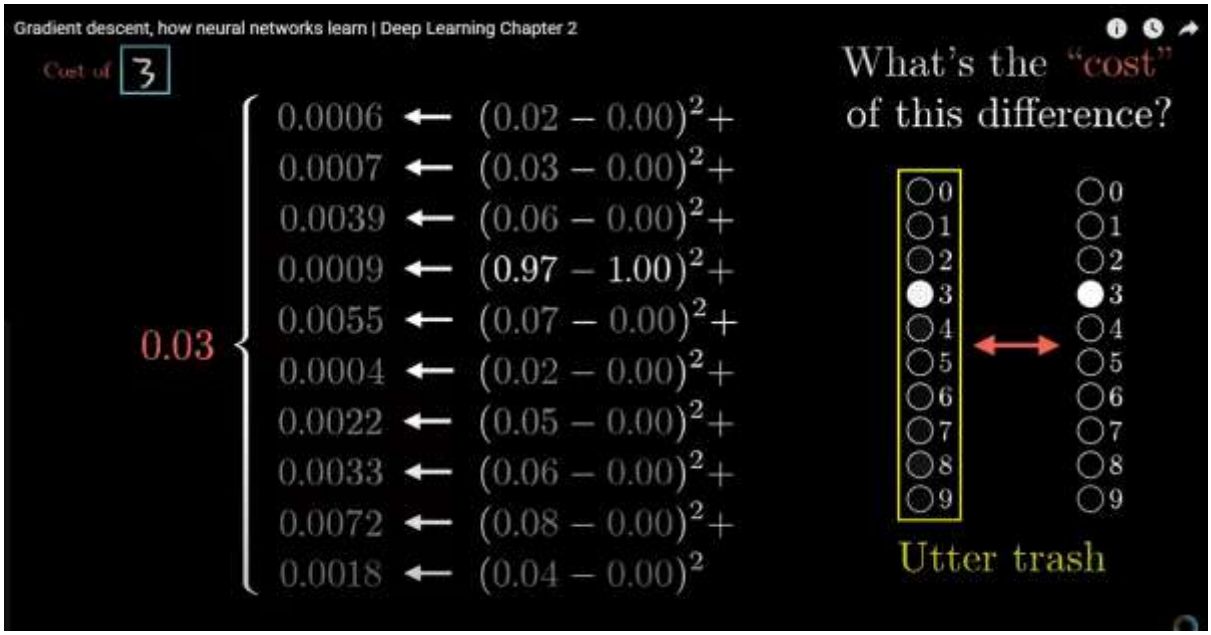
to the left is what network gave with 3,5,6,8,9 ON and what i want is only 3 to be 1 and all other 0

so i say network - bad boy u gave me wring output

so to say this mathematically that what u gave is not what i desired is by MSE

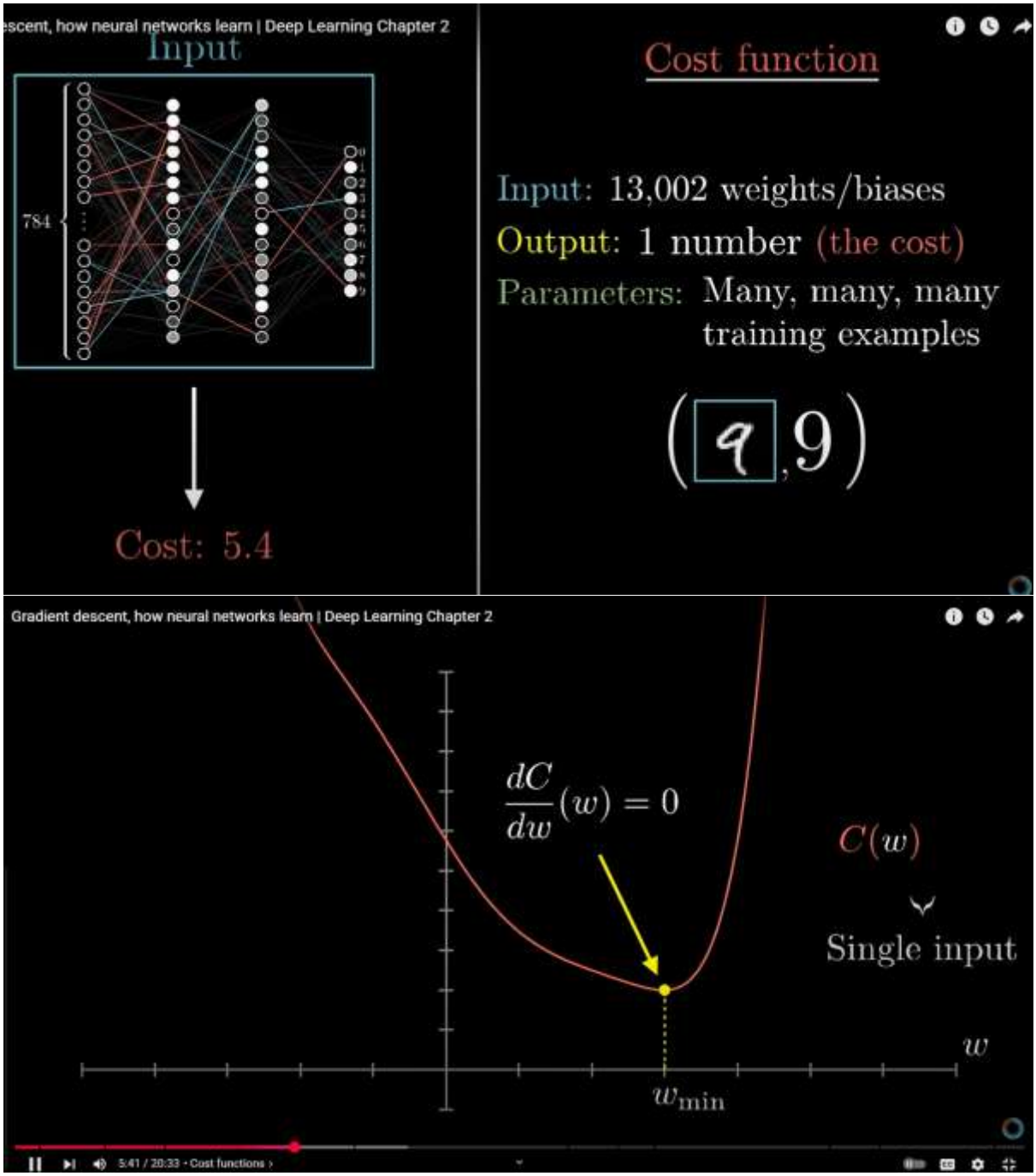


this sum is big so bad boy

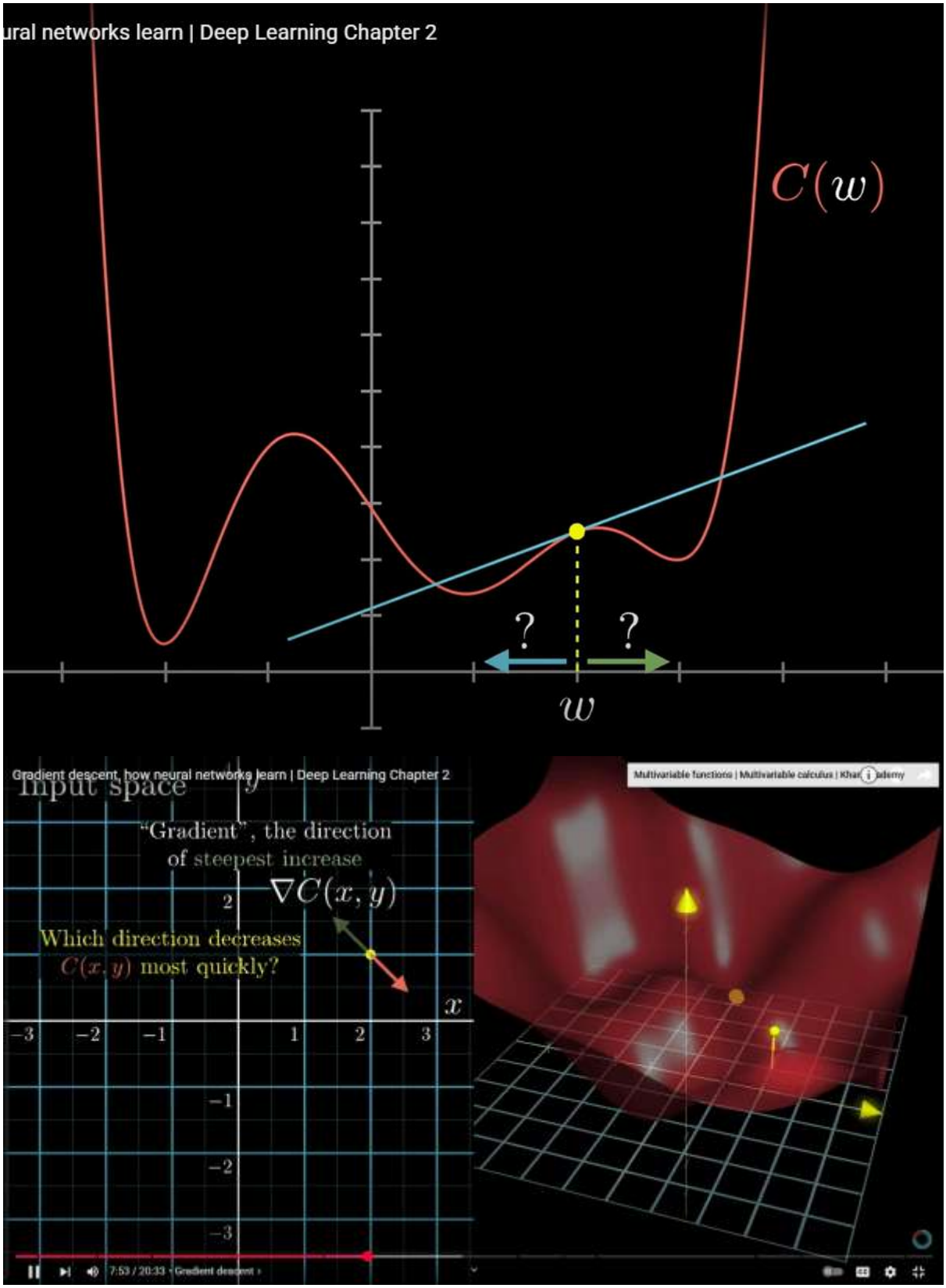


this sum is small so good boy

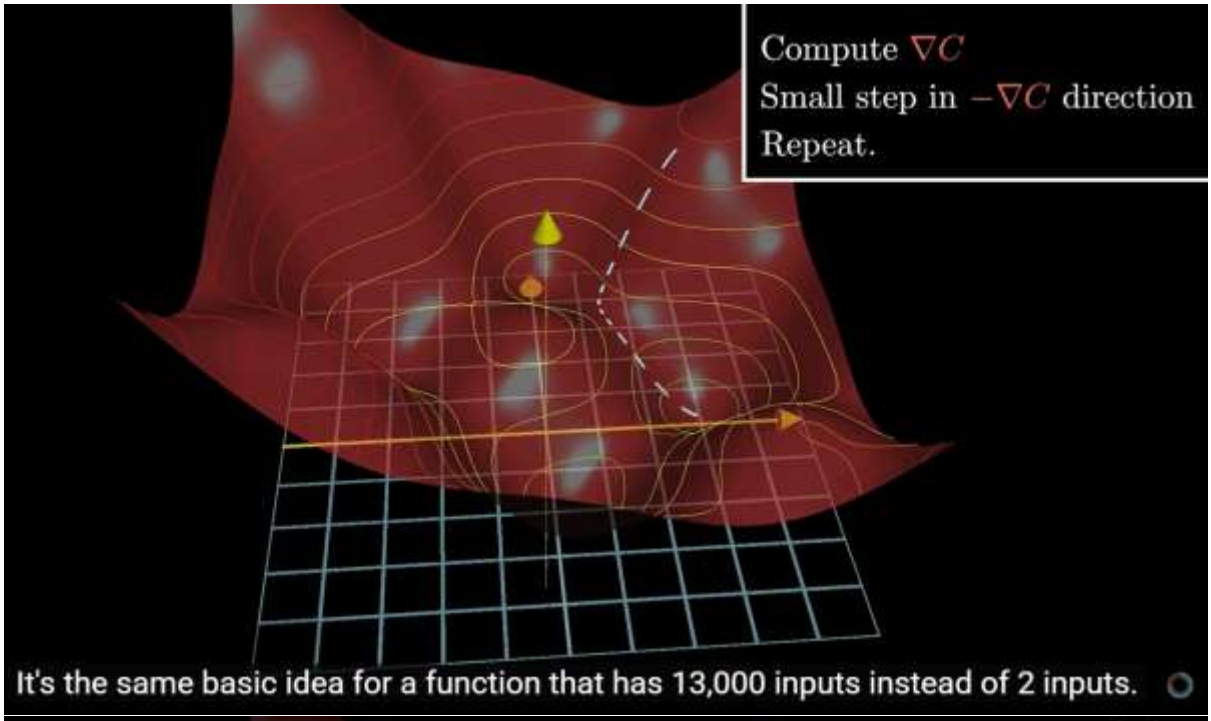
then we find avg cost of all training data

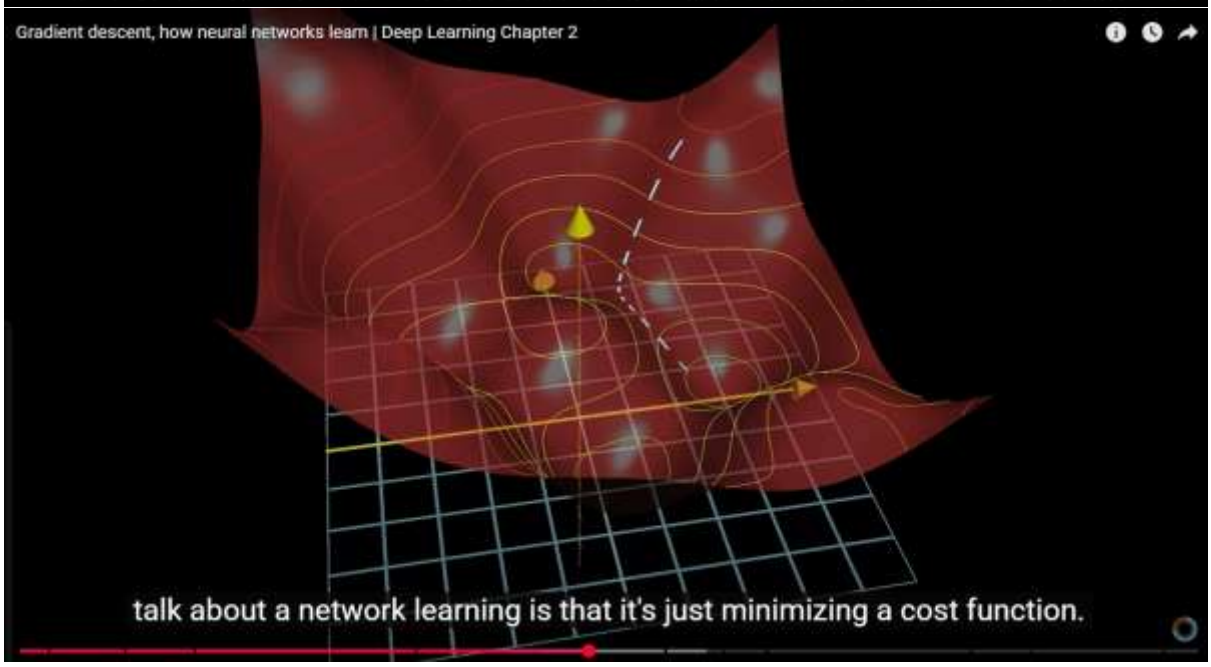


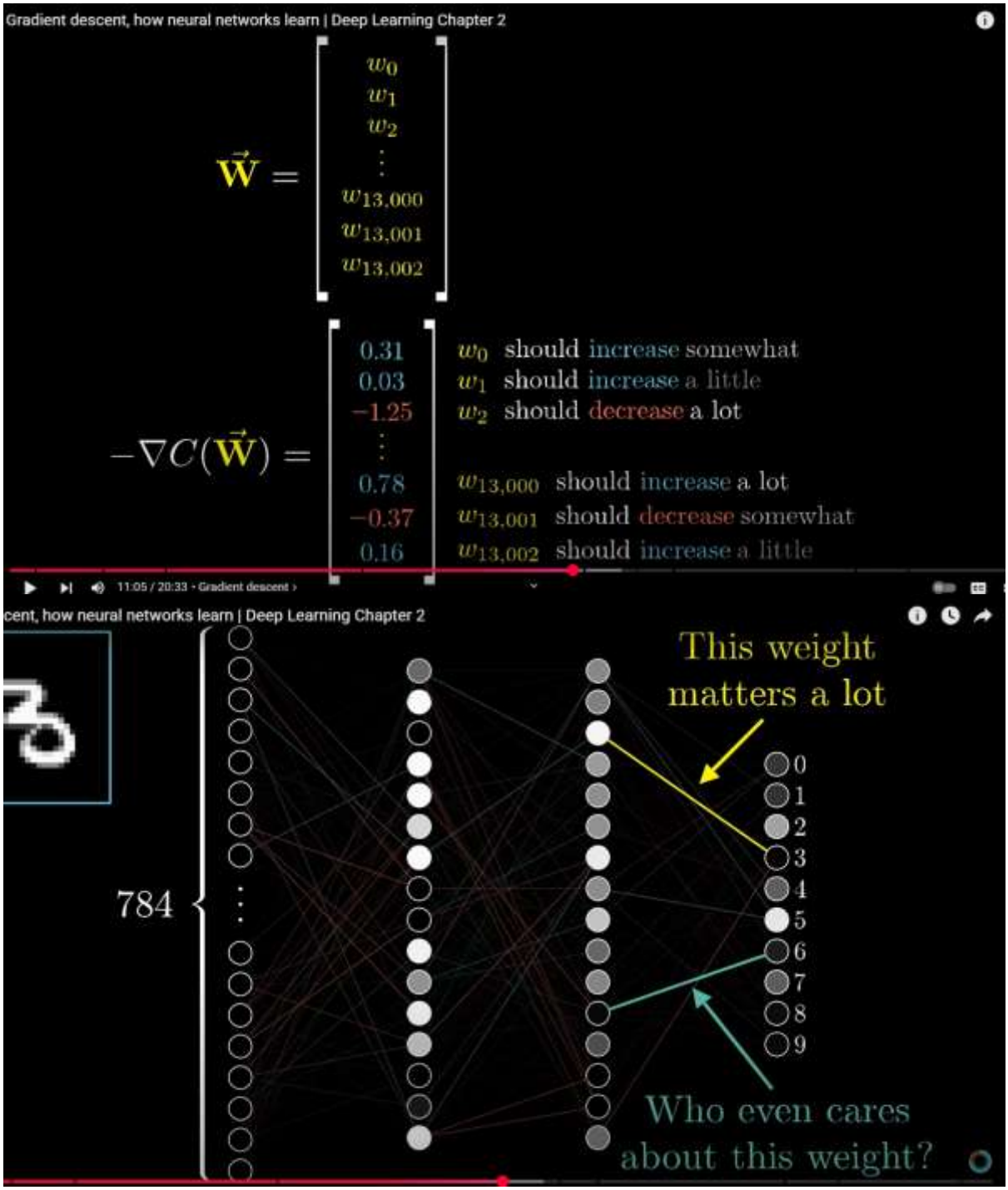
but this is not feasible for very big values

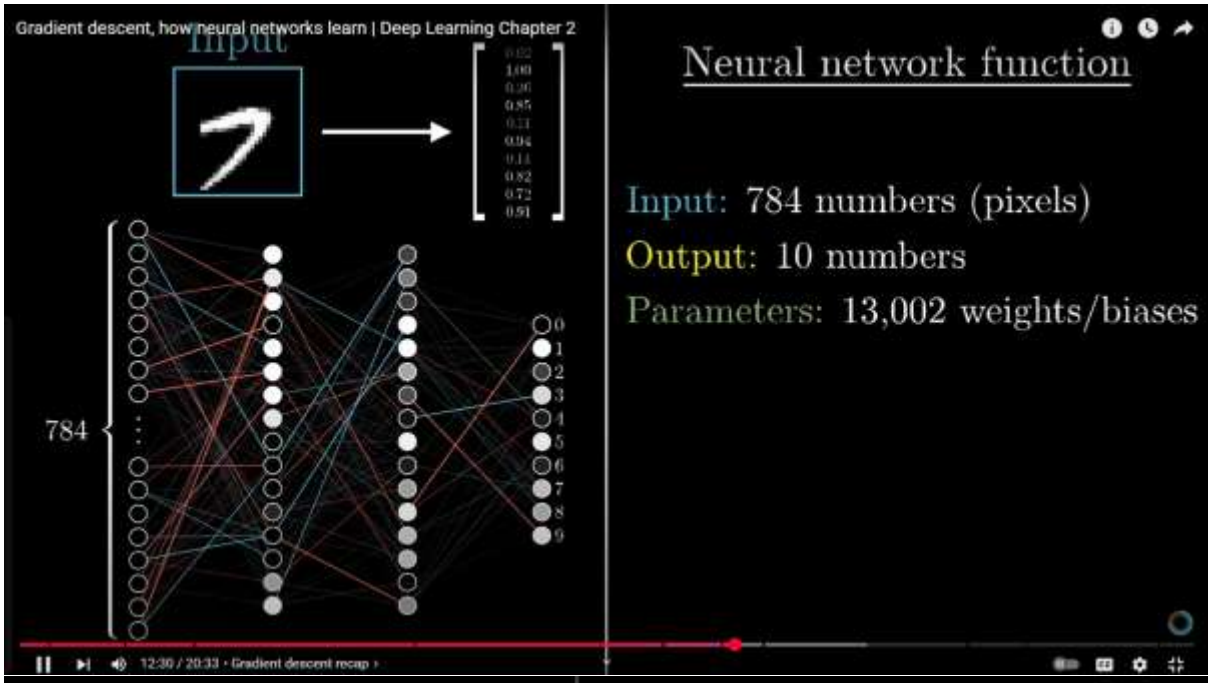


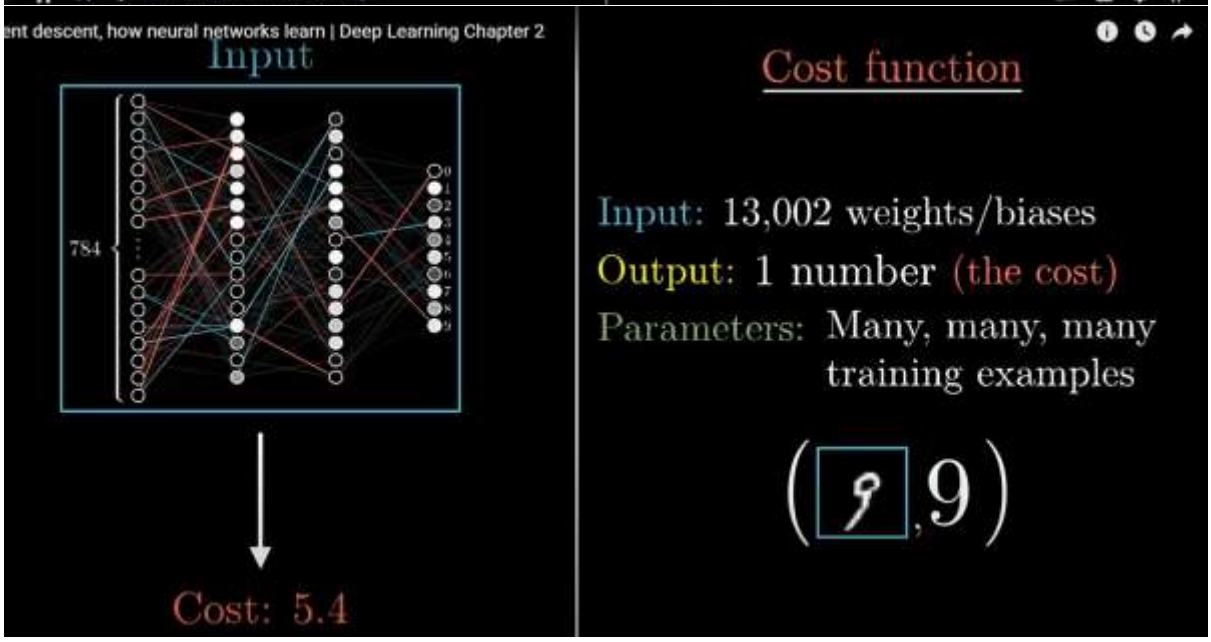
find vector that tells the downhill direction and how steep that is

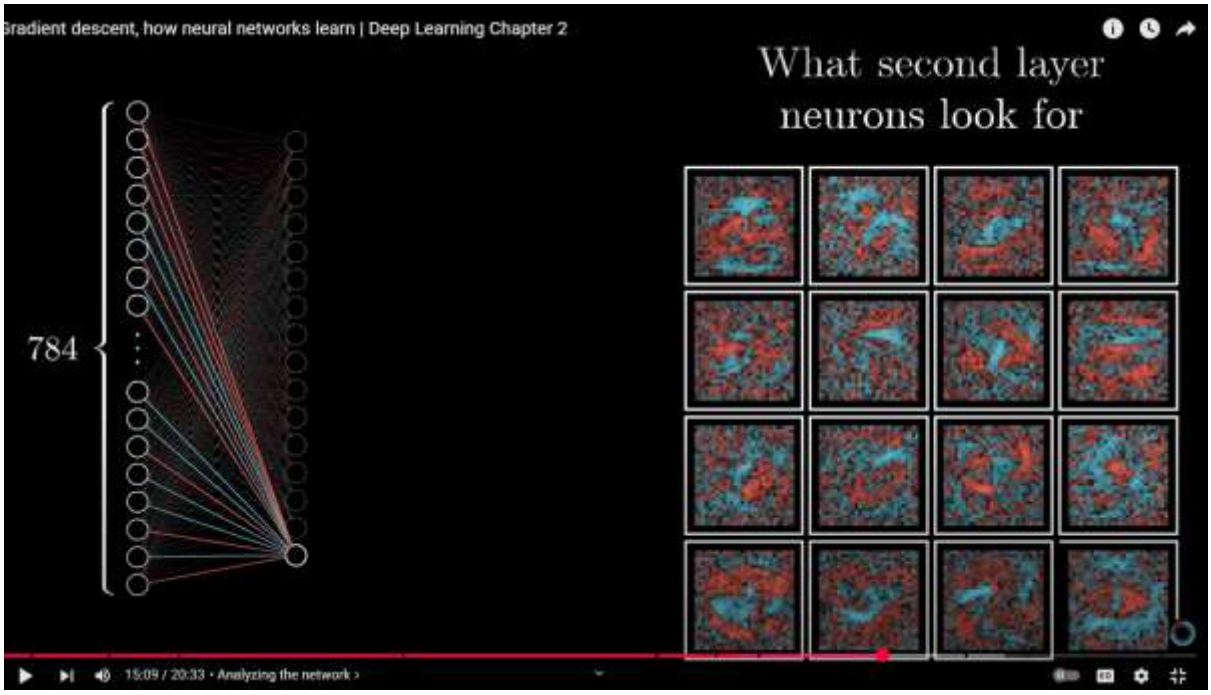

Compute ∇C
Small step in $-\nabla C$ direction
Repeat.
It's the same basic idea for a function that has 13,000 inputs instead of 2 inputs.


Gradient descent, how neural networks learn | Deep Learning Chapter 2
talk about a network learning is that it's just minimizing a cost function.


Gradient descent, how neural networks learn | Deep Learning Chapter 2
$\vec{\mathbf{W}} =$
w_0
w_1
w_2
$w_{13,000}$
$w_{13,001}$
$w_{13,002}$
$-\nabla C(\vec{\mathbf{W}}) =$
0.31
0.03
−1.25
0.78
−0.37
0.16
w_0 should increase somewhat
w_1 should increase a little
w_2 should decrease a lot
$w_{13,000}$ should increase a lot
$w_{13,001}$ should decrease somewhat
$w_{13,002}$ should increase a little
11:05 / 20:33 • Gradient descent ›
cent, how neural networks learn | Deep Learning Chapter 2
784
This weight matters a lot
0
1
2
3
4
5
6
7
8
9
Who even cares about this weight?

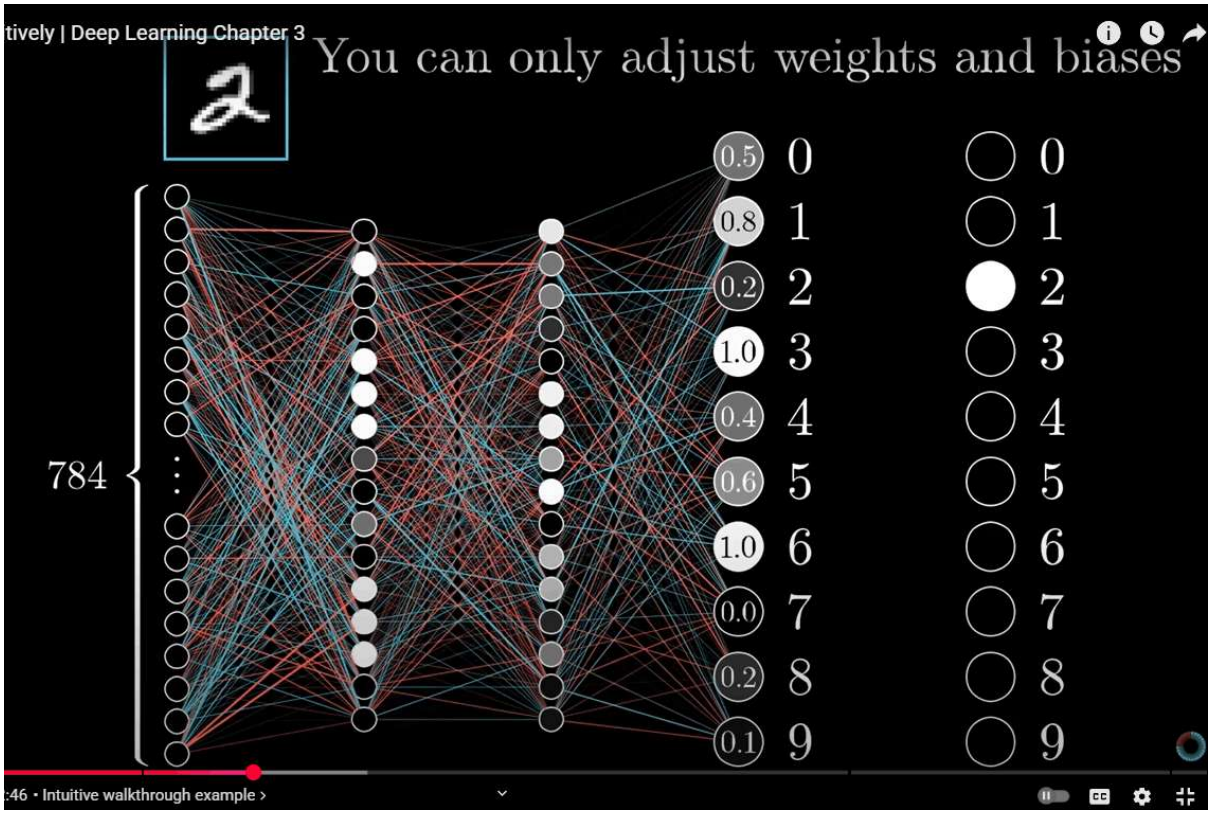

Gradient descent, how neural networks learn | Deep Learning Chapter 2
Input
784
0
1
2
3
4
5
6
7
8
9
12:30 / 20:33 • Gradient descent recap ›
Neural network function
Input: 784 numbers (pixels)
Output: 10 numbers
Parameters: 13,002 weights/biases

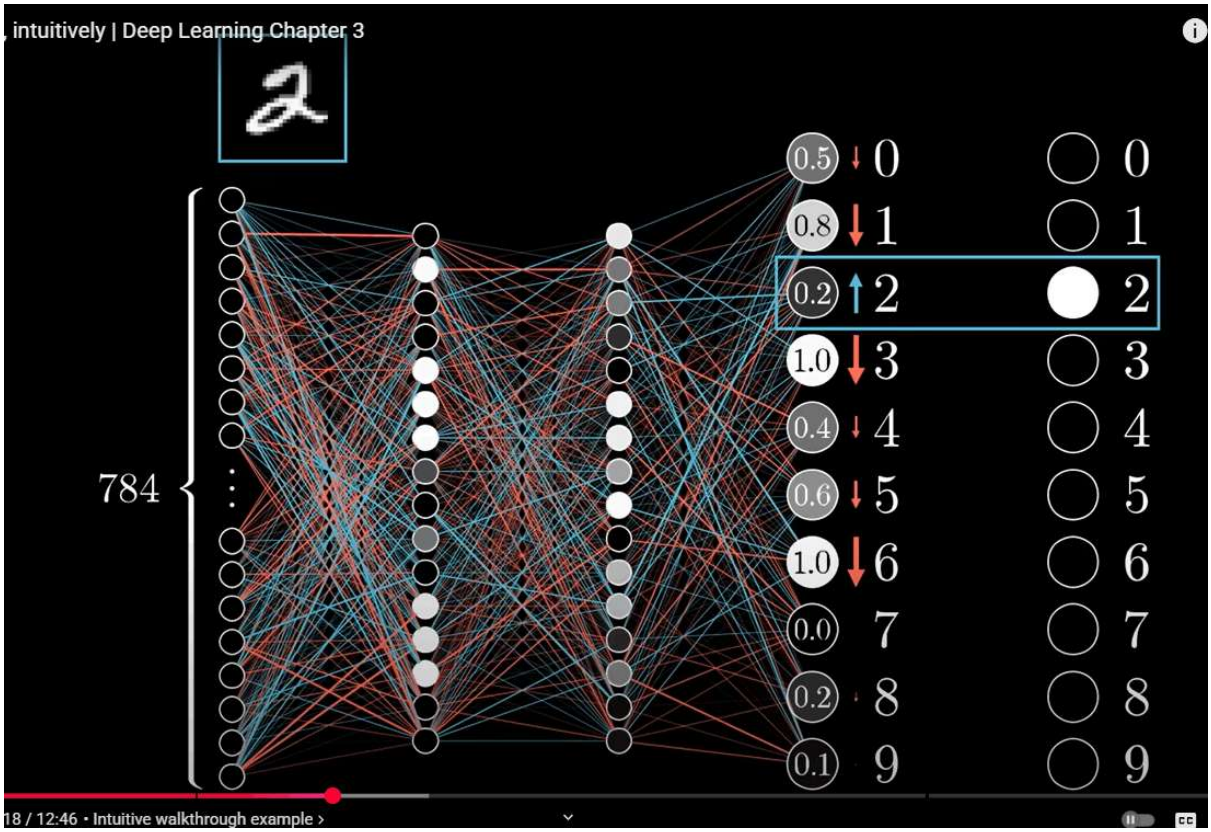

ent descent, how neural networks learn | Deep Learning Chapter 2
Input
784
Cost: 5.4
Cost function
Input: 13,002 weights/biases
Output: 1 number (the cost)
Parameters: Many, many, many training examples
(, 9)



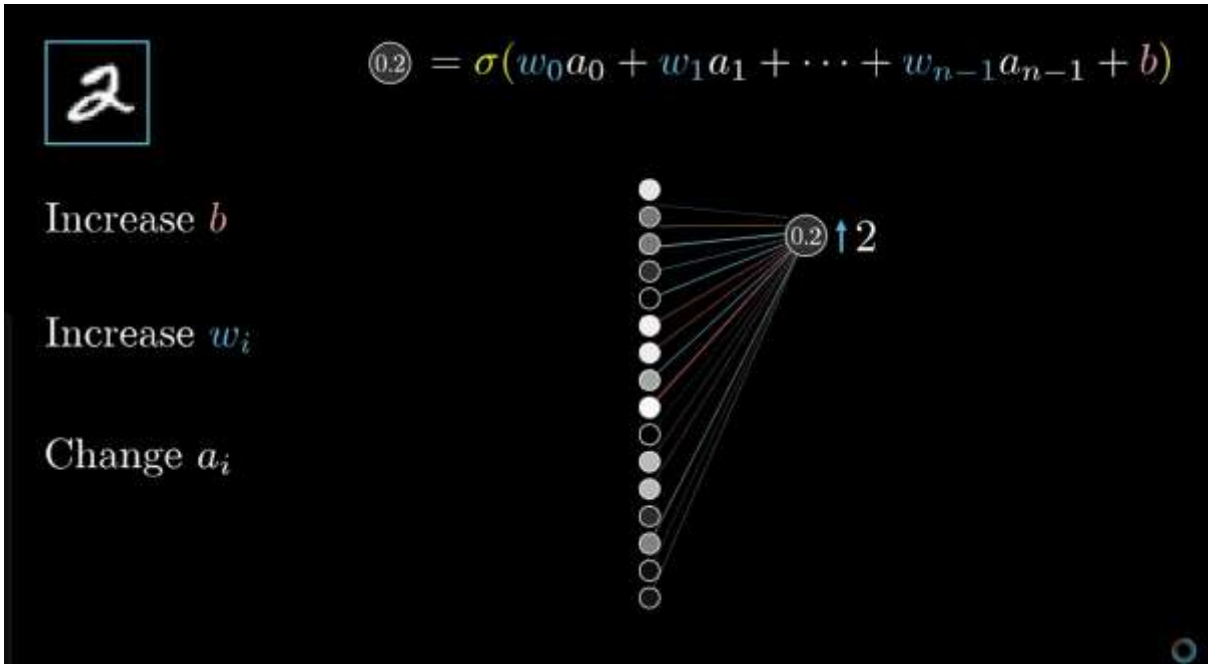
initally

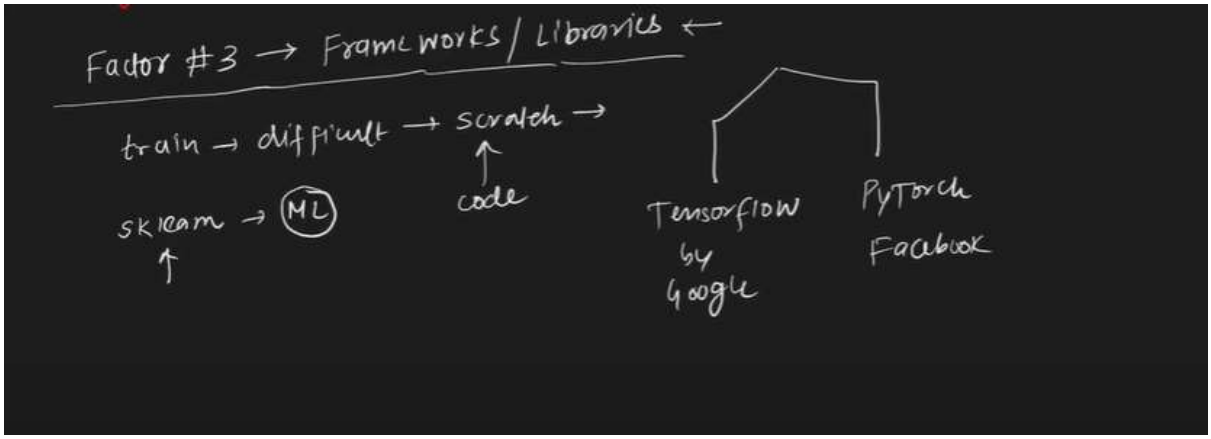
bad network





the sizes of UP and DOWN should be proponiate





tensorflow was very tough so they launches tensorflow + keras

all these are famous neural networks pretrained by researches that u can download and use

- type 1 - MLP - regression / classification problems
- type 2 - CNN - image processing
- type 3 - RNN / LSTM - NLP application
- type 4 - auto encoders - compression without losing quality
- type 5 - GAN - image / music generation

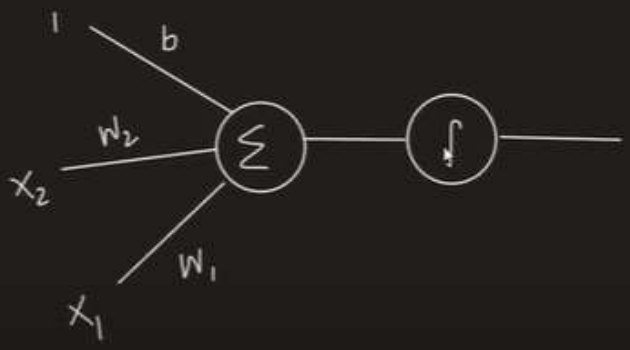
Type of Neural Network	Key Characteristics	Applications
Convolutional Neural Network (CNN)	Designed for grid-like data like images; uses convolutional layers.	Used in Image Recognition and medical image analysis.
Recurrent Neural Network (RNN)	Suitable for sequential data such as text and speech; has internal memory.	Applied in Speech recognition and language translation.
Long Short-Term Memory (LSTM)	A type of RNN for long-term dependencies; uses memory cells.	Utilized for language translation and sentiment analysis.
Transformer Networks	Processes sequences using a self-attention mechanism.	Commonly used for Natural Language Processing (NLP) and machine translation.
Generative Adversarial Networks (GANs)	Involves competing generator and discriminator networks.	Used for image generation and data augmentation.
Artificial Neural Networks (ANNs)	A foundational type with input, hidden, and output layers.	Applied in Image classification and predictive analytics.
Multilayer Perceptrons (MLPs)	Feedforward ANNs with multiple layers.	Used in Facial recognition and classification analysis.

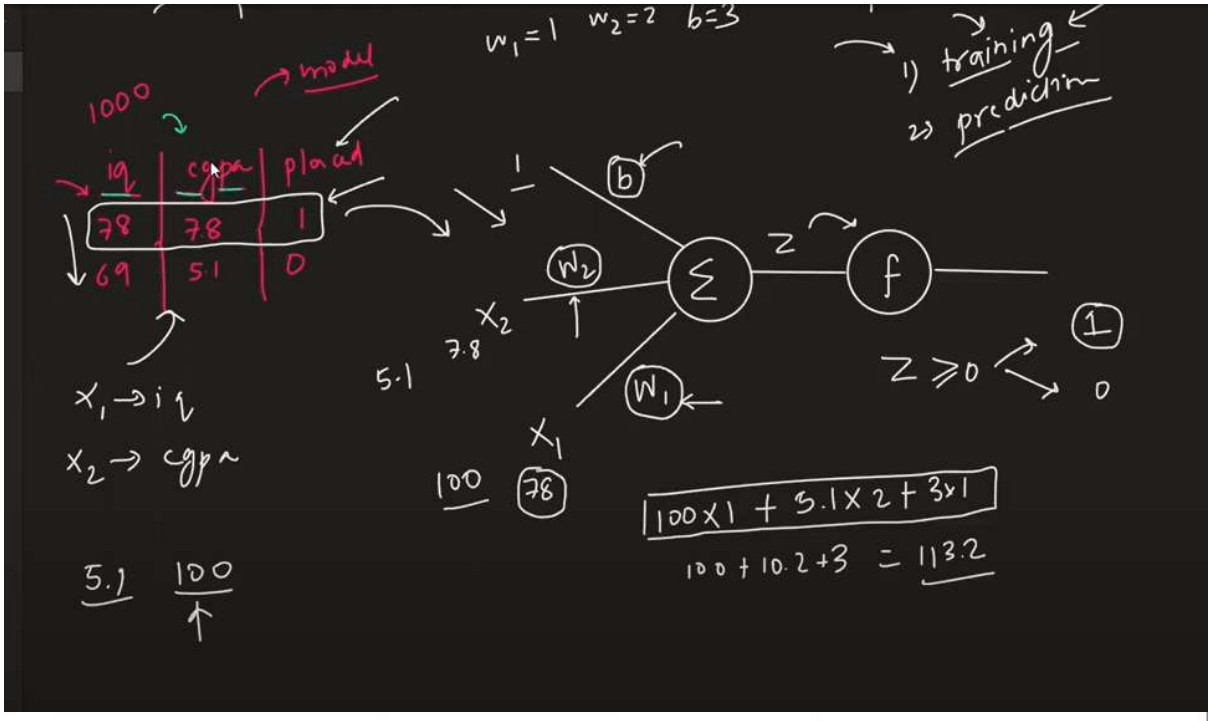
What is a Perceptron?

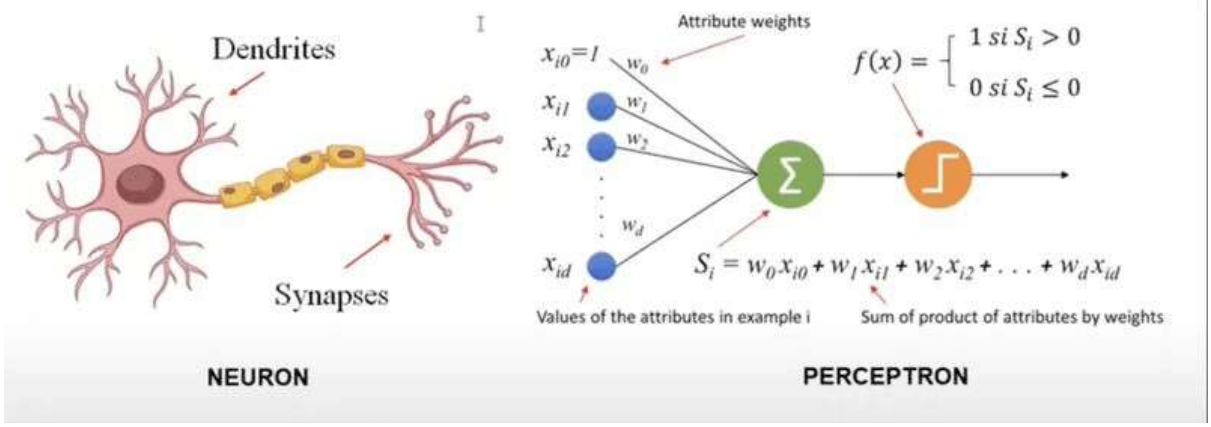
Tuesday, January 4, 2022 6:45 AM

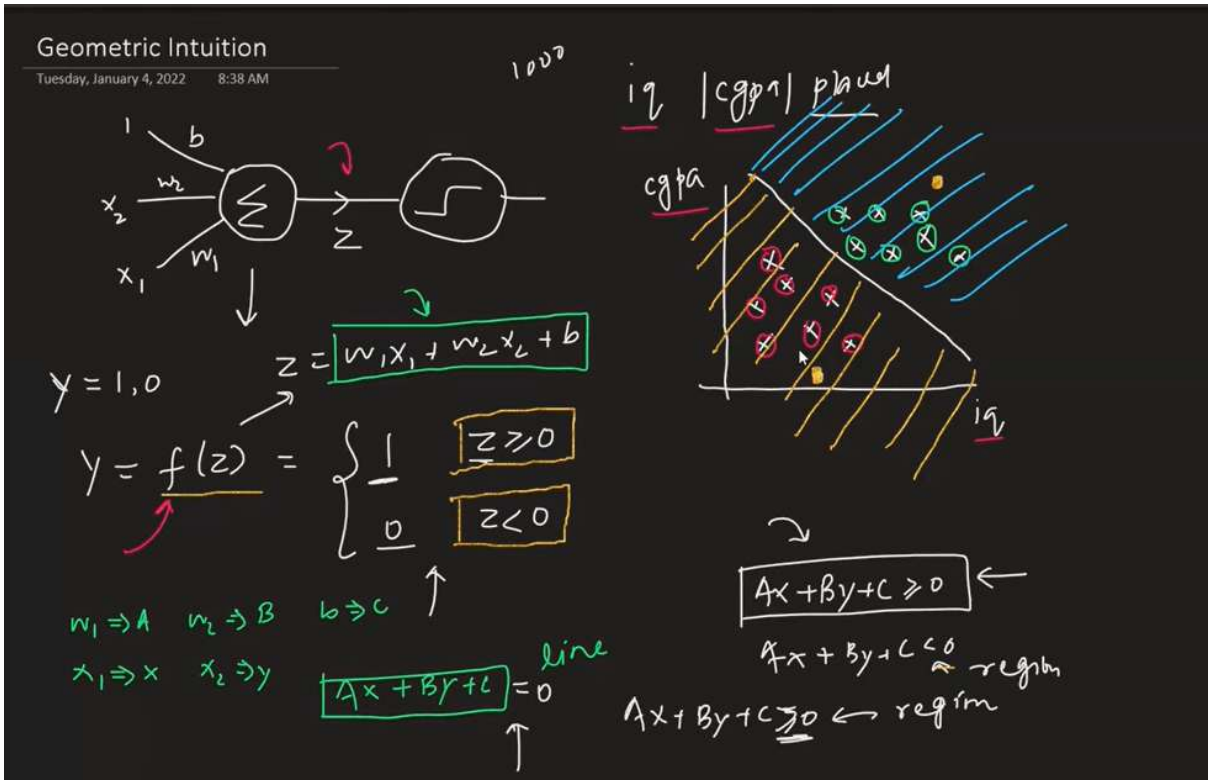
Algorithm → supervised ml
↳ design

- mathematical model
- function



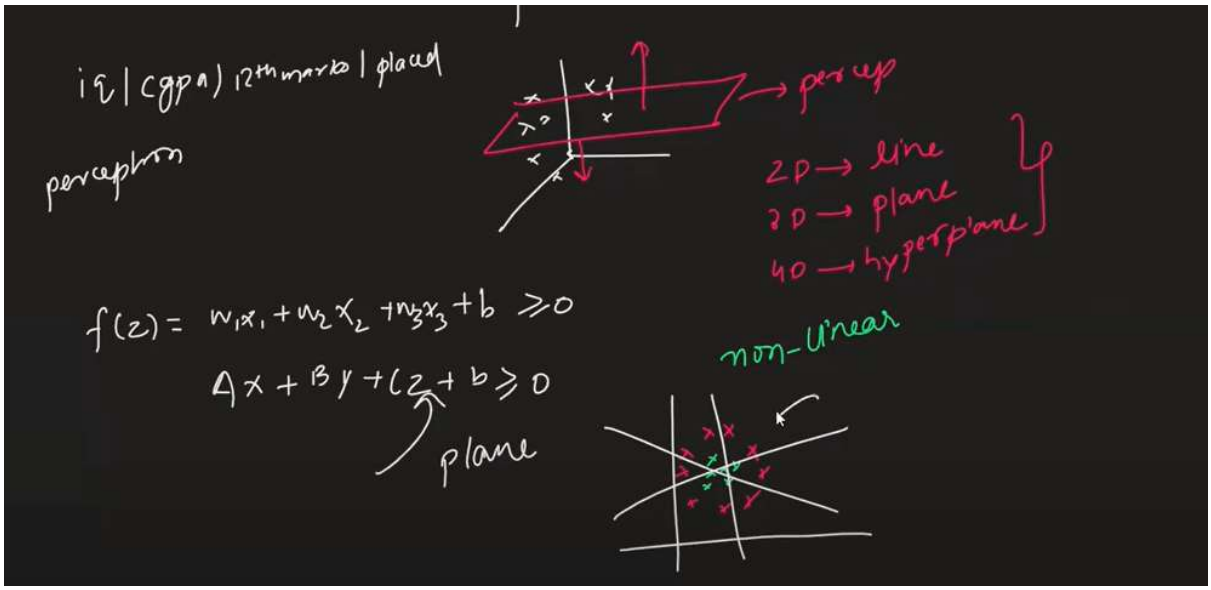

$w_1=1$ $w_2=2$ $b=3$
1) training
2) prediction
1000
model
iq | cgpa | placed
78 | 7.8 | 1
69 | 51 | 0
$x_1 \rightarrow$ iq
$x_2 \rightarrow$ cgpa
5.1 100
b
w_2
w_1
Σ
z
f
x_2
7.8
5.1
x_1
100
78
$z \geq 0$
1
0
$100 \times 1 + 5.1 \times 2 + 3 \times 1$
$100 + 10.2 + 3 = 113.2$


Dendrites
Synapses
Attribute weights
$x_{i0}=1$
x_{i1}
x_{i2}
x_{id}
w_0
w_1
w_2
w_d
Σ
$f(x) = \begin{cases} 1 \text{ si } S_i > 0 \\ 0 \text{ si } S_i \leq 0 \end{cases}$
$S_i = w_0 x_{i0} + w_1 x_{i1} + w_2 x_{i2} + \ldots + w_d x_{id}$
Values of the attributes in example i
Sum of product of attributes by weights
NEURON
PERCEPTRON



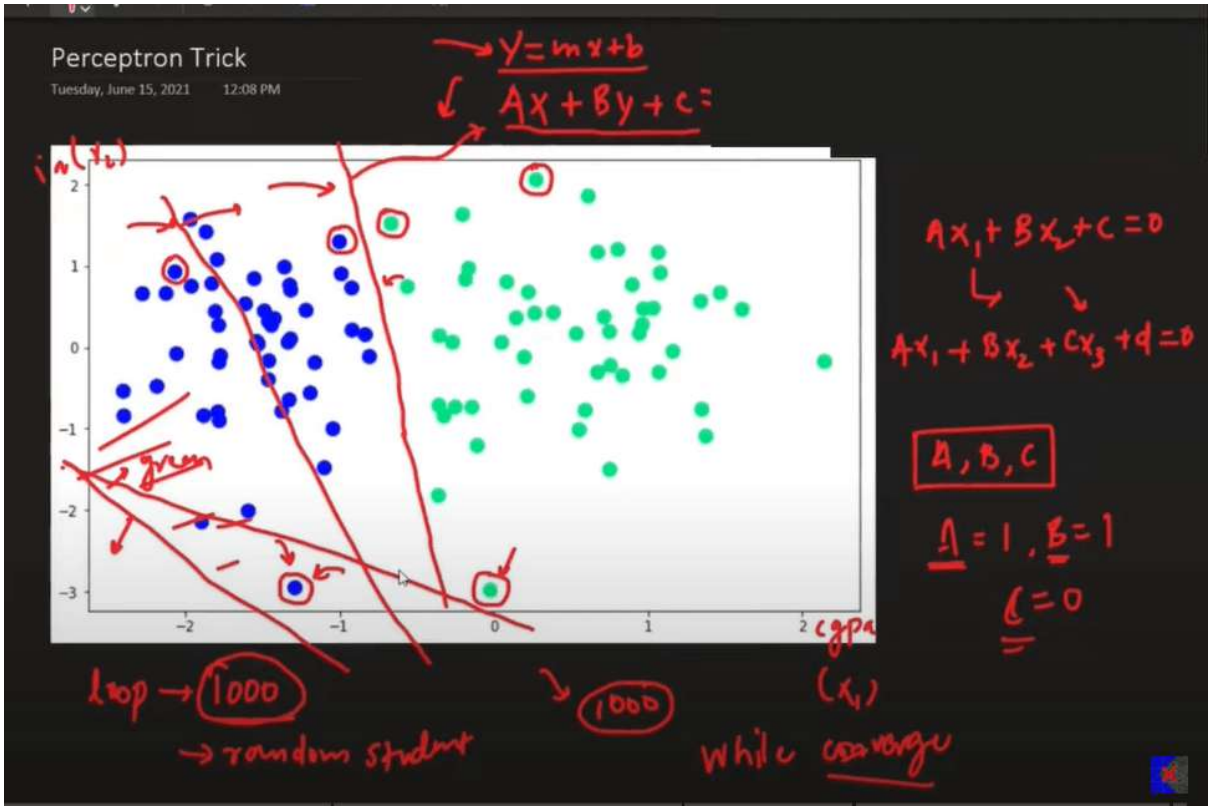
perceptron is a binary classifier

perceptron fails with non linear dataset



scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

rasbt.github.io/mlxtend/user_guide/plotting/plot_decision_regions/

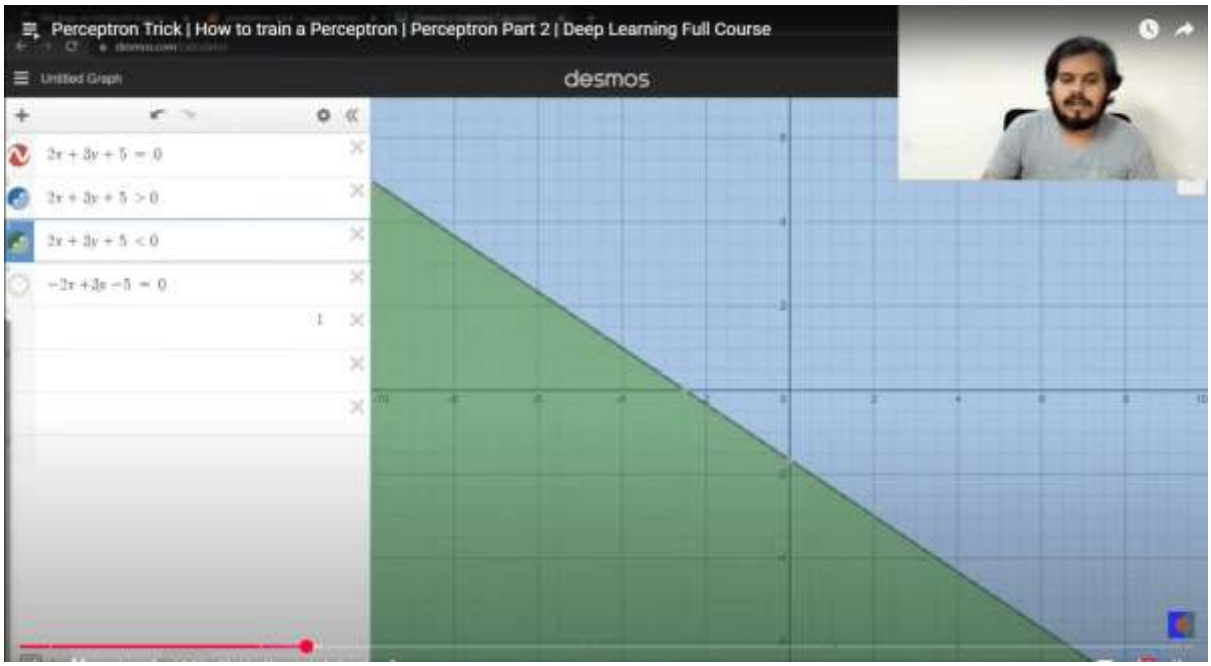


we start with random ABC values

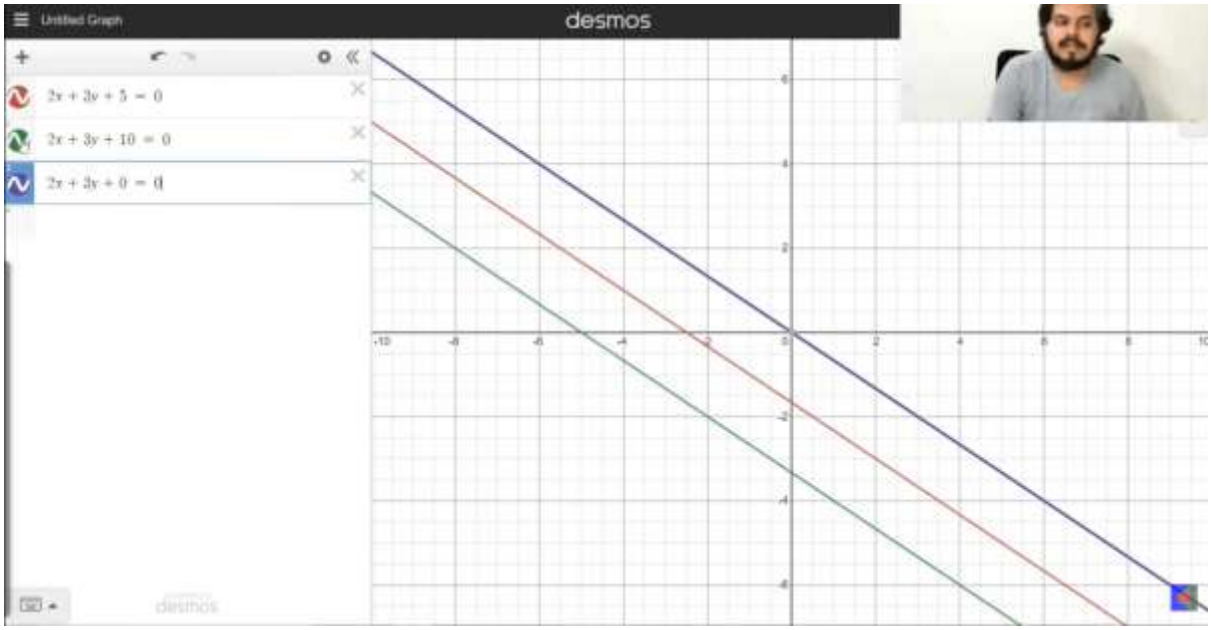
and then based on tha t line we ask random points whether they are guessed right or not

if they are guessed right no changes but if wrong then we shift line such that that selected point falls in correct region

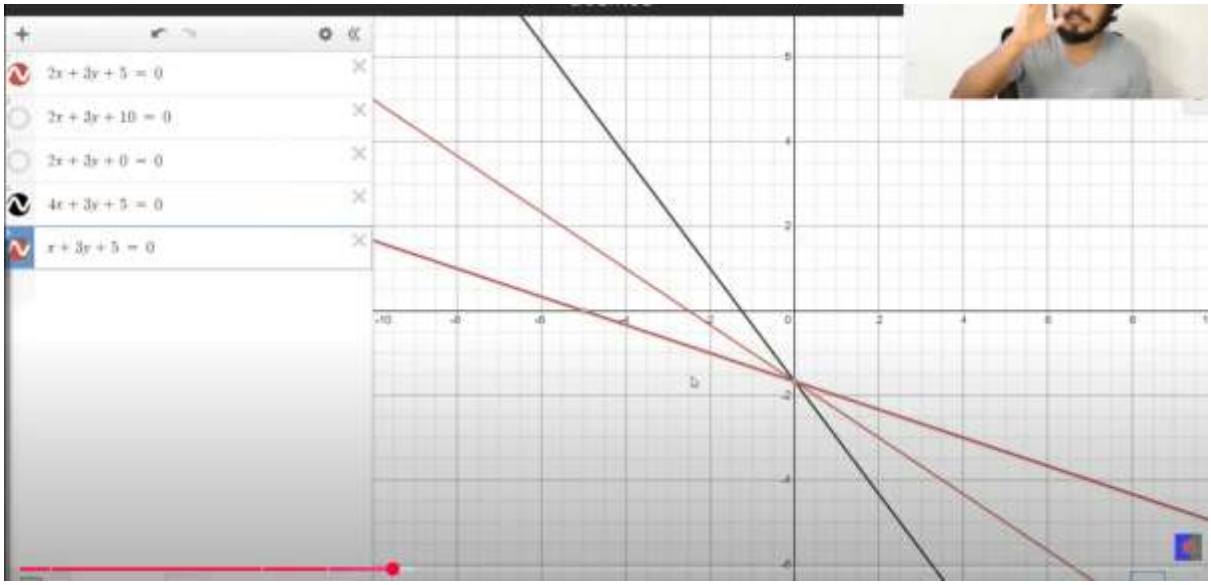
we can do this till convergence



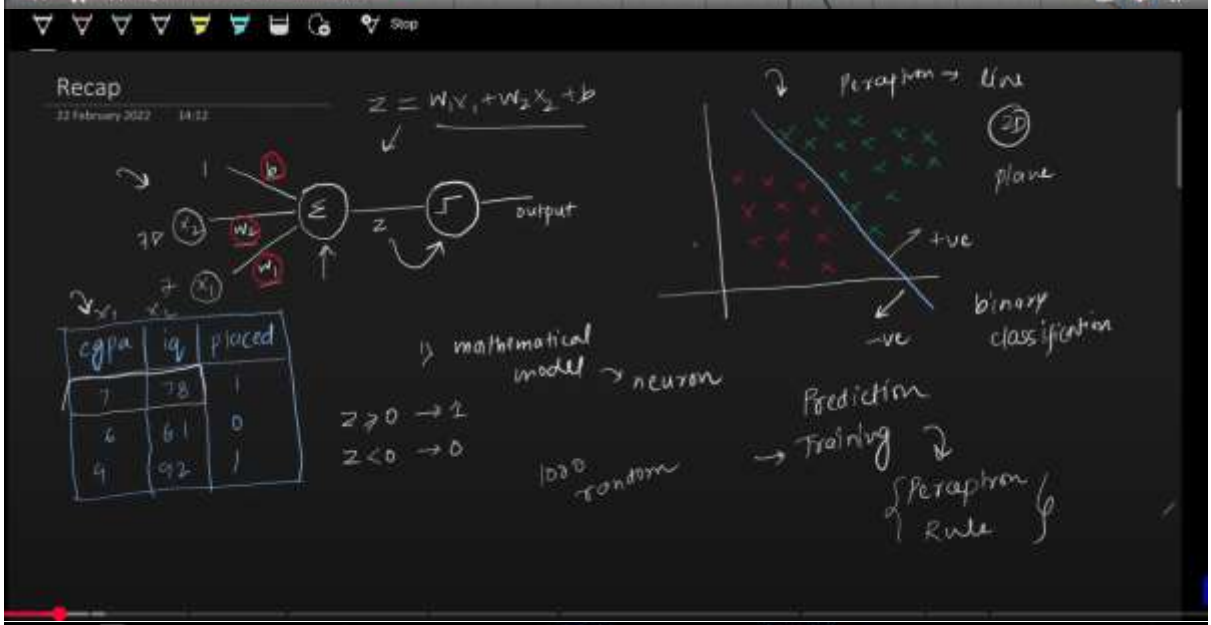
C is moving line up or down

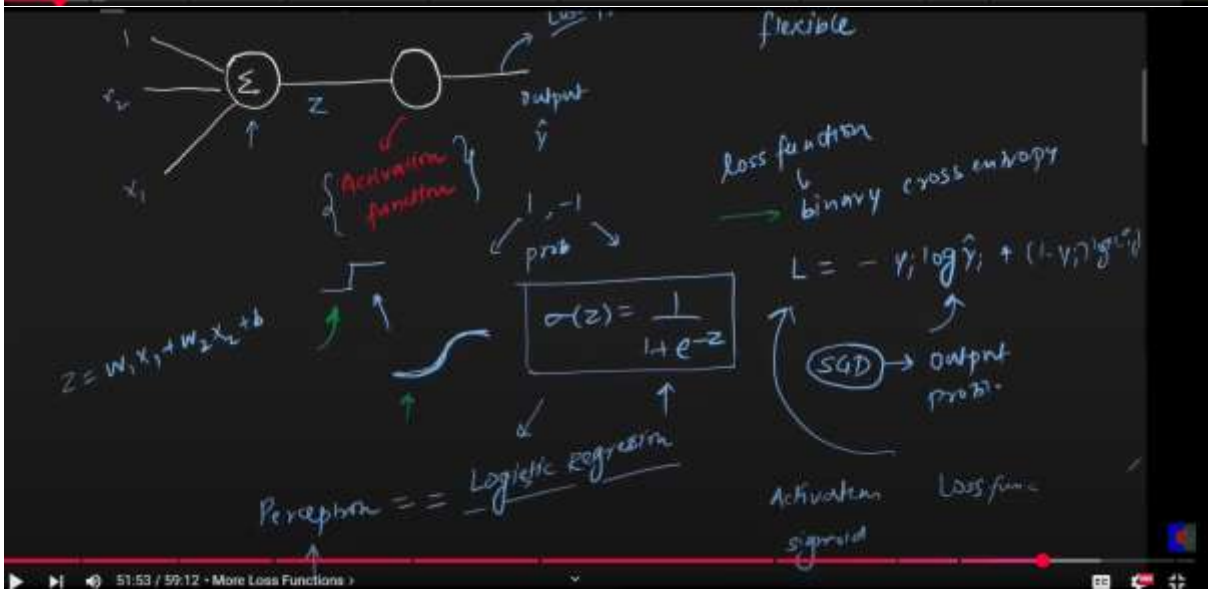


A , B is rotating line



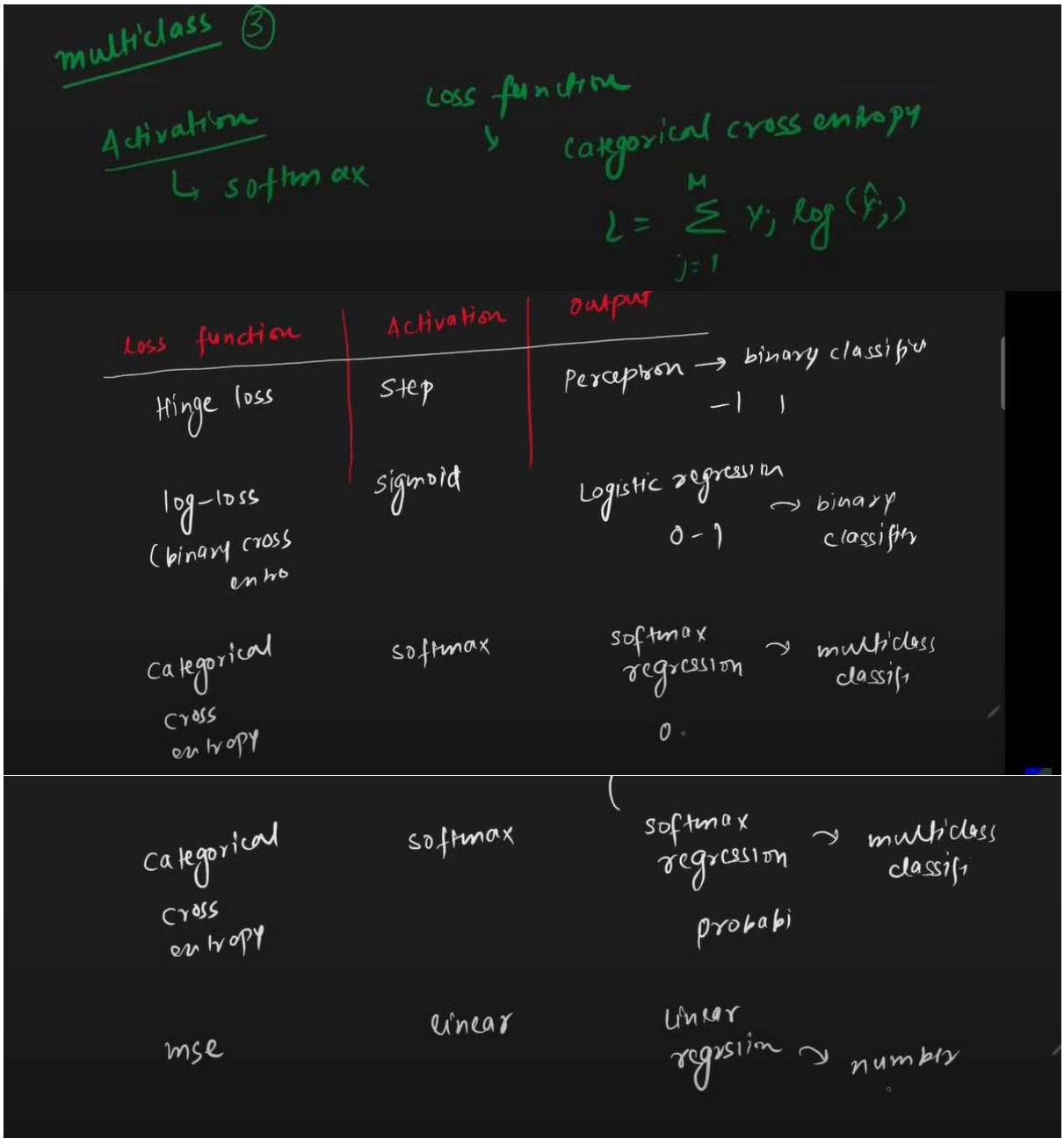

desmos
Untitled Graph
2x + 3y + 5 = 0
2x + 3y + 10 = 0
2x + 3y + 0 = 0
4x + 3y + 5 = 0
x + 3y + 5 = 0
2x + 6y + 5 = 0
2x + y + 5 = 0

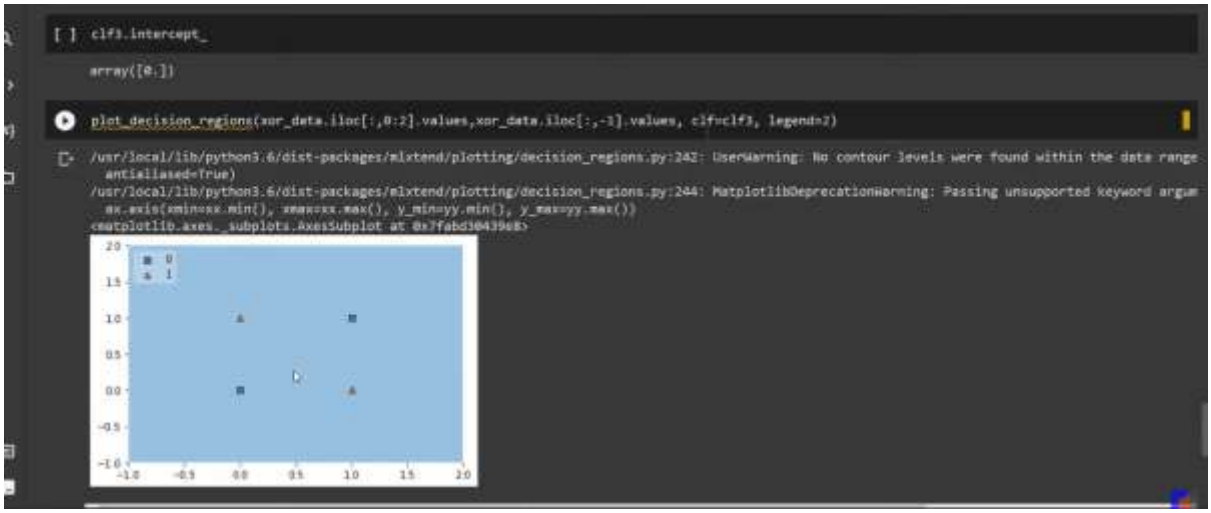

Recap
22 February 2022 14:12
$z = w_1x_1 + w_2x_2 + b$
output
Perceptron → line
2D
plane
+ve
-ve
binary classification
cgpa
iq
placed
mathematical model → neuron
$z \geq 0 \rightarrow 1$
$z < 0 \rightarrow 0$
1000 random
Prediction
Training
Perceptron Rule


flexible
output
Activation function
Loss function
binary cross entropy
$L = -y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)$
prob
$\sigma(z) = \frac{1}{1+e^{-z}}$
$z = w_1x_1 + w_2x_2 + b$
SGD → output prob.
Perceptron == Logistic Regression
Activation
sigmoid
Loss fun
51:53 / 59:12 • More Loss Functions ›

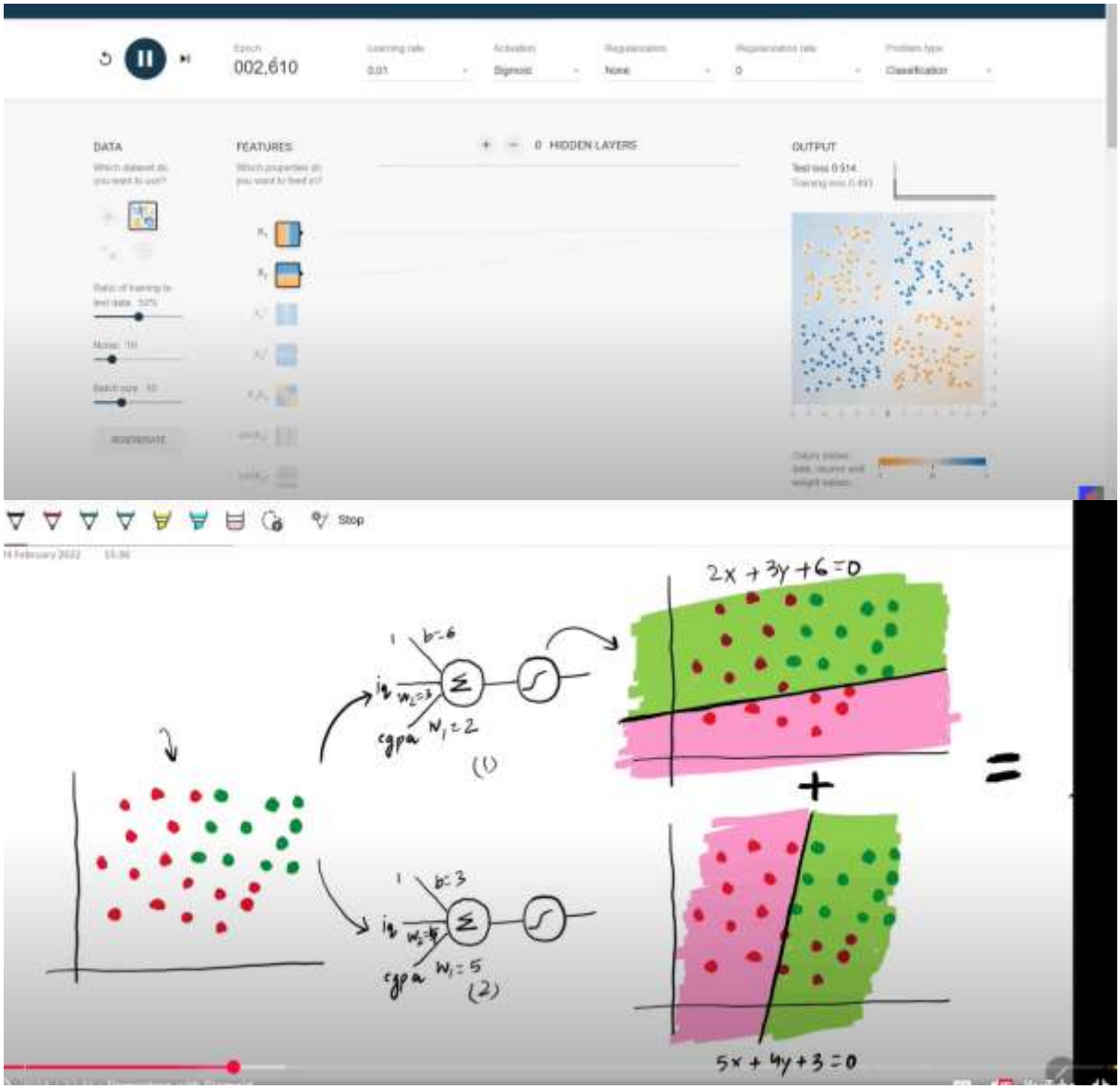
logistic regression has

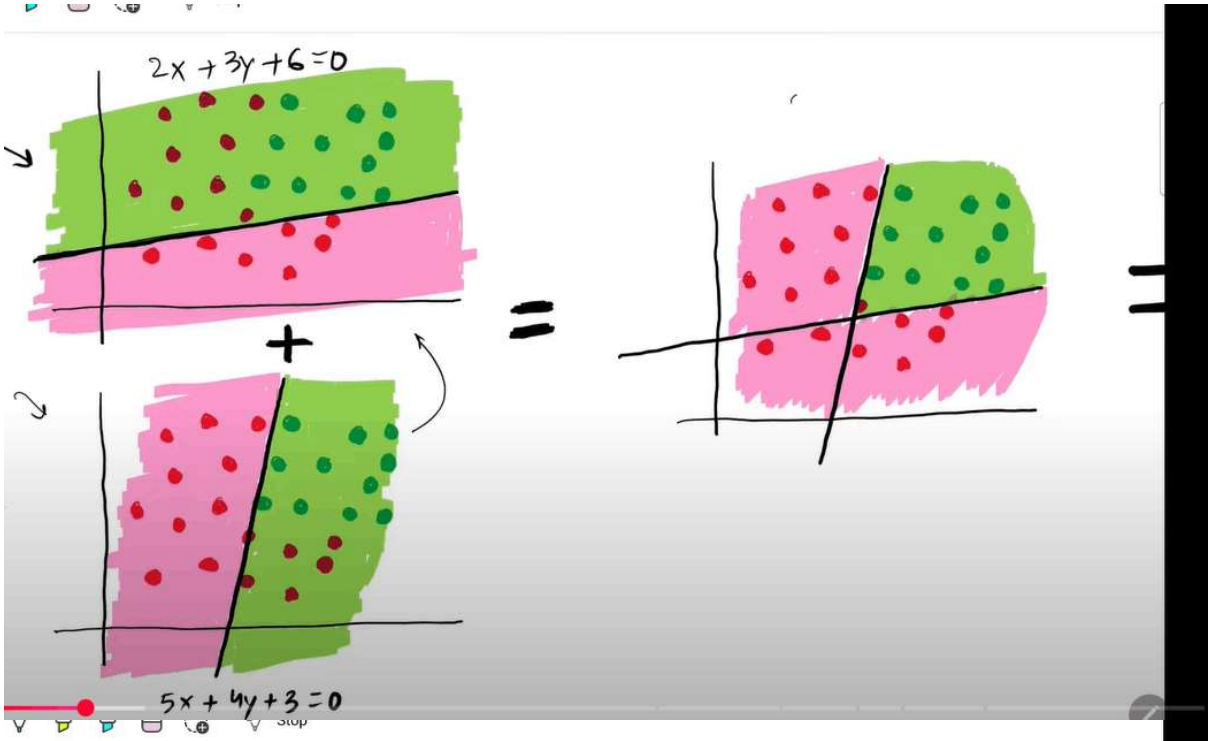
- activation func - sigmoid
- loss function - binary cross entropy

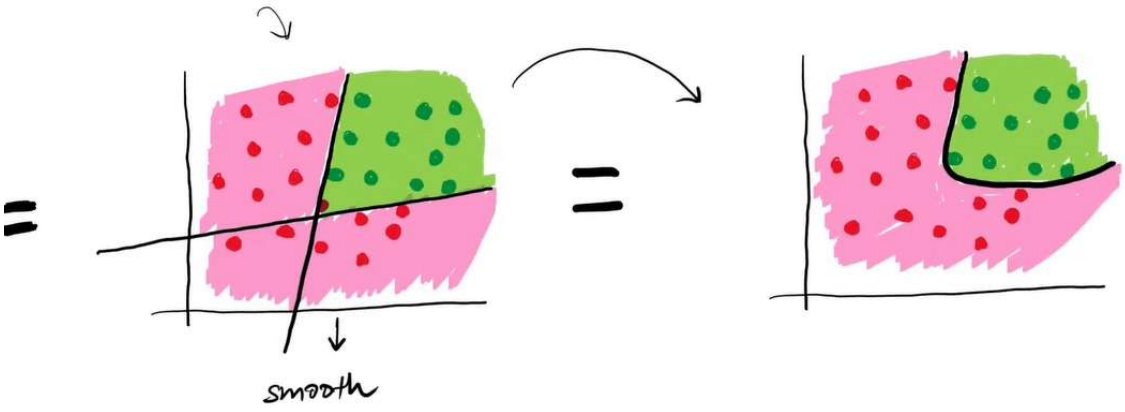


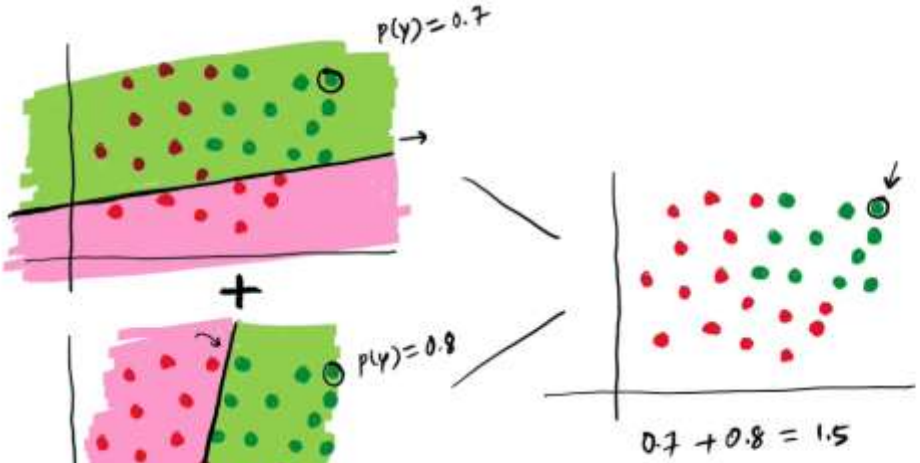


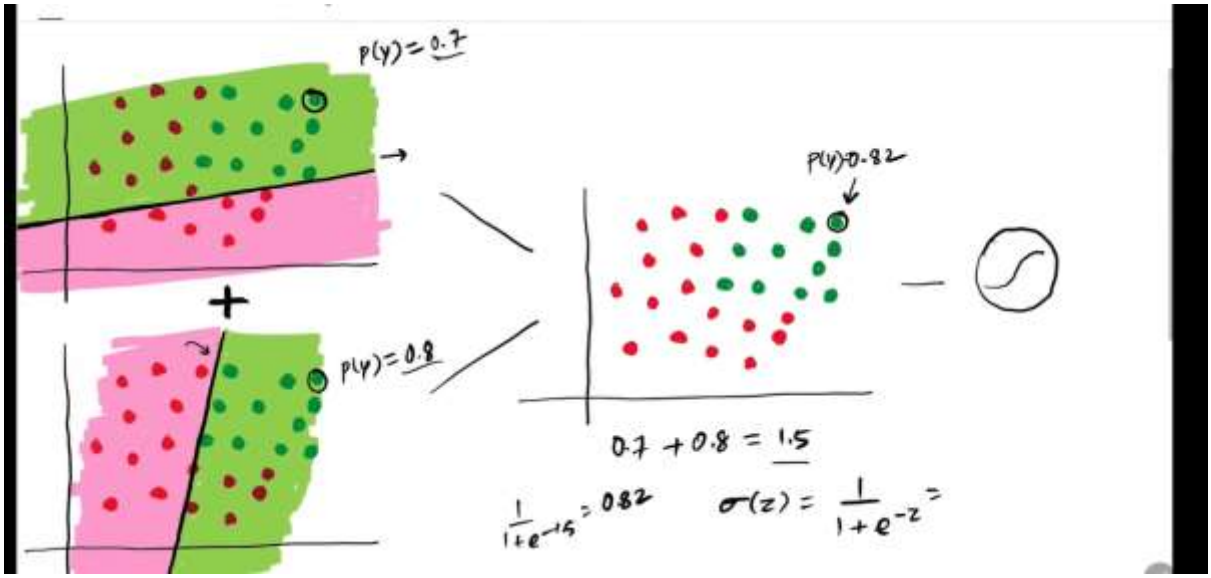
no line possible this is XOR data



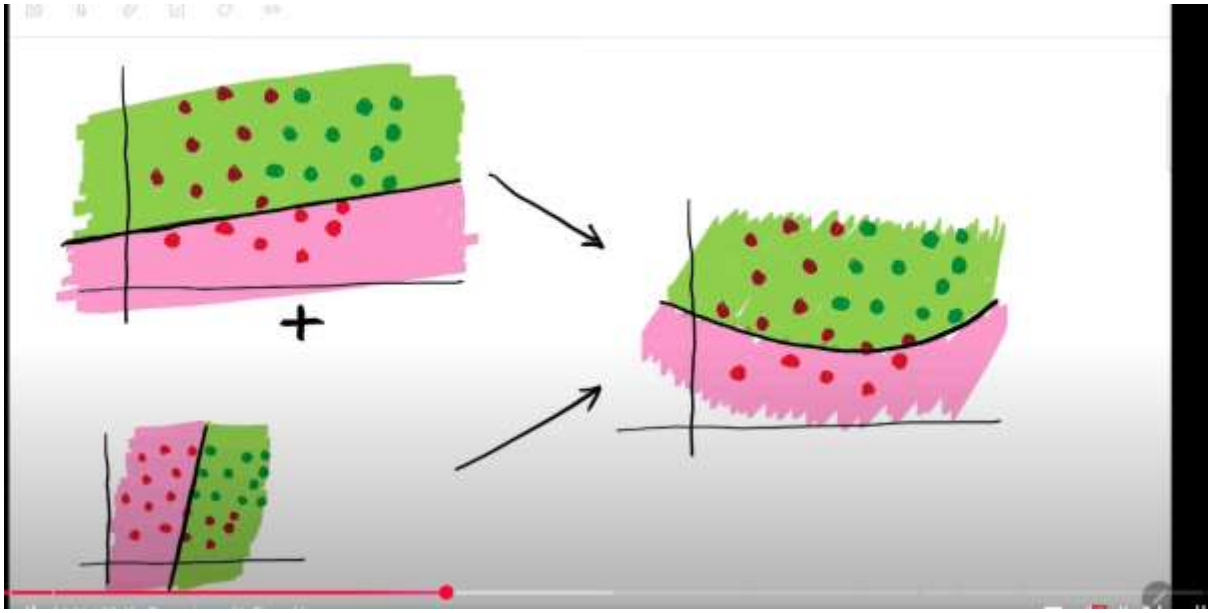

2x + 3y + 6 = 0
5x + 4y + 3 = 0


smooth


P(y) = 0.7
P(y) = 0.8
0.7 + 0.8 = 1.5



nowsuppose i want below type of curve

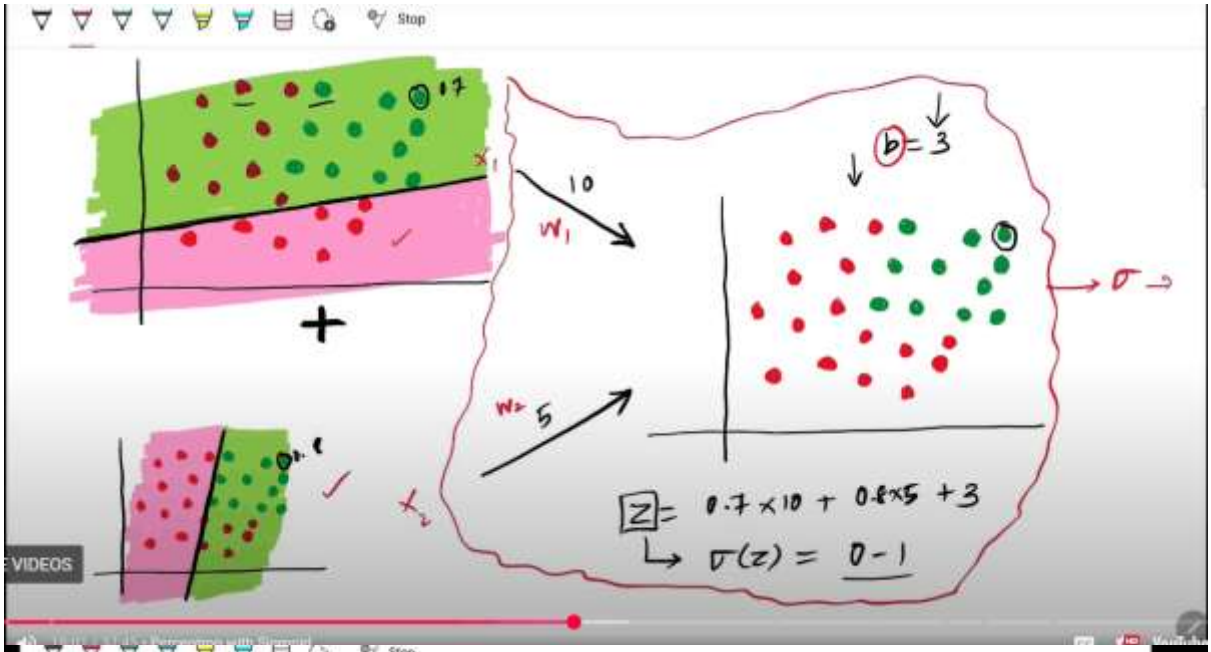


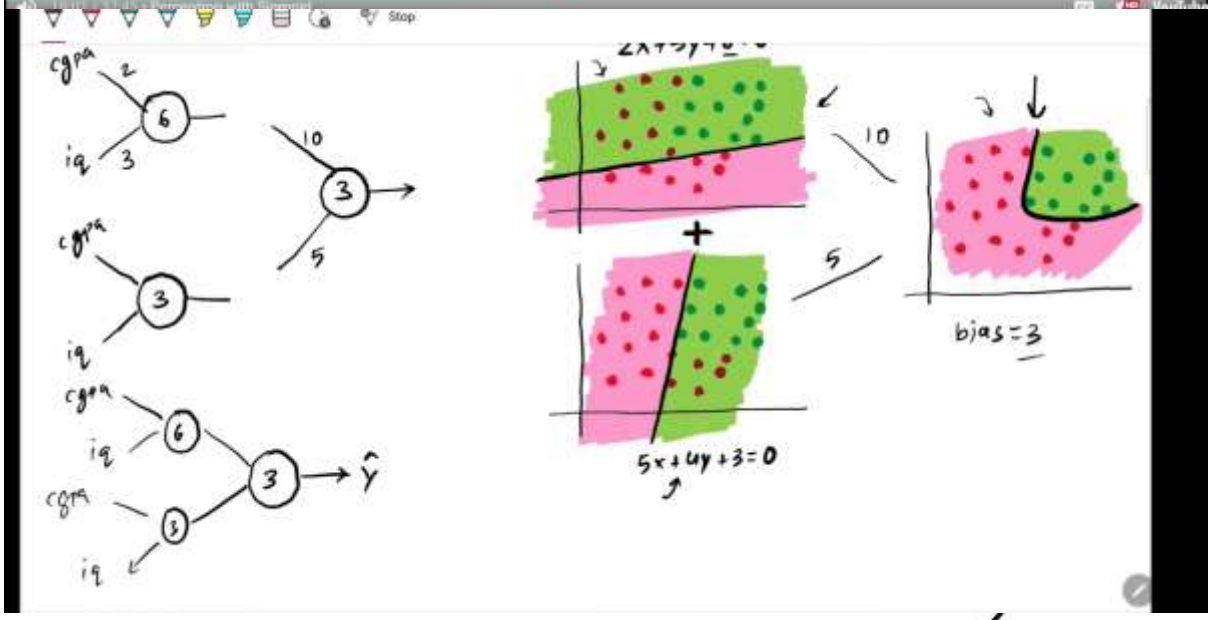
here perceptron 1 has more impact then perceptron 2

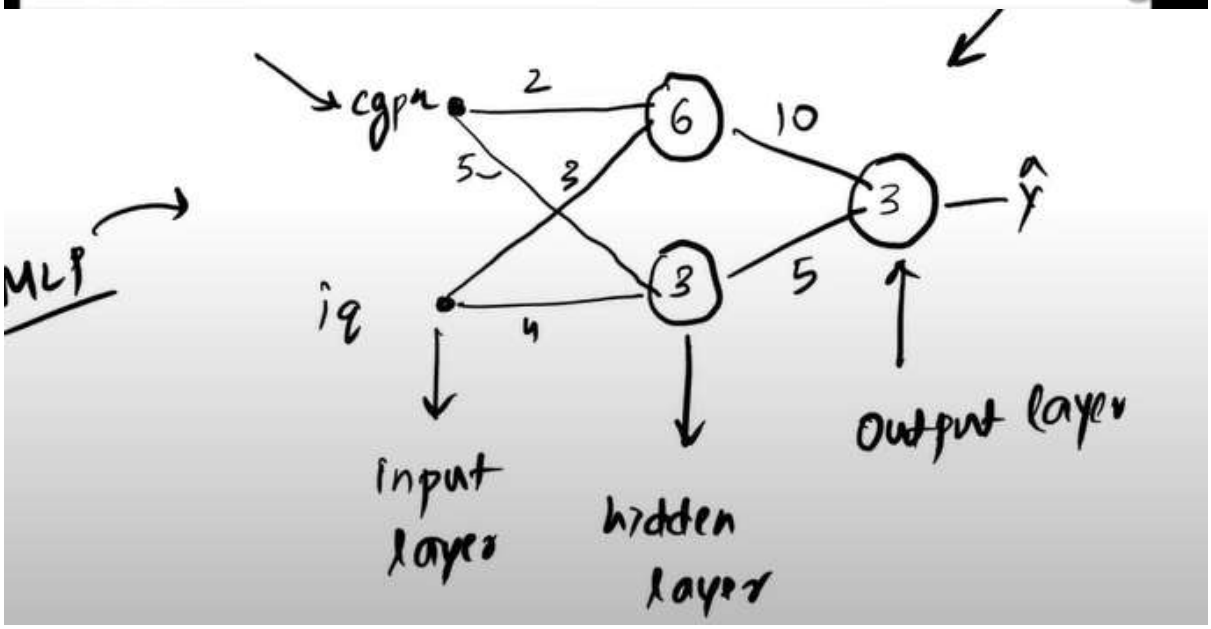
to do so we add weights to perceptron 1 and also bias

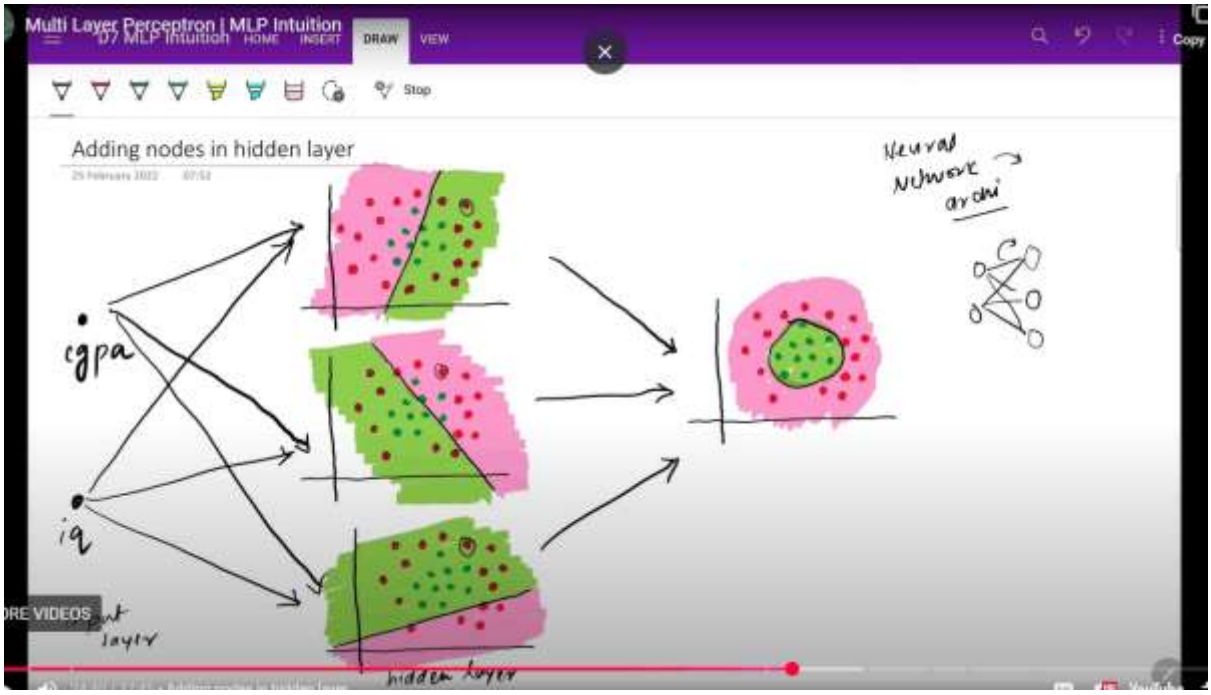
and then pass to sigmoid

so in short the area marked in red below became a new perceptron

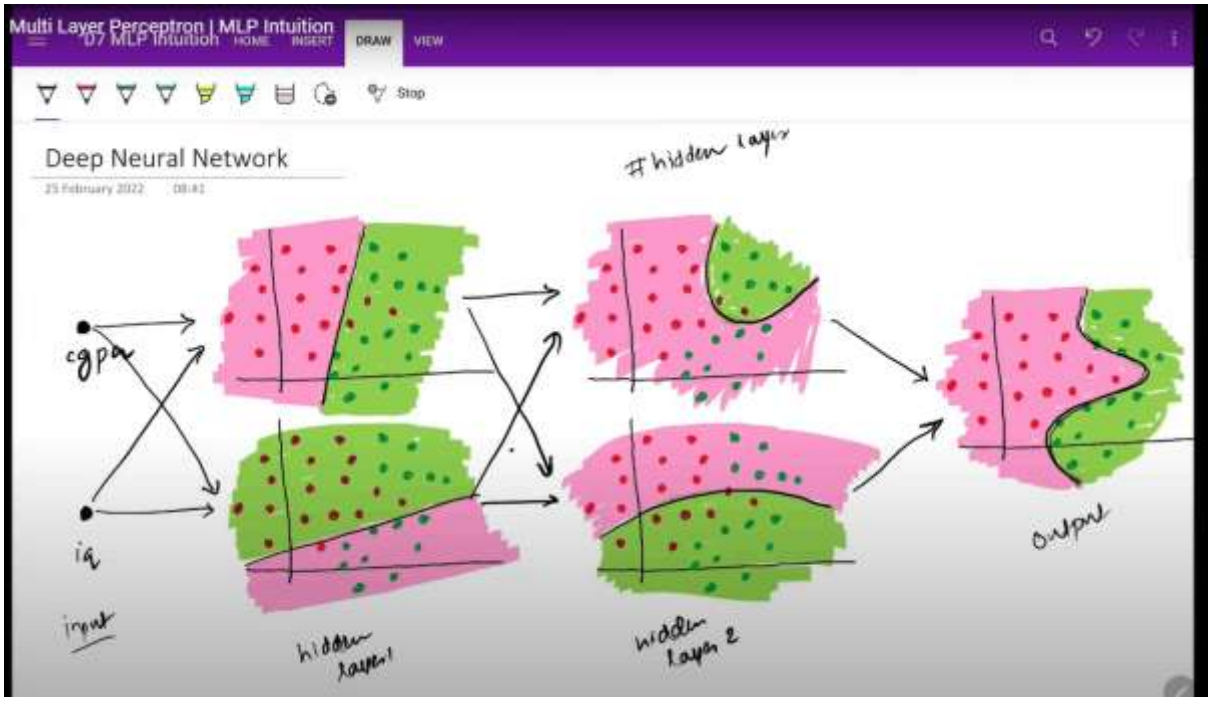

Stop
b = 3
10
W_1
W_2 5
$z = 0.7 \times 10 + 0.8 \times 5 + 3$
$\sigma(z) = 0-1$
VIDEOS

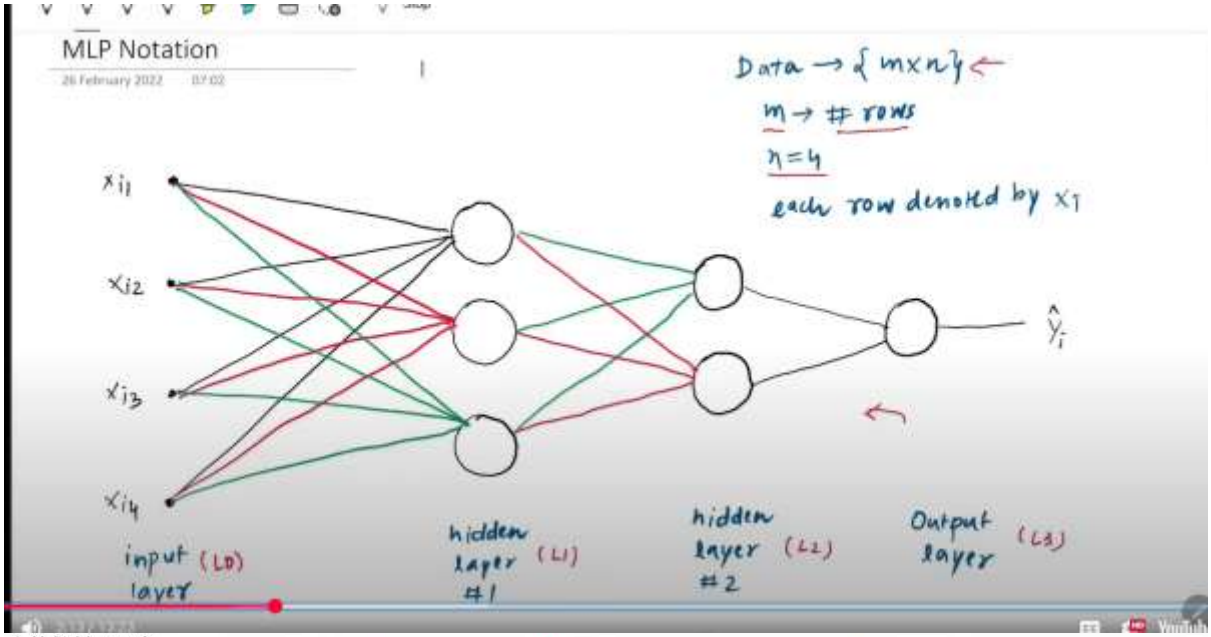

Stop
cgpa
iq
10
5
bias = 3
$5x + 4y + 3 = 0$
$\hat{y}$

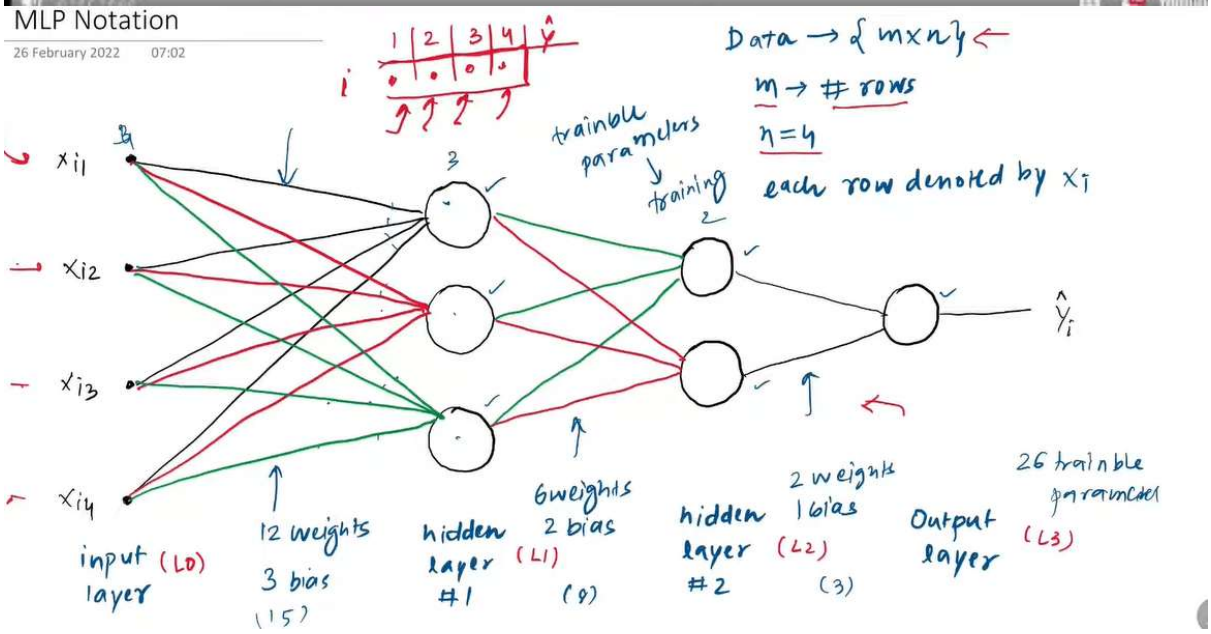

MLP
cgpa
iq
2
5
3
4
10
5
$\hat{y}$
input layer
hidden layer
output layer

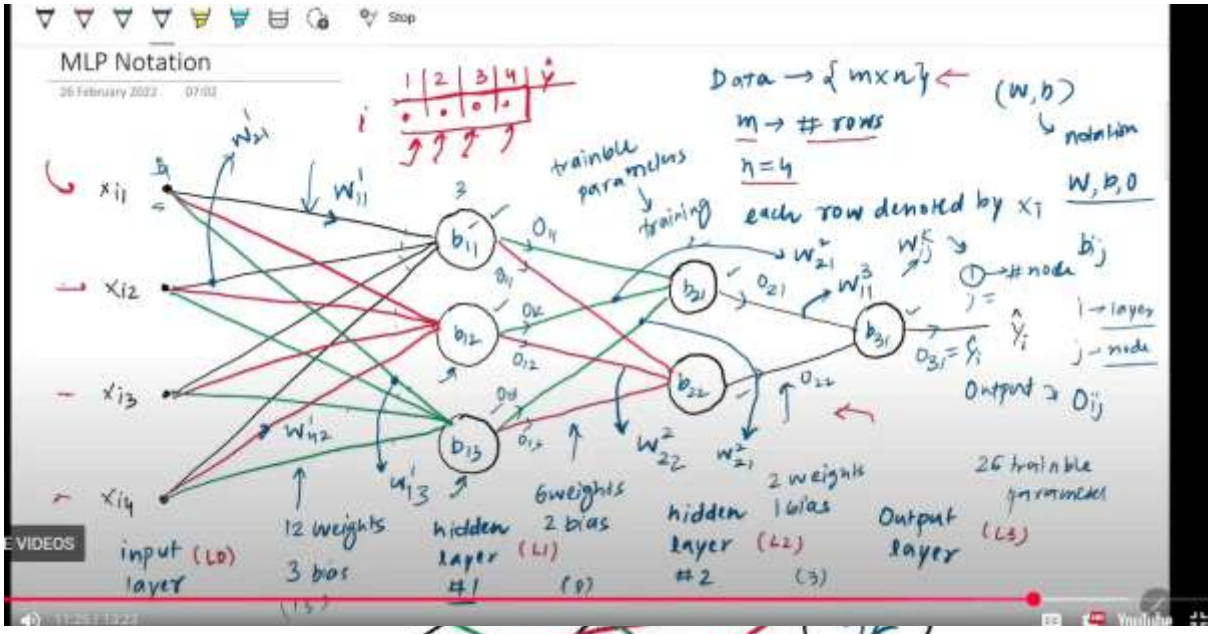


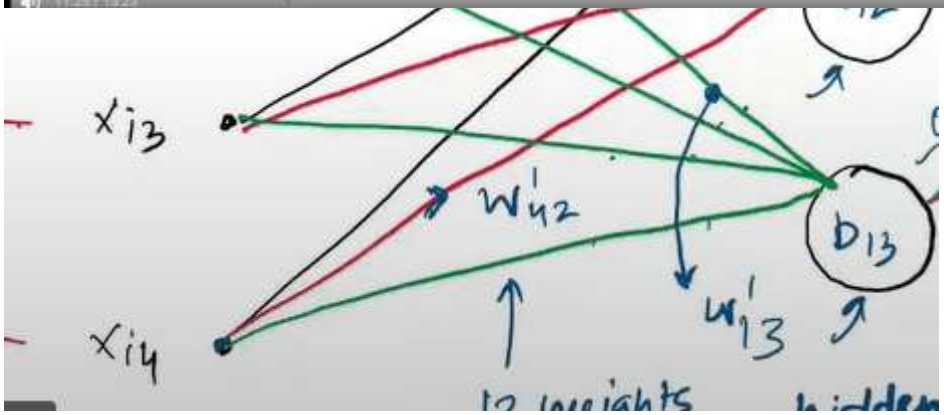
complex non linear data




MLP Notation
26 February 2022 07:02
Data → { m x n }
m → # rows
n = 4
each row denoted by x_i
x_{i1}
x_{i2}
x_{i3}
x_{i4}
$\hat{y}_i$
input layer (L0)
hidden layer #1 (L1)
hidden layer #2 (L2)
Output layer (L3)


MLP Notation
26 February 2022 07:02
Data → { m x n }
m → # rows
n = 4
each row denoted by x_i
trainable parameters → training
x_{i1}
x_{i2}
x_{i3}
x_{i4}
$\hat{y}_i$
12 weights
3 bias
(15)
6 weights
2 bias
(8)
2 weights
1 bias
(3)
26 trainable parameters
input layer (L0)
hidden layer #1 (L1)
hidden layer #2 (L2)
Output layer (L3)





here W142 means

1 -> going to layer 1

4 -> coming from node 4

2 -> going to node 2 g

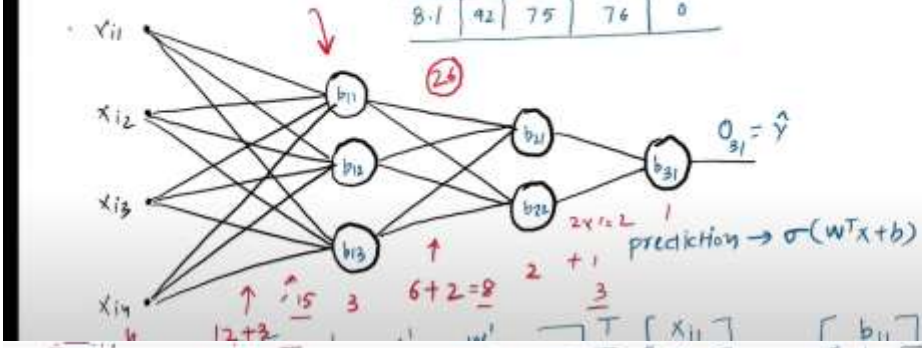
Forward Propogartion

Forward Propogation

03 March 2022 06:07

cgpa	iq	10th m	12th m	placed
7.2	72	69	81	1
8.1	92	75	76	0

→ # of trainable parameters



Layer #1

$$\begin{bmatrix} w'_{11} & w'_{12} & w'_{13} \\ w'_{21} & w'_{22} & w'_{23} \\ w'_{31} & w'_{32} & w'_{33} \\ w'_{41} & w'_{42} & w'_{43} \end{bmatrix}^T \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \\ x_{i4} \end{bmatrix} + \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix}$$

4x3 → 3x4 ; 3x1 ; 4x1 ; 3x1 → 3x1

$$= \begin{bmatrix} w'_{11}x_{i1} + w'_{21}x_{i2} + w'_{31}x_{i3} + w'_{41}x_{i4} \\ w'_{12}x_{i1} + w'_{22}x_{i2} + w'_{32}x_{i3} + w'_{42}x_{i4} \\ w'_{13}x_{i1} + w'_{23}x_{i2} + w'_{33}x_{i3} + w'_{43}x_{i4} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix}$$

3x1

VIDEOS

$$= \sigma\left(\begin{bmatrix} w'_{11}x_{i1} + w'_{21}x_{i2} + w'_{31}x_{i3} + w'_{41}x_{i4} + b_{11} \\ w'_{12}x_{i1} + w'_{22}x_{i2} + w'_{32}x_{i3} + w'_{42}x_{i4} + b_{12} \\ w'_{13}x_{i1} + w'_{23}x_{i2} + w'_{33}x_{i3} + w'_{43}x_{i4} + b_{13} \end{bmatrix} \right)$$

$$= \begin{bmatrix} O_{11} \\ O_{12} \\ O_{13} \end{bmatrix}$$

Layer #2

$$\begin{bmatrix} w^2_{11} & w^2_{12} \\ w^2_{21} & w^2_{22} \\ w^2_{31} & w^2_{32} \end{bmatrix}^T \begin{bmatrix} o_{11} \\ o_{12} \\ o_{13} \end{bmatrix} + \begin{bmatrix} b_{21} \\ b_{22} \end{bmatrix} \rightarrow$$

$3\times2 \xrightarrow{T} 2\times3 \qquad 3\times1 \qquad 2\times1$

$$= \sigma\left(\begin{bmatrix} w^2_{11}o_{11} + w^2_{21}o_{12} + w^2_{31}o_{13} + b_{21} \\ w^2_{12}o_{11} + w^2_{22}o_{12} + w^2_{32}o_{13} + b_{22} \end{bmatrix} \right) = \begin{bmatrix} o_{21} \\ o_{22} \end{bmatrix}$$

Layer #3

$$\rightarrow \begin{bmatrix} w^3_{11} \\ w^3_{21} \end{bmatrix}^T \begin{bmatrix} o_{21} \\ o_{22} \end{bmatrix} + \begin{bmatrix} b_{31} \end{bmatrix}$$

$2\times1 \xrightarrow{T} 1\times2 \qquad 2\times1 \qquad 1\times1$

$$= \sigma\left(\begin{bmatrix} w^3_{11}o_{21} + w^3_{21}o_{22} + b_{31} \end{bmatrix} \right) = \hat{y}_i$$

$$a^{[1]} = \sigma(a^{[0]} w^{[1]} + b^{[1]})$$

$$a^{[2]} = \sigma(a^{[1]} w^{[2]} + b^{[2]})$$

$$o_{31} = \hat{y} \qquad a^{[3]} = \sigma(a^{[2]} w^{[3]} + b^{[3]})$$

diction $\rightarrow \sigma(w^T x + b)$

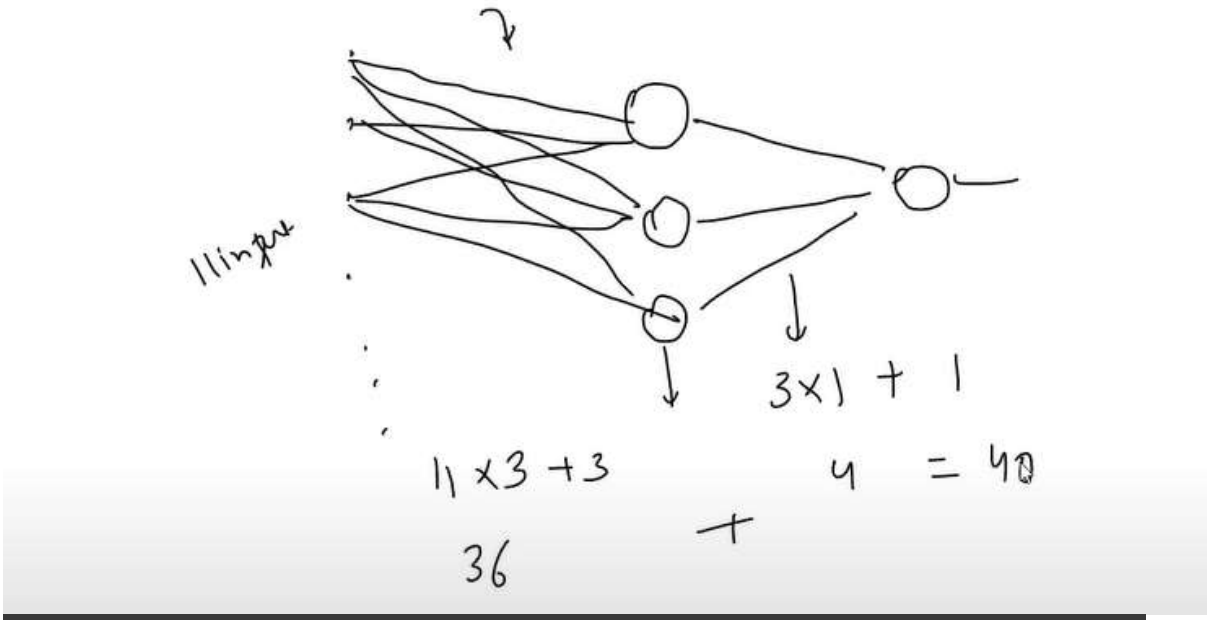
$$\sigma\left(\sigma\left(\left(\sigma(a^{[0]} w^{[1]} + b^{[1]})\right) w^{[2]} + b^{[2]}\right) w^{[3]} + b^{[3]}\right)$$

$a^{[1]}$ $a^{[2]}$

EOS

14:12 / 15:30

a[3] is final output



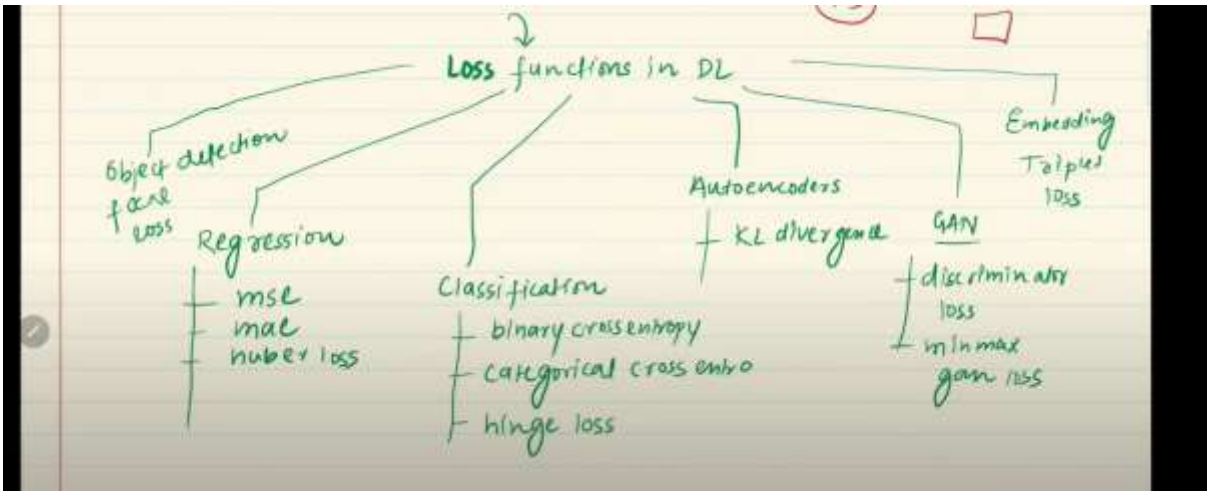

```
model.summary()
```

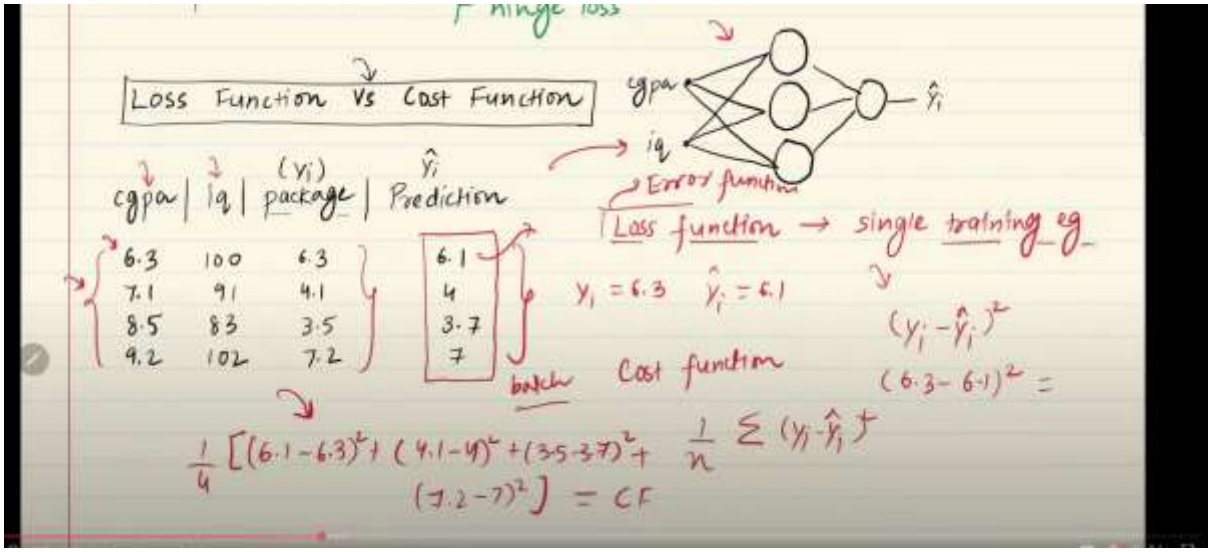
Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 3)	36
dense_1 (Dense)	(None, 1)	4

Total params: 40 (160.00 B)
Trainable params: 40 (160.00 B)
Non-trainable params: 0 (0.00 B)

Loss functions based on problem

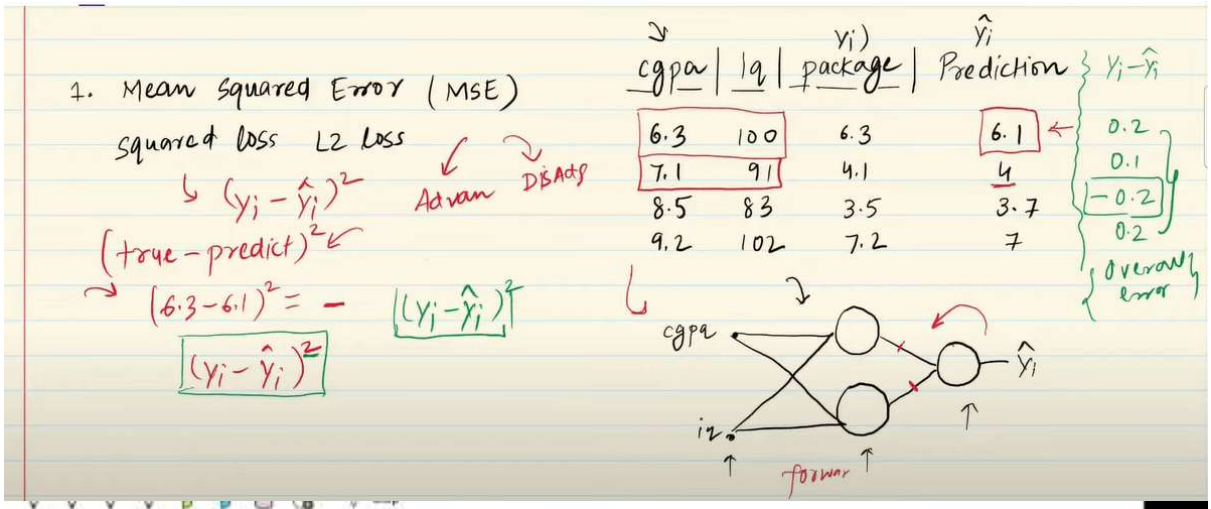


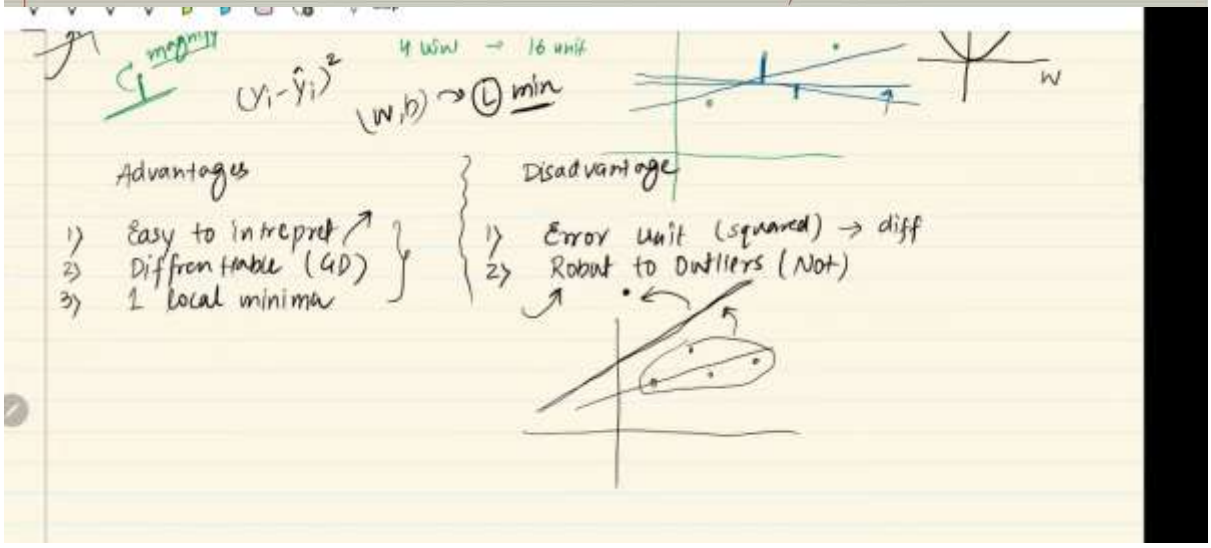


loss function is for single training example

cost function is for entire batch

so sum of all loss functions





for MSE last neuron should be linear type

```
model = Sequential()

model.add(Dense(7,activation='relu',input_dim=7))
model.add(Dense(7,activation='relu'))
model.add(Dense(1,activation='linear'))
```

+ Code + Markdown

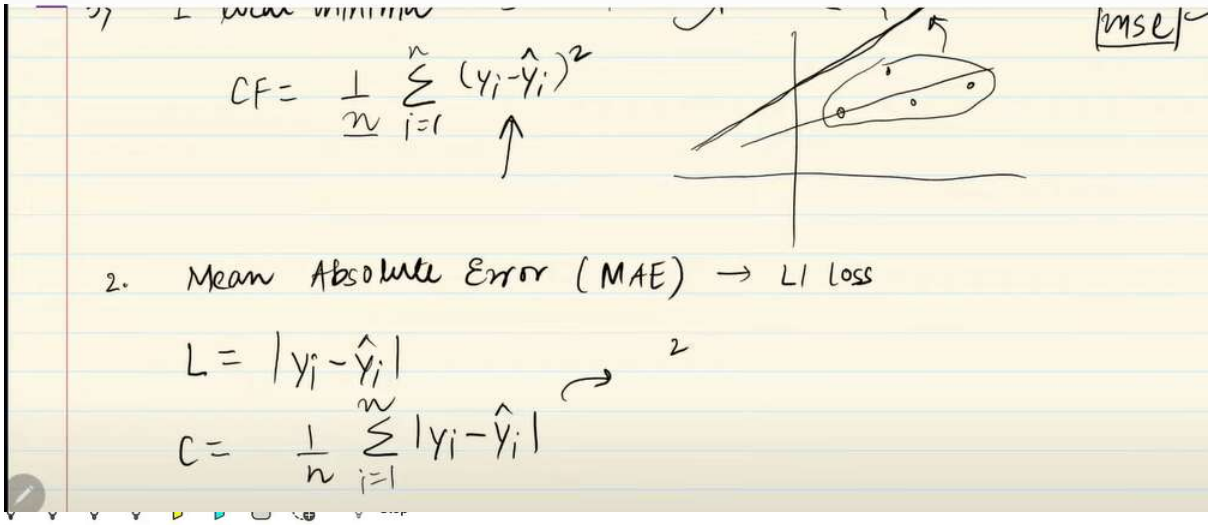
[ ]:

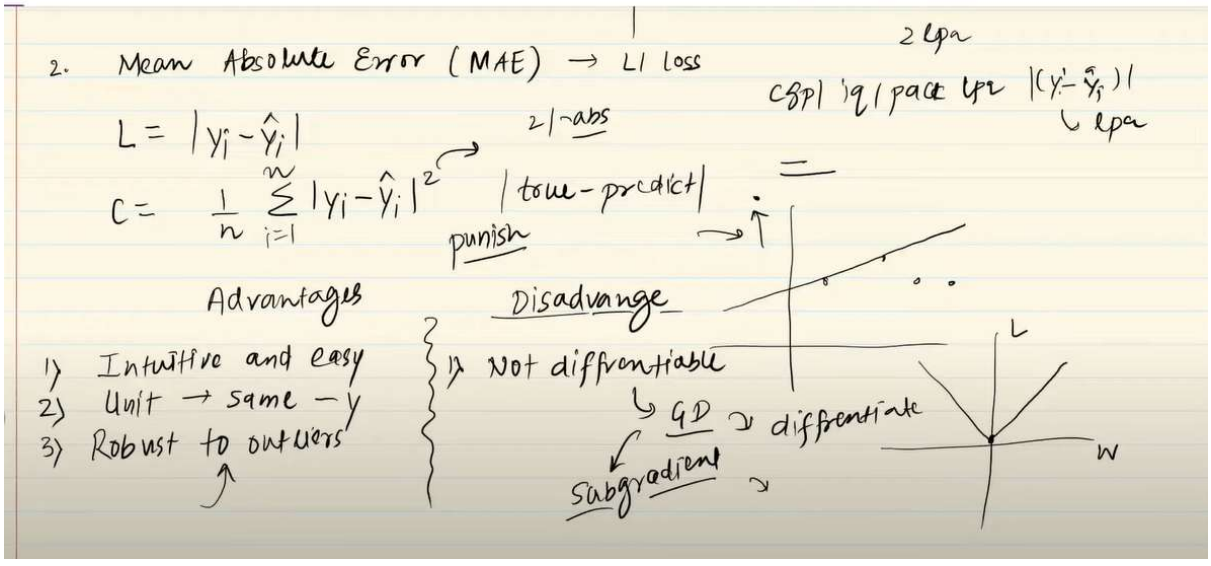
```
model.summary()
```

+ Code + Markdown

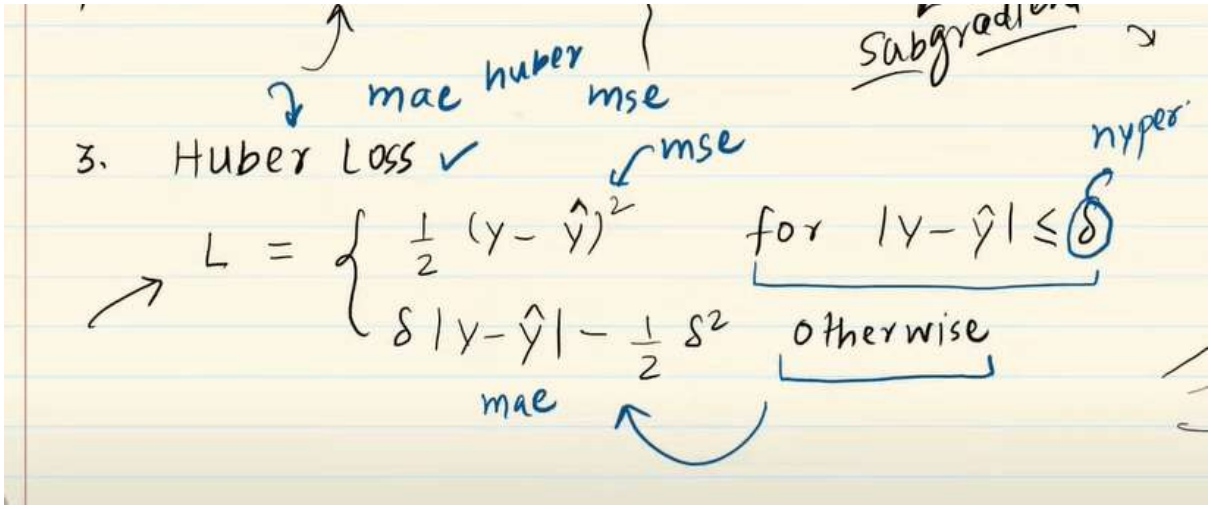
[ ]:

```
model.compile(loss='mean_squared_error',optimizer='Adam')
```



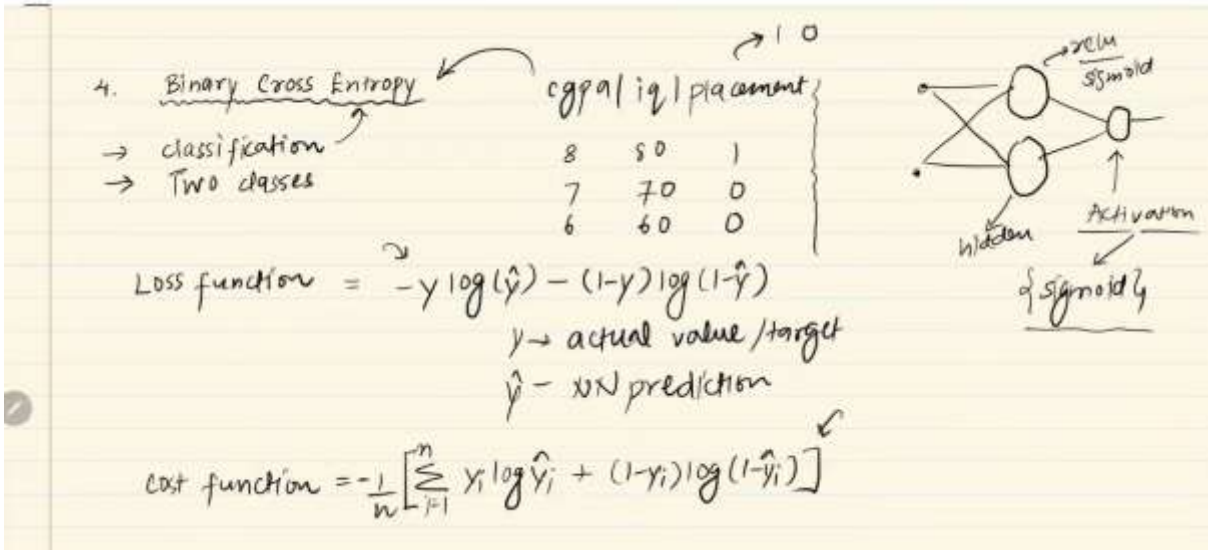


MSE doesnt handle outliers and MAE goes slowly towards outliers



Binary cross entropy also called logloss is used when

- classification problem
- output neuron is YES NO , 0 1
- output function should be sigmoid

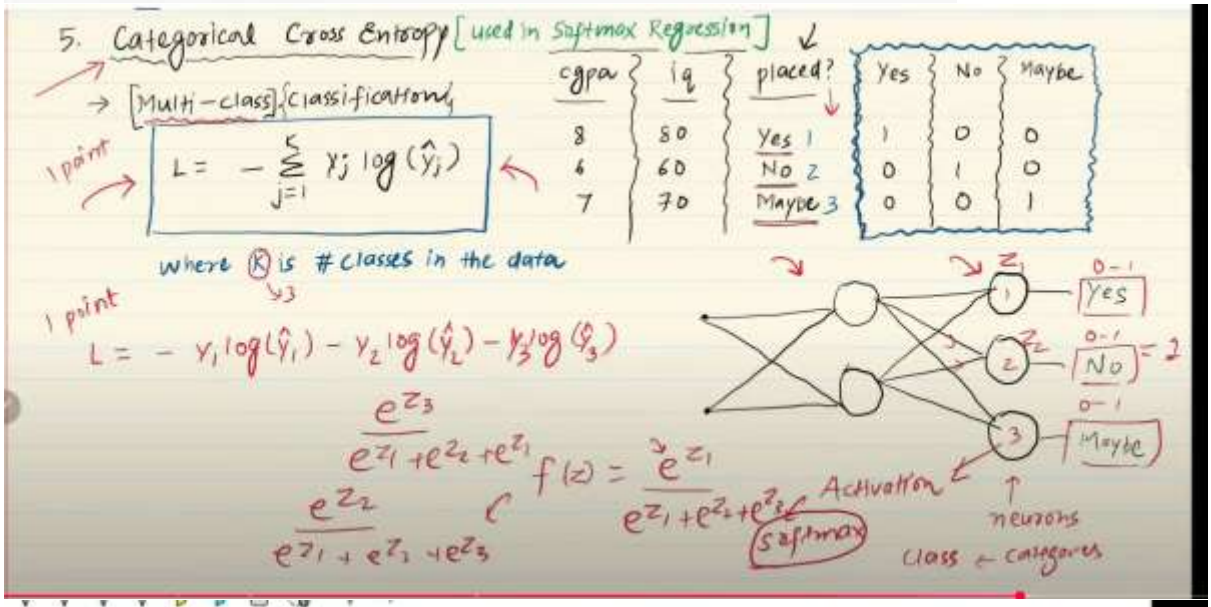


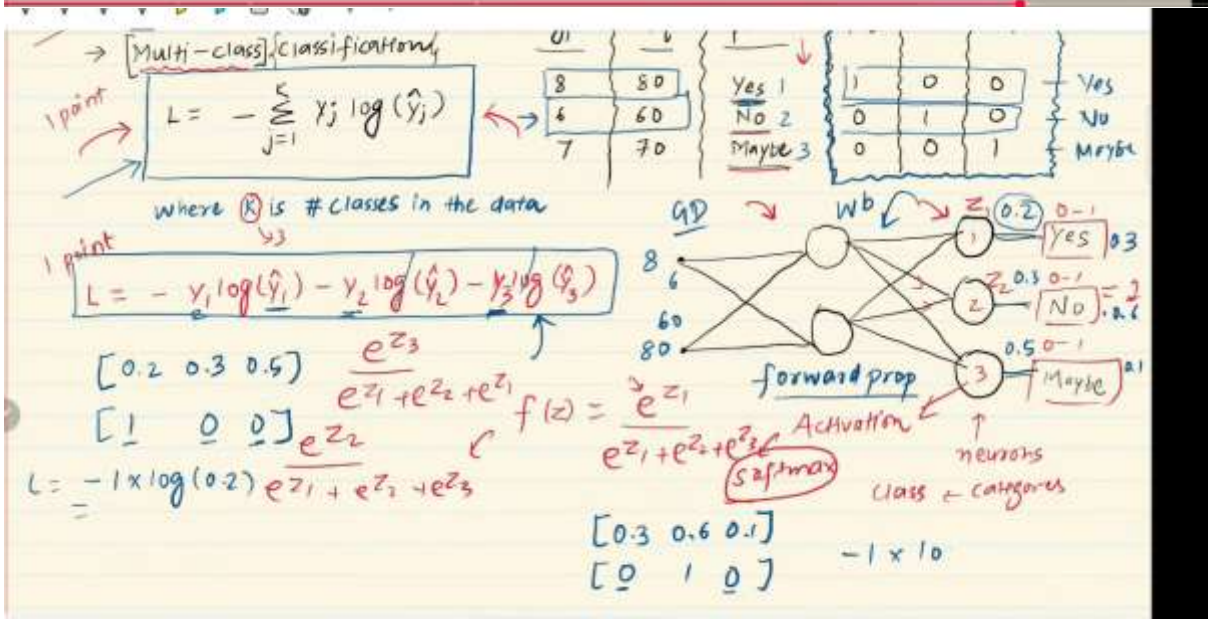
categorical cross entropy is used when

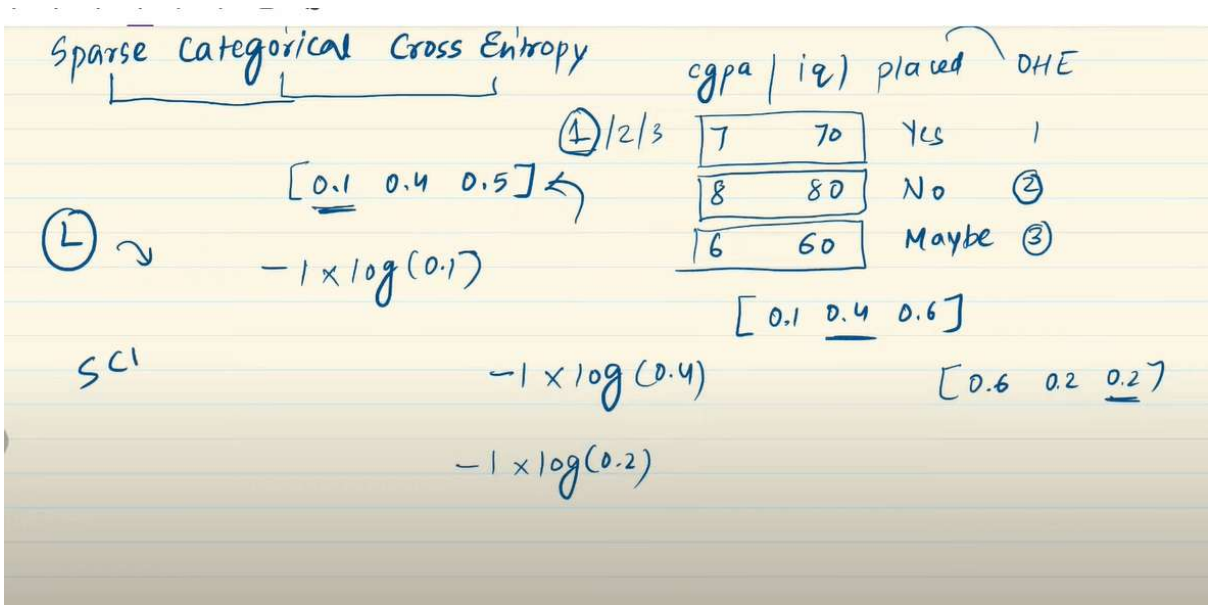
- multiclass classification problem
- output neuron is YES NO maybe
- output function should be softmax

```
model = Sequential()

model.add(Flatten(input_shape=(28,28)))
model.add(Dense(128,activation='relu'))
model.add(Dense(32,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

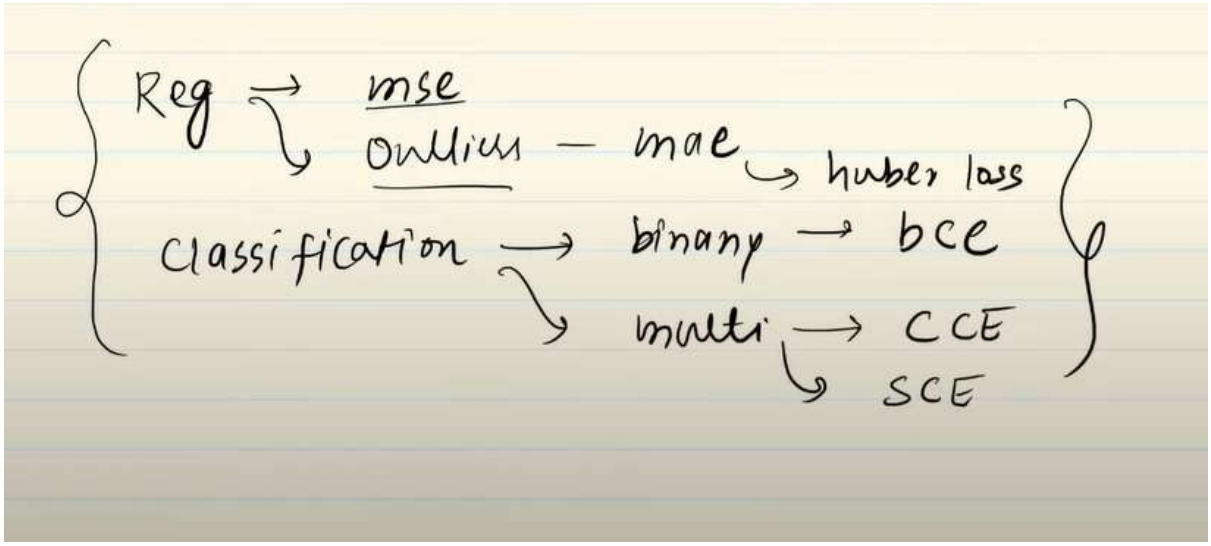
```
model.summary()
```







if there are too many classes we should use sparse categorical cross entropy



Backpropagation, short for "backward propagation of errors", is a fundamental algorithm used to train artificial neural networks (ANNs). It plays a crucial role in enabling neural networks to learn from data and improve their predictive accuracy

Training means finding correct values of weights and biases

1. **Forward Pass:** Input data is fed through the neural network, generating an output.
2. **Error Calculation:** The network's output is compared to the desired output, and a loss function calculates the error or difference between them.
3. **Backward Pass (Backpropagation):** The error is then propagated backward through the network, from the output layer towards the input layer. Using the chain rule of calculus, backpropagation calculates the gradient of the loss function with respect to

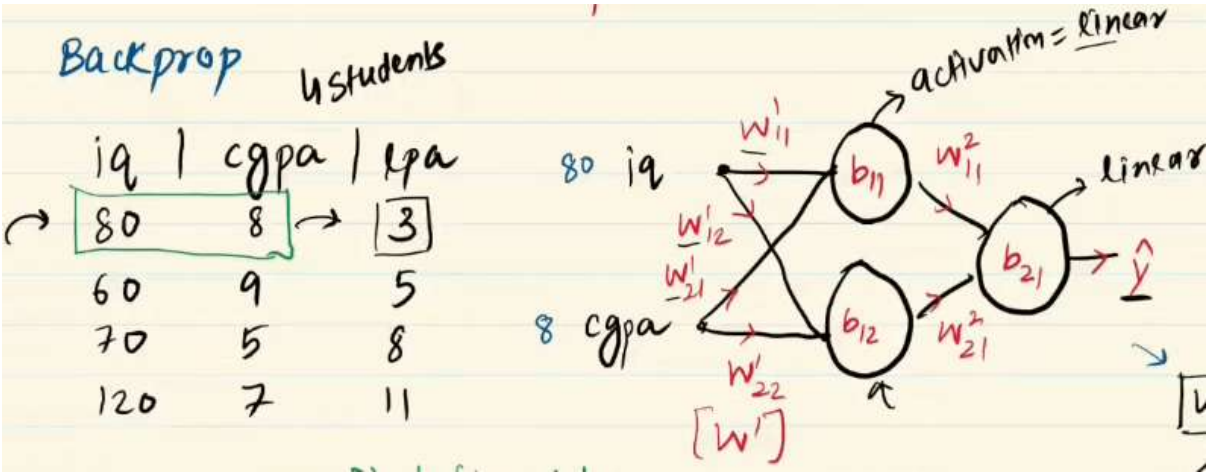
each weight and bias in the network. This essentially tells us how much each weight and bias contributes to the overall error.

4. **Weight and Bias Update:** Optimization algorithms like gradient descent or stochastic gradient descent use these gradients to adjust the weights and biases of the network. The goal is to minimize the loss function, thereby reducing the error and improving the network's performance.
5. **Iteration:** Steps 1-4 are repeated iteratively with multiple passes over the training data until the network learns to make accurate predictions

step 1 - initialize random values for weights and biases

u can also initialize 1 for weights and 0 for biases

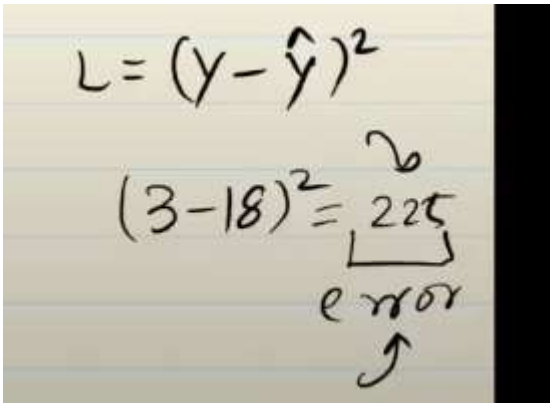
step 2 - select 1 data entry and give neurons values as inputs



step 3 - do initial prediction with inputs , weights = 1 , biases = 0

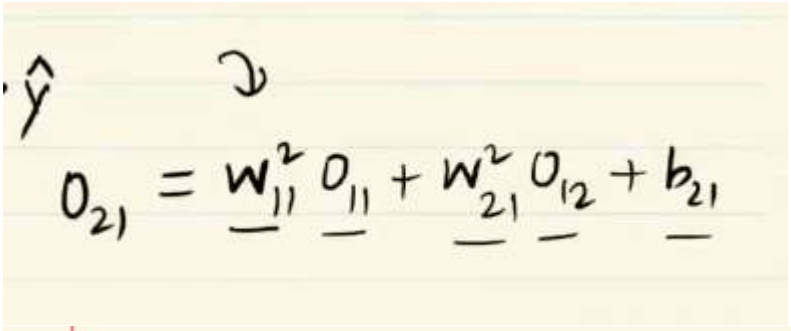
for now assume prediction came out as 18LPA --- but it should be 3 LPA

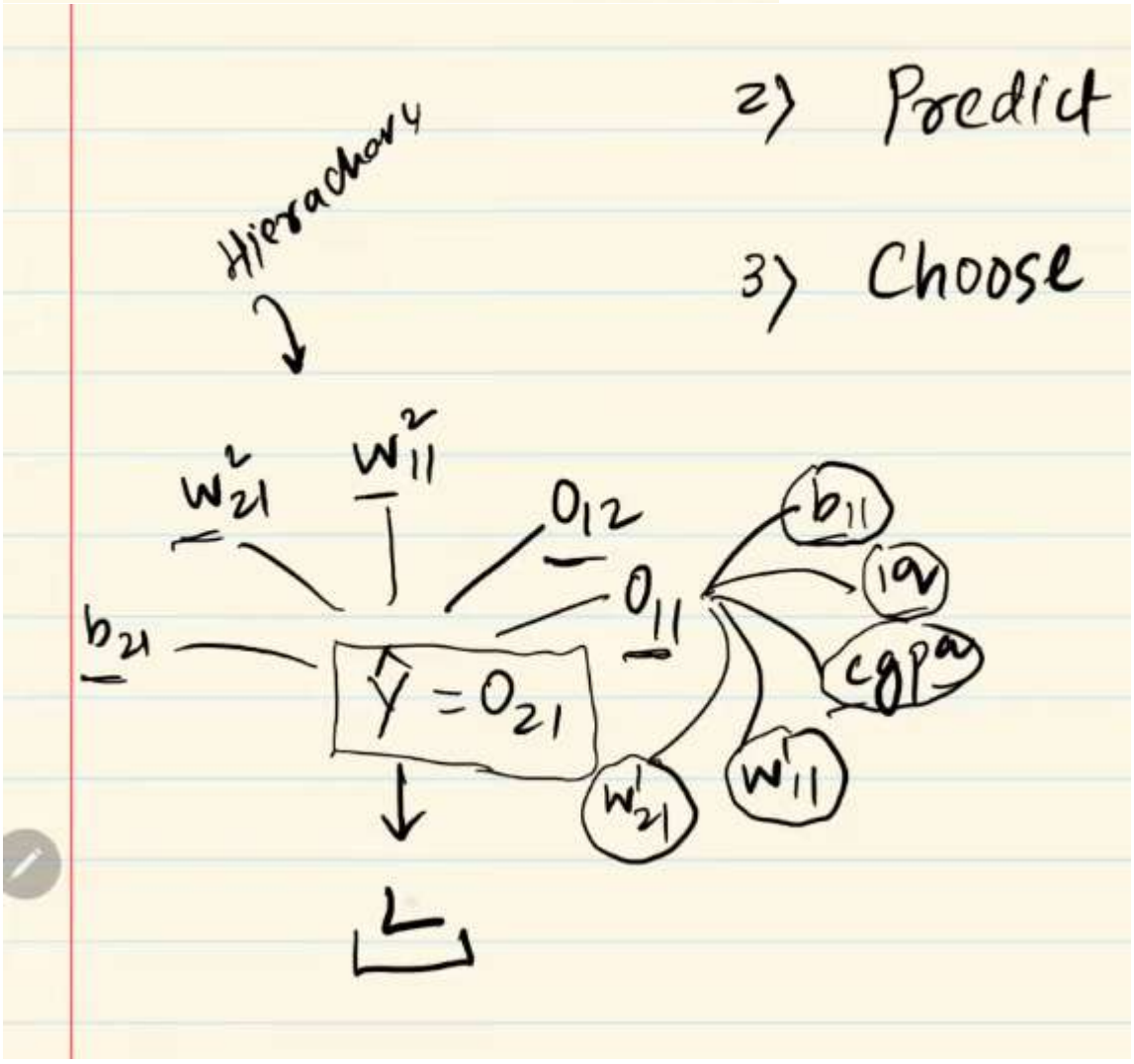
step 4 - choose 1 loss function and calculate error



now to minimize loss function output ---> we need smaller y hat value ---> y hat value is O21

---> O21 value is dependent on 5 variables ---> and tree grows into branches





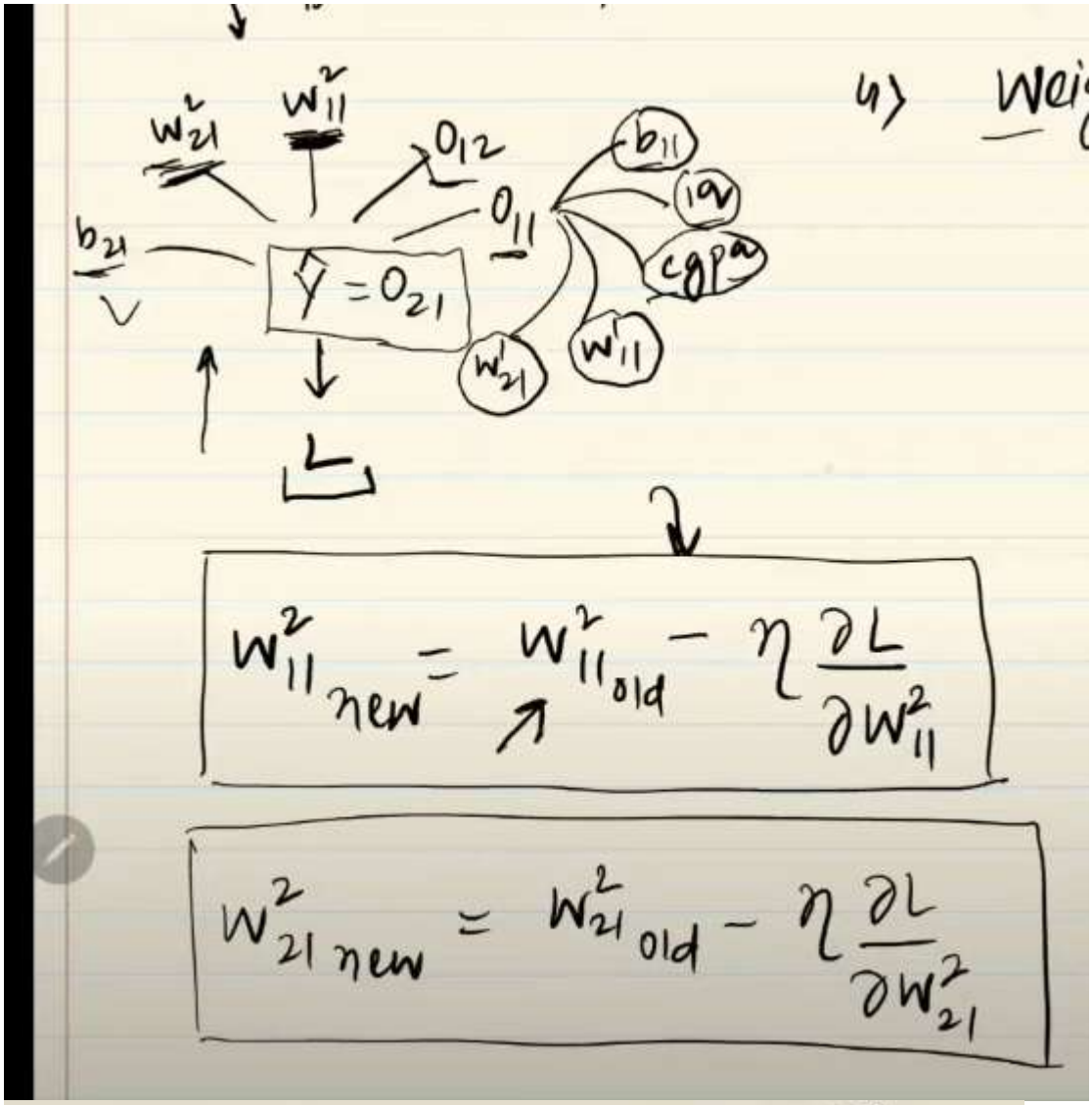
step 5 - update weights and biases using gradient descend

4) Weights and bias update ✓

↳ Gradient descent

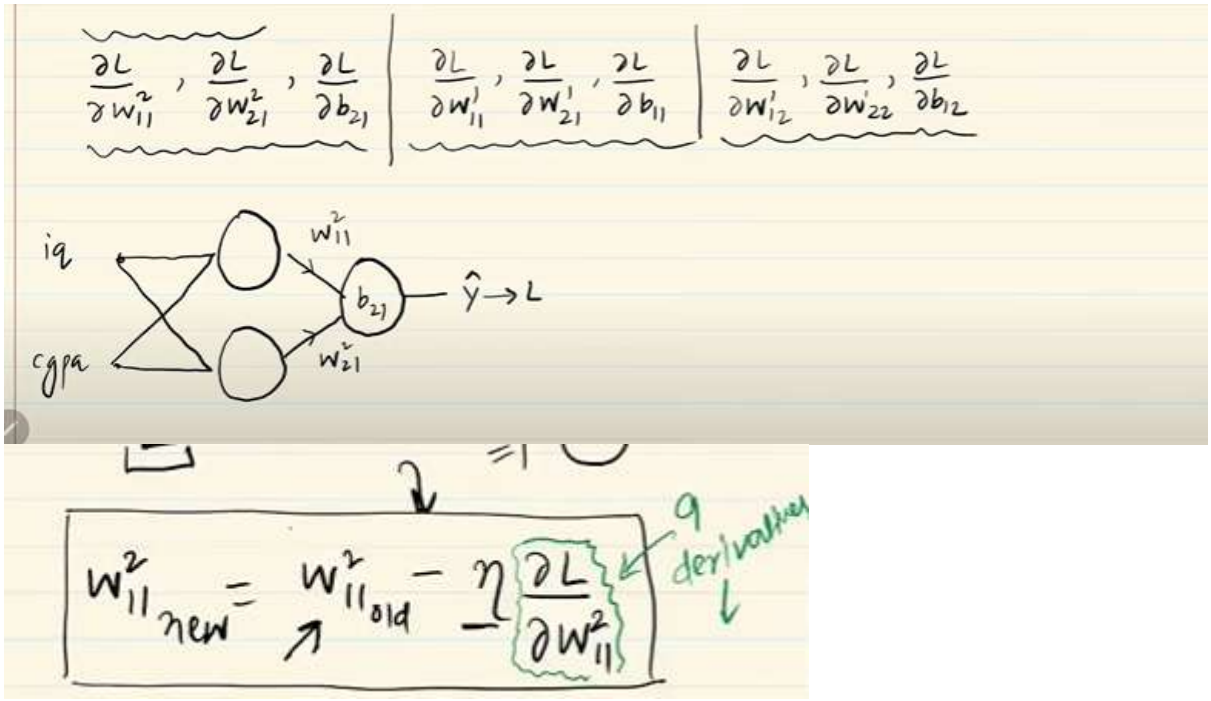
$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w_{old}}$$

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

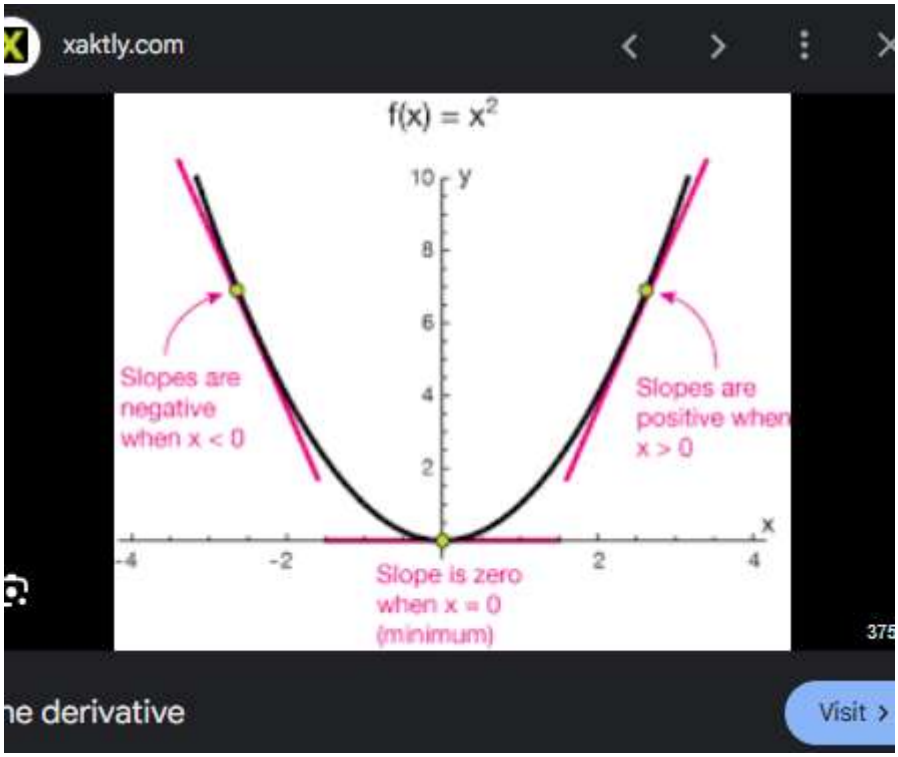


this is derivative of loss wrt weights

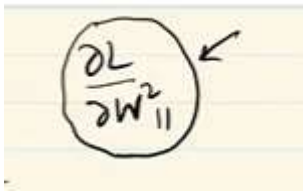
for example below is 9 derivatives (3 for each perceptron)



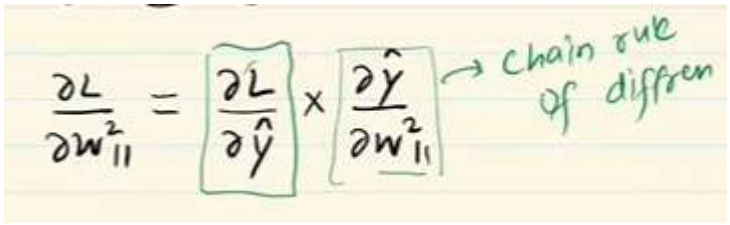
dy/dx says that upon a small change in X what is changes in Y



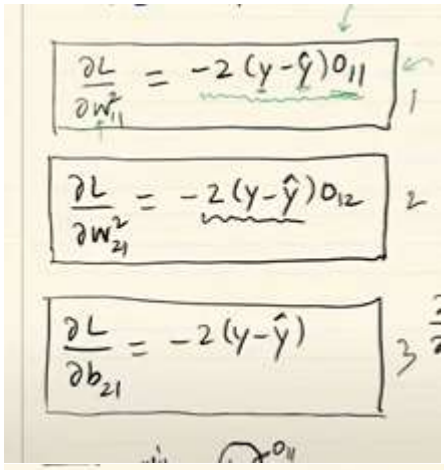
so accordingly below derivative means that when weights are changed what is its impact on L (loss)

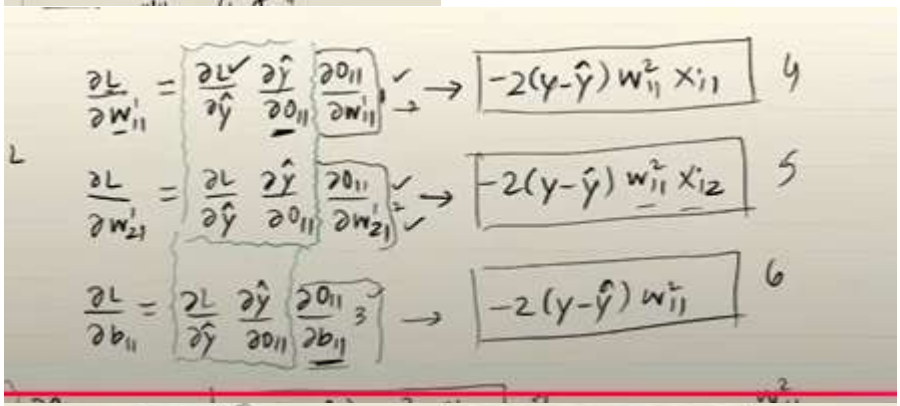


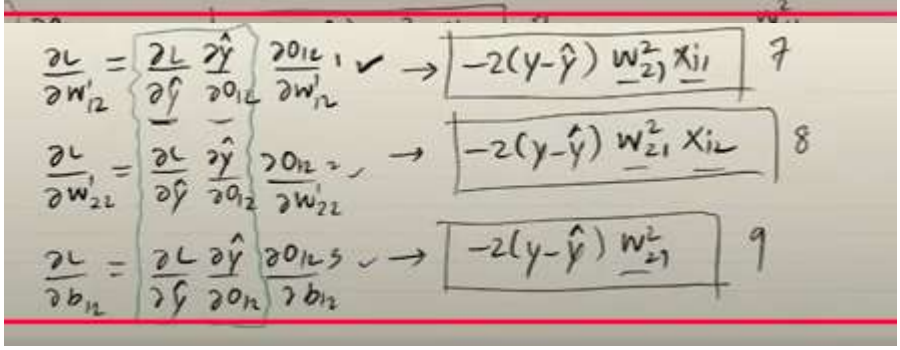
this above is written as

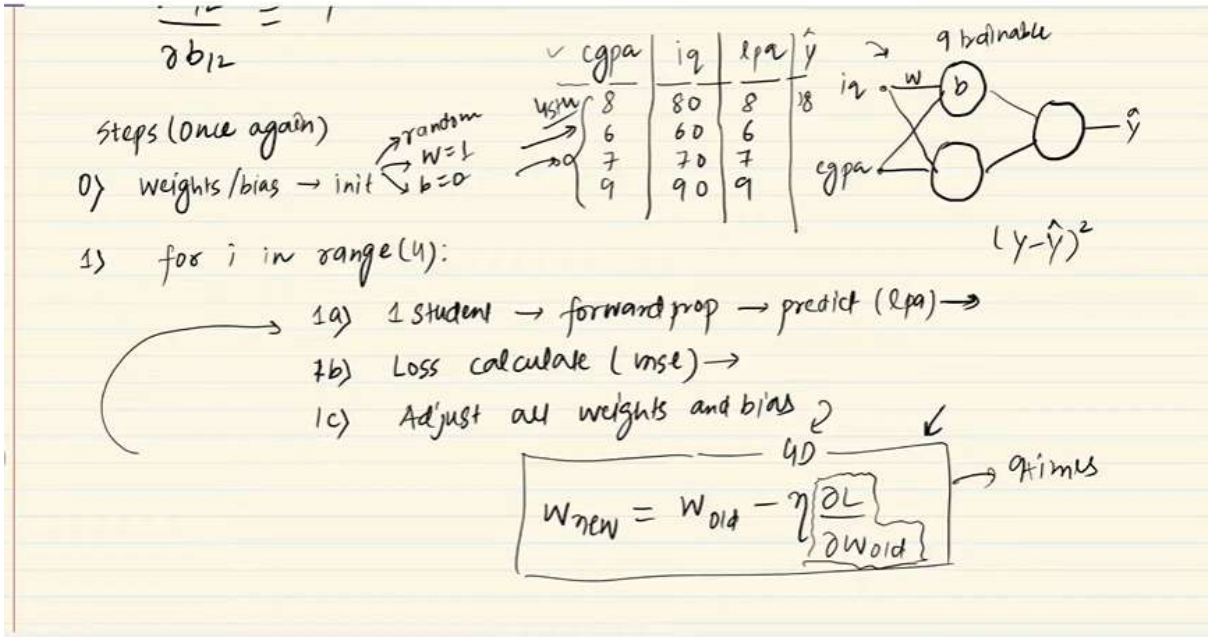


so weights change - > impact on y hat -> y hat changes -> impact on L loss



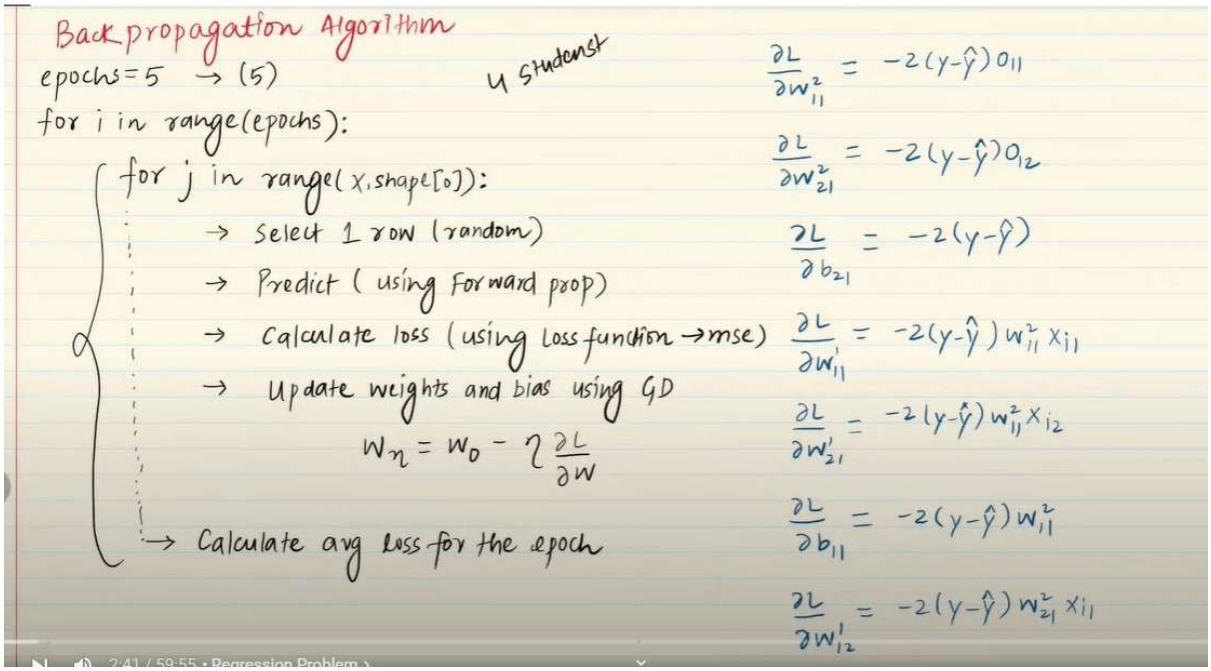






step 1 will be done 100/1000 times till convergence is not met

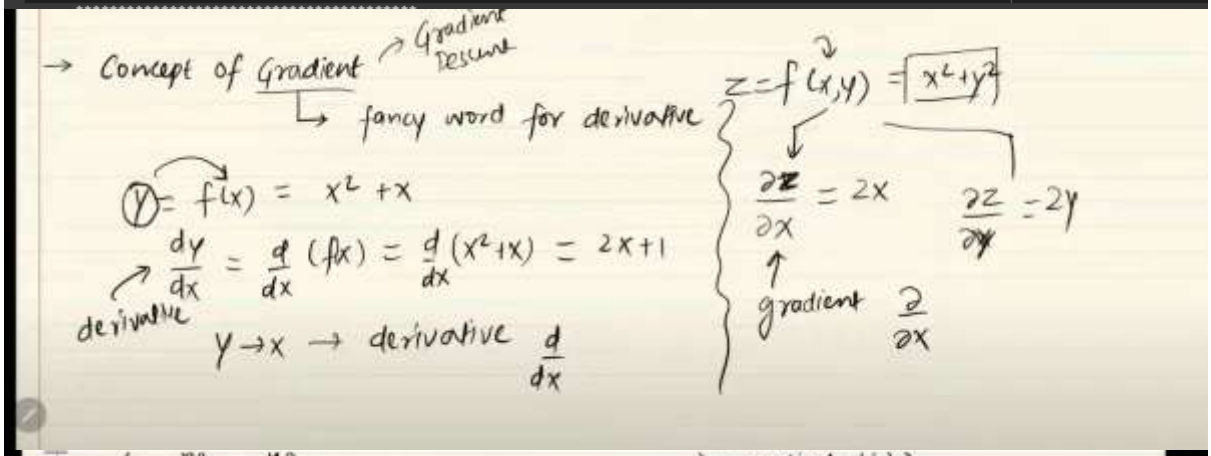
this is called epochs

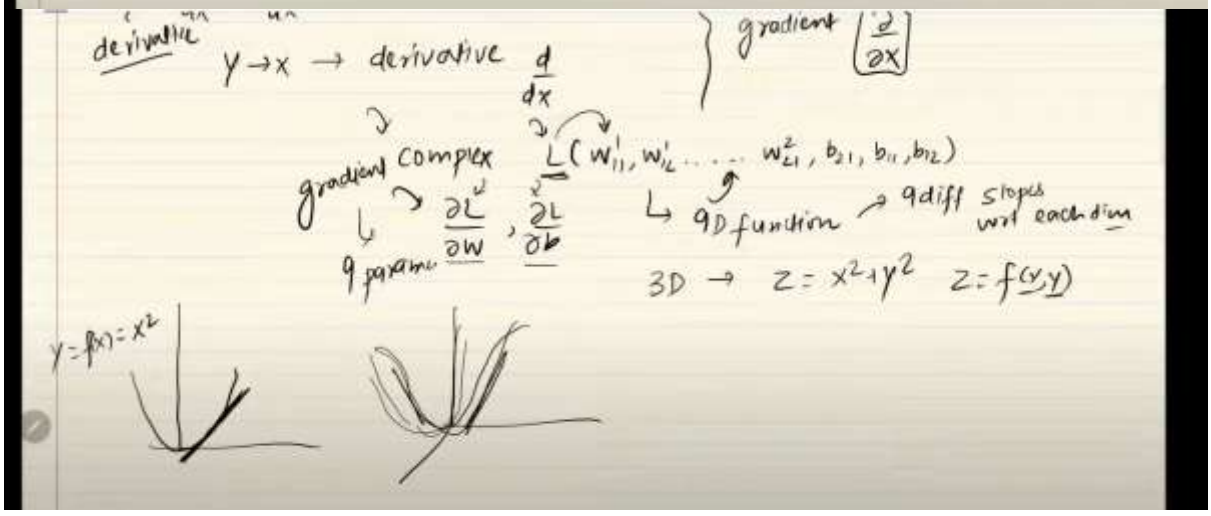


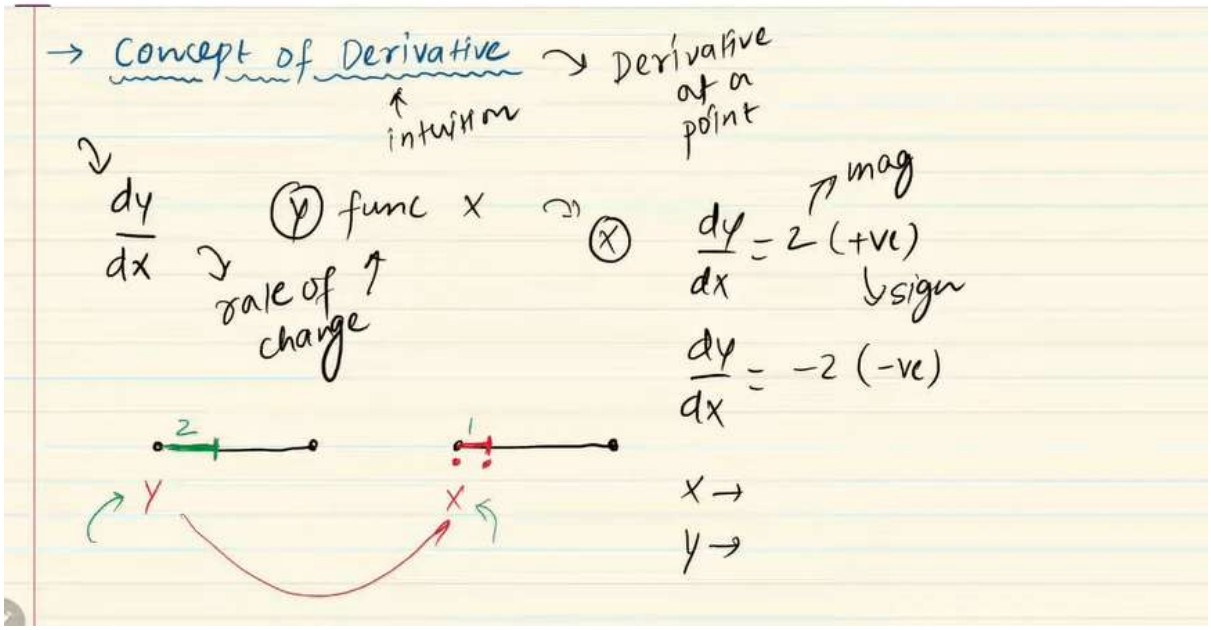
\

\\

```
[22]  # epochs implementation

      parameters = initialize_parameters([2,2,1])
      epochs = 5

      for i in range(epochs):

        Loss = []

        for j in range(df.shape[0]):

          X = df[['cgpa', 'profile_score']].values[j].reshape(2,1) # Shape(no of features, no. of
          training example)
          y = df[['lpa']].values[j][0]

          # Parameter initialization


          y_hat,A1 = L_layer_forward(X,parameters)
          y_hat = y_hat[0][0]

          update_parameters(parameters,y,y_hat,A1,X)

          Loss.append((y-y_hat)**2)

        print('Epoch - ',i+1,'Loss - ',np.array(Loss).mean())

      parameters
```



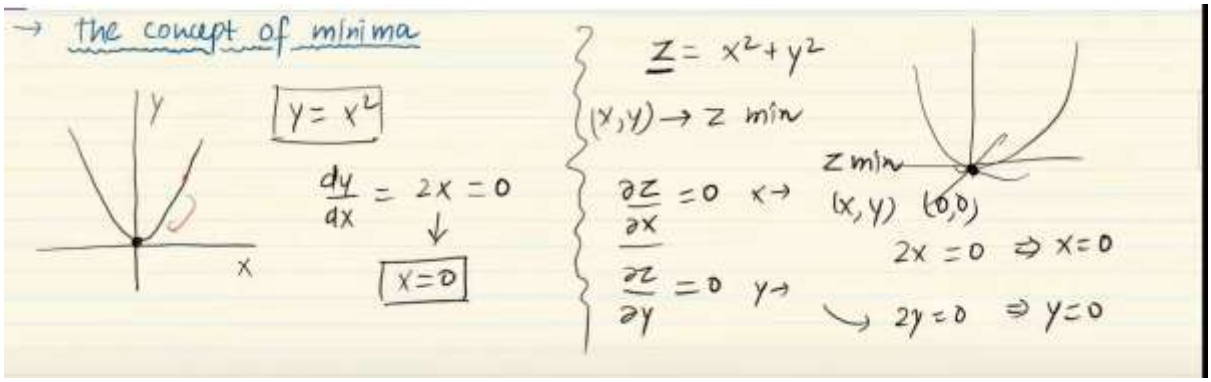




suppose dy/dx = 2 ..... so on 1 times change of x there will be 2 times change in y

2 is magnitude

so 2 is magnitude / slope / rate of change

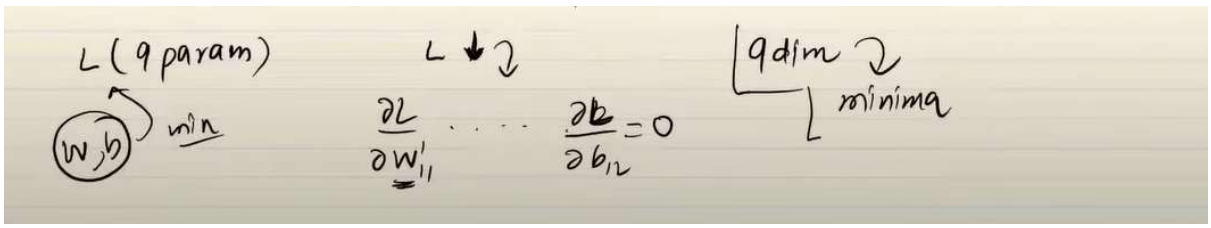


to find minima we just say =0

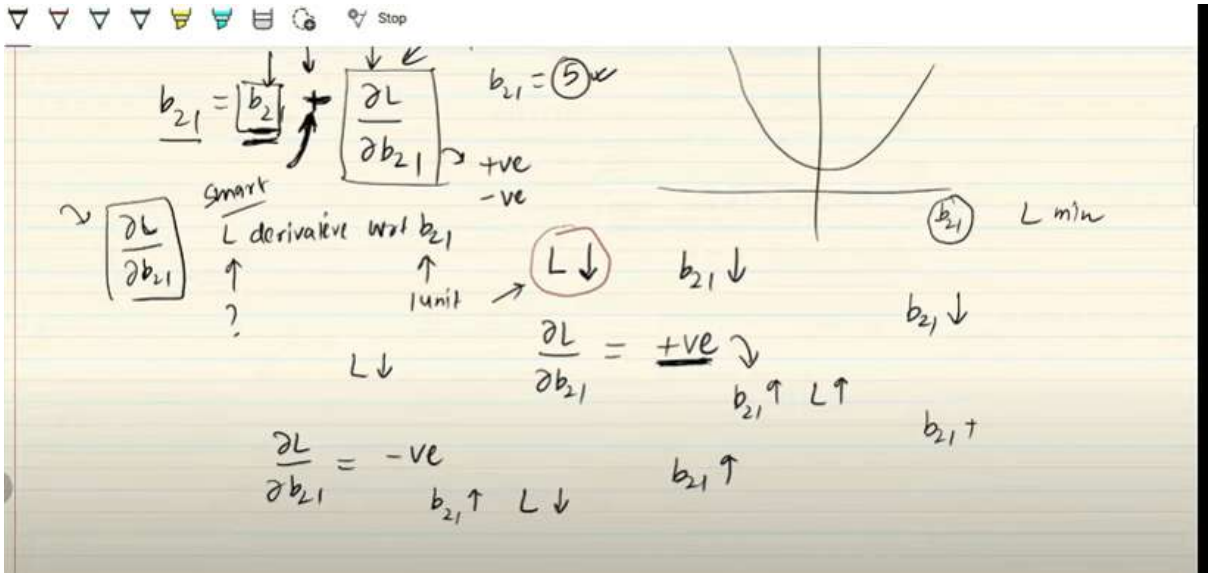
so for y=x^2

we do dy/dx = 2x = 0

so x= 0



for example assume we are changing b21

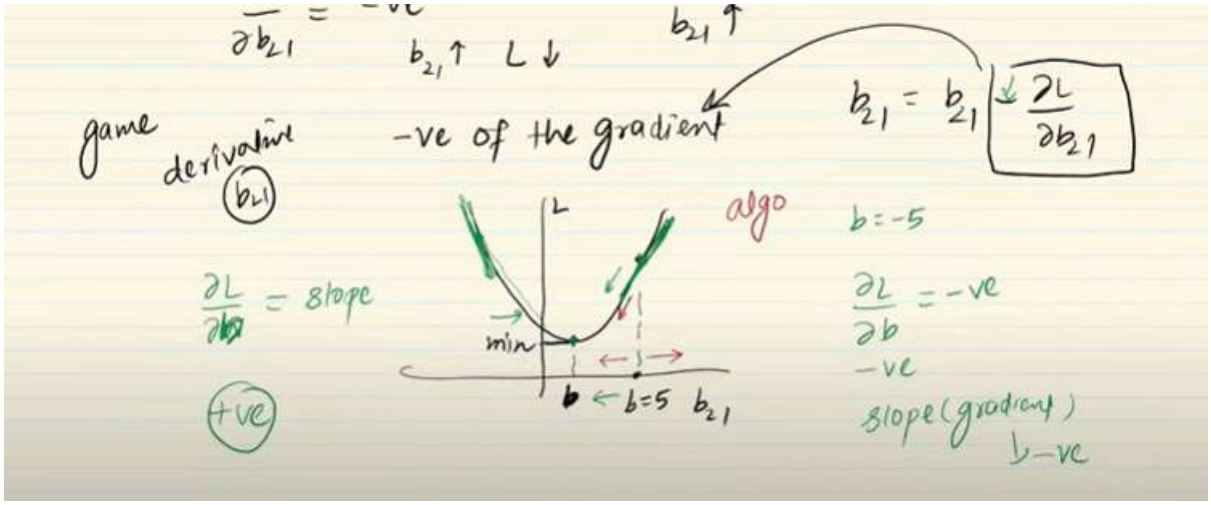


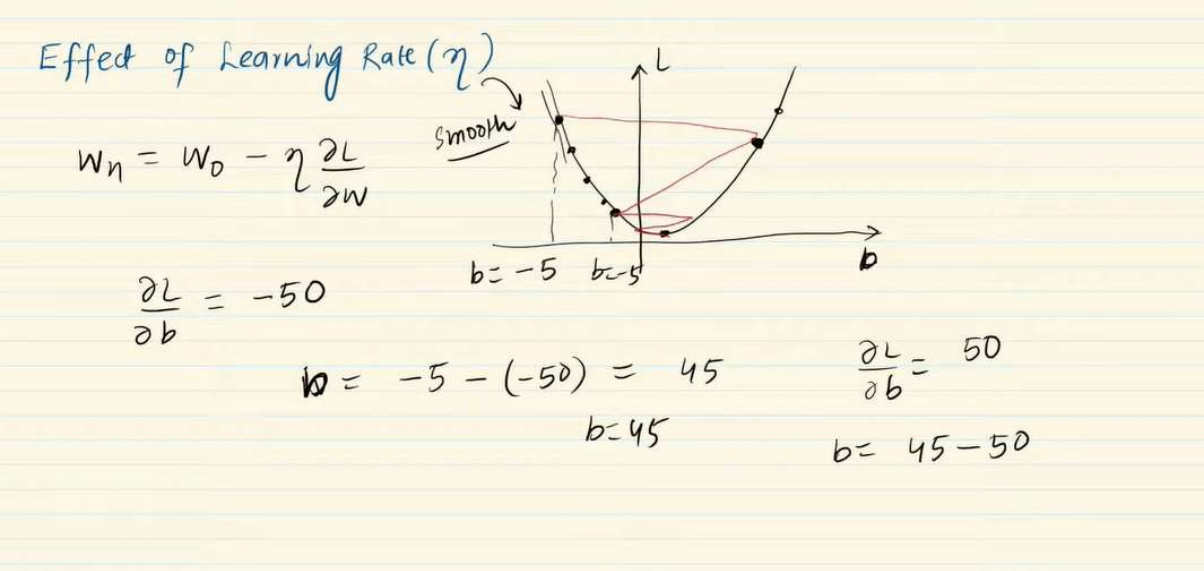
if derivative of L/b21 is +ve then we decrease b21 value

and then subtract it from old b21 value

if derivative of L/b21 is -ve then we increase b21 value

and then subtract it from old b21 value (bcz - + is - anyways)





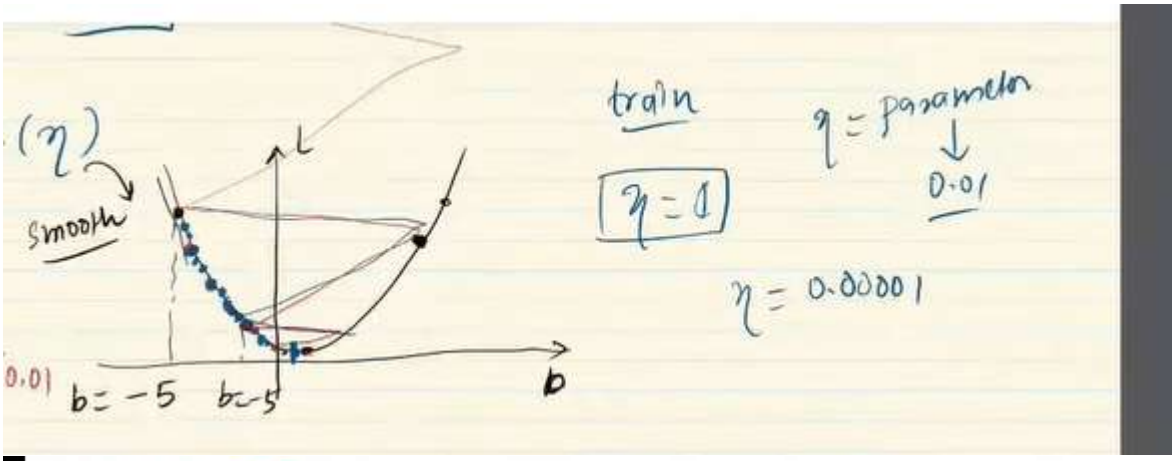
if formula does not have learning rate

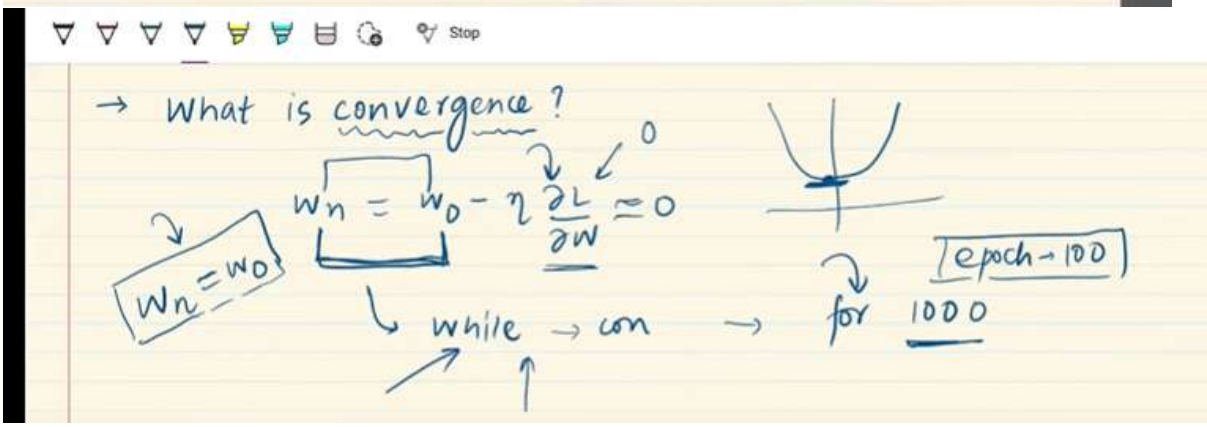
then first b = -5

then b = 45

then b= 5

so its very zig zag





convergence is point where wold and wnew has not much difference

now we dont know exactly how many loops will it take to reach that

so we run it 100/1000 epochs time

developers-dot-devsite-v2-prod.appspot.com/machine-learning/crash-course/backprop-scroll

Gradient Descent

Thursday, April 7, 2022 7:44 AM

→ optimization

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

$J(\theta)$ = Loss function

Neural

$\frac{\partial L}{\partial b}$

Loss (w, b)

rate

step

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 η 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.

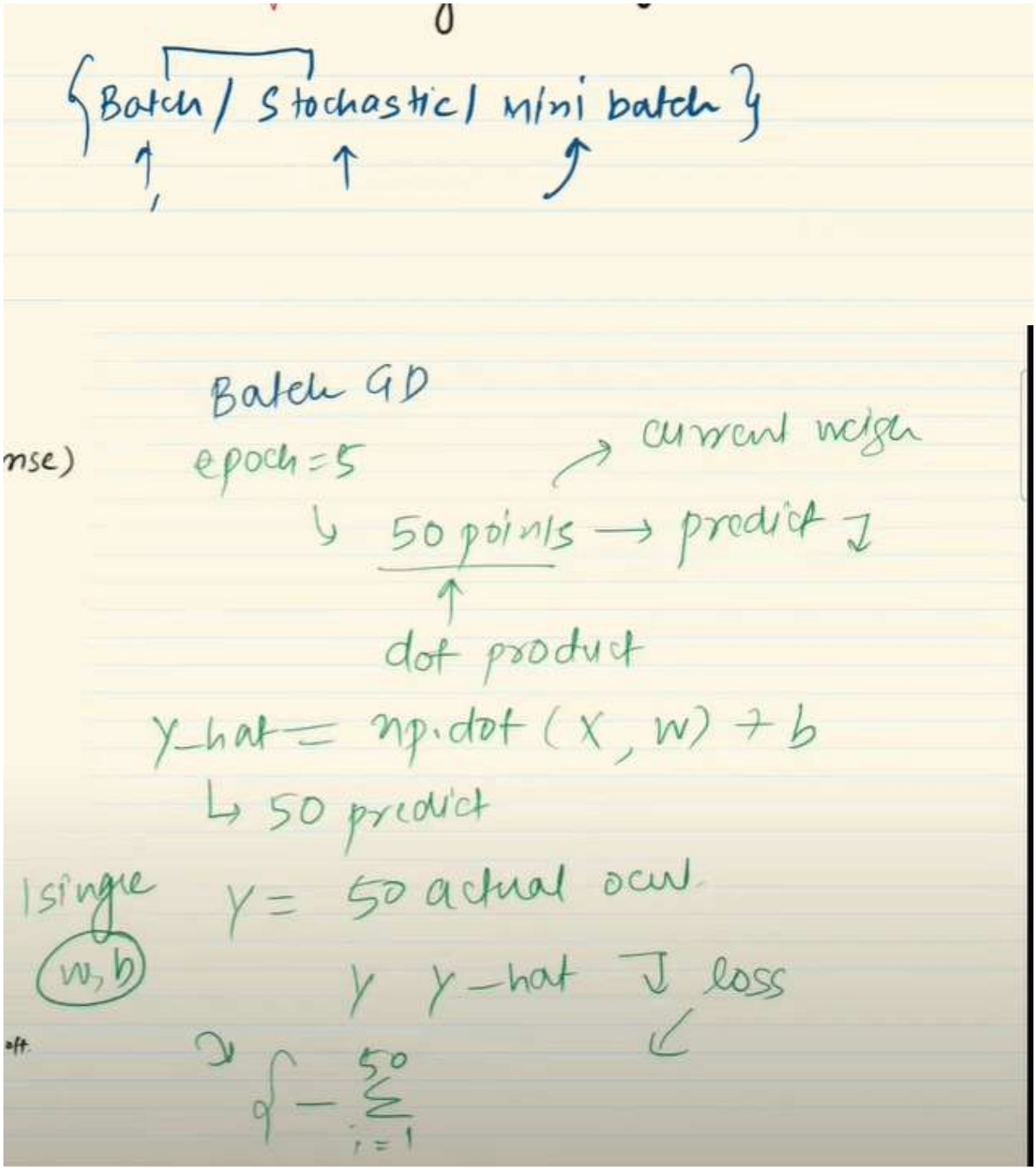
Back propagation Algorithm

```
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 weights and bias using GD
```

$$W_n = W_0 - \eta \frac{\partial L}{\partial W}$$

→ Calculate avg loss for the epoch

Loss



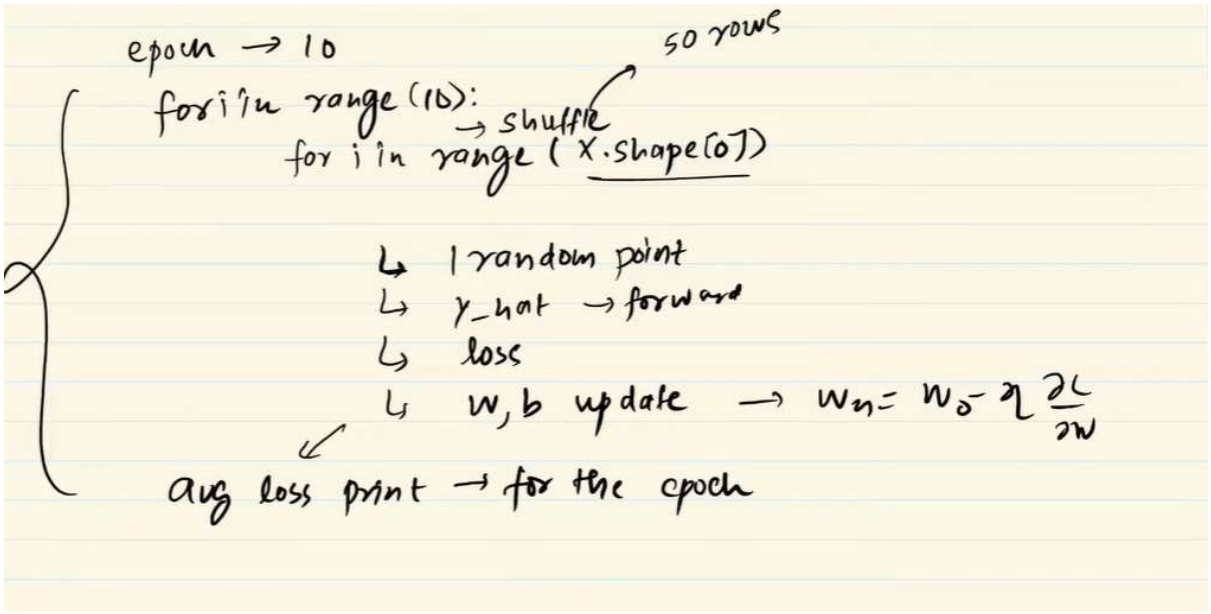
i have curr weights and biases

in a single epoch using current weight and biases all predictions are made for entire dataset

then using 50 new predictions 1 single time weights and biases are updated

so no. of epochs = no. of updates

5 epoch = 5 updates



i have curr weights and biases

in a single epoch using current weight and biases 50 times loop runs everytimes selects 1 random point - > calculates loss and updates w and b

so suppose 10 epoch and 50 rows

then 10*50 = 500 weights , biases updates

---

suppose i have 50 rows

then in code

batch_size = 50 means batch gradient descendent

batch_size = 1 means SGD

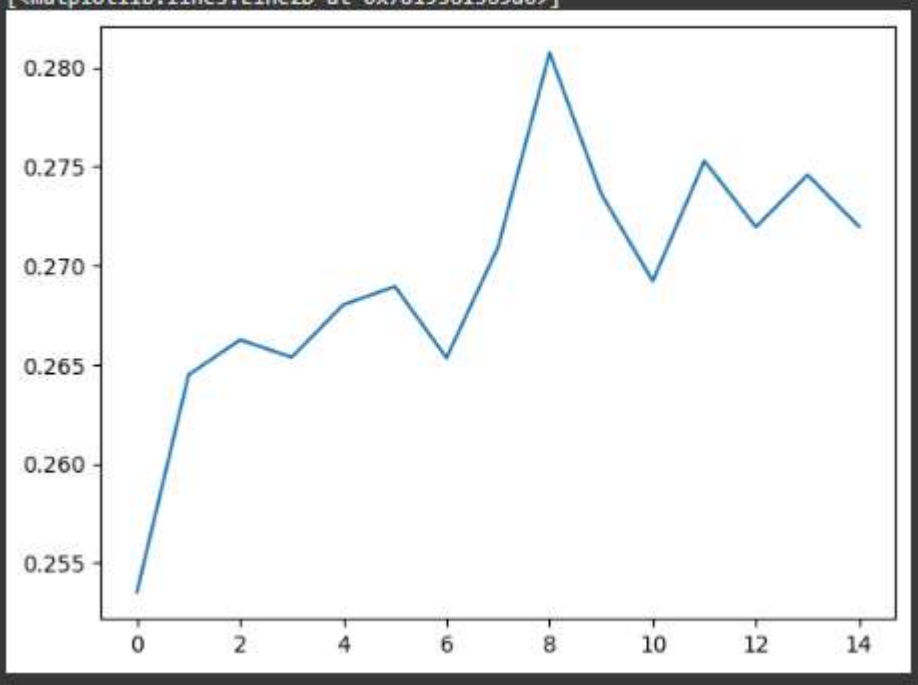
SGD

```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
start = time.time()
history = model.fit(X_scaled,y,epochs=15,batch_size=1,validation_split=0.2)
print(time.time() - start)
```

```
Epoch 1/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8601 - loss: 0.2756 - val_accuracy: 0.9750 - val_loss: 0.1789
Epoch 2/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8688 - loss: 0.3250 - val_accuracy: 0.9750 - val_loss: 0.1809
Epoch 3/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8676 - loss: 0.2654 - val_accuracy: 0.9750 - val_loss: 0.1705
Epoch 4/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8794 - loss: 0.2818 - val_accuracy: 0.9750 - val_loss: 0.1657
Epoch 5/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8969 - loss: 0.2699 - val_accuracy: 0.9750 - val_loss: 0.1653
Epoch 6/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9141 - loss: 0.2210 - val_accuracy: 0.9750 - val_loss: 0.1637
Epoch 7/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8863 - loss: 0.2333 - val_accuracy: 0.9875 - val_loss: 0.1672
Epoch 8/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8721 - loss: 0.2865 - val_accuracy: 0.9875 - val_loss: 0.1621
Epoch 9/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8873 - loss: 0.2841 - val_accuracy: 0.9875 - val_loss: 0.1596
Epoch 10/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8693 - loss: 0.2953 - val_accuracy: 0.9875 - val_loss: 0.1541
Epoch 11/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8900 - loss: 0.2774 - val_accuracy: 0.9875 - val_loss: 0.1559
Epoch 12/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8770 - loss: 0.2821 - val_accuracy: 0.9875 - val_loss: 0.1524
Epoch 13/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8711 - loss: 0.3325 - val_accuracy: 0.9875 - val_loss: 0.1506
Epoch 14/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9048 - loss: 0.2560 - val_accuracy: 0.9875 - val_loss: 0.1516
Epoch 15/15
320/320 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8960 - loss: 0.2415 - val_accuracy: 0.9875 - val_loss: 0.1464
18.50076150894165
```

```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'])
```

```
[<matplotlib.lines.Line2D at 0x7819561569d0>]
```



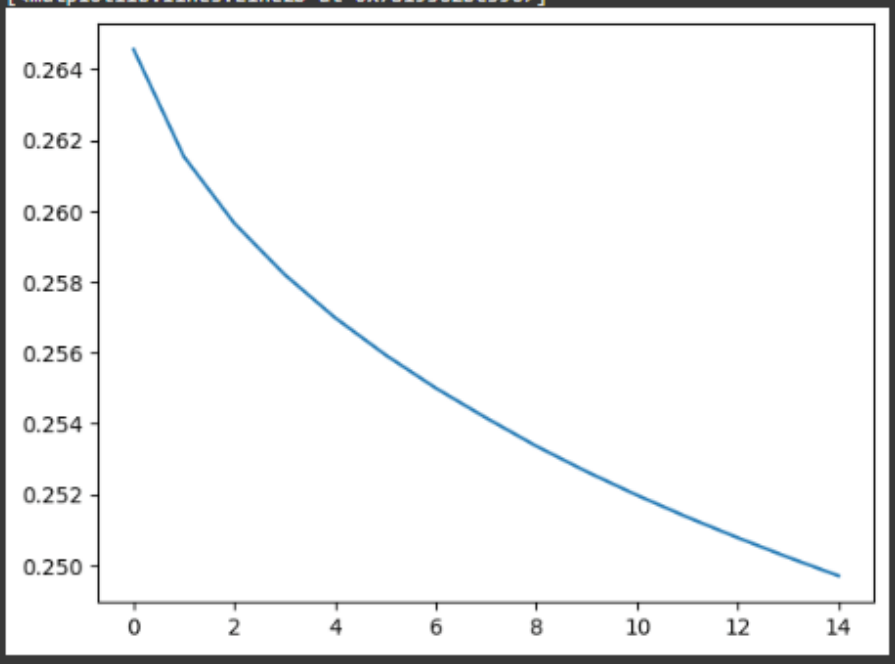
Batch

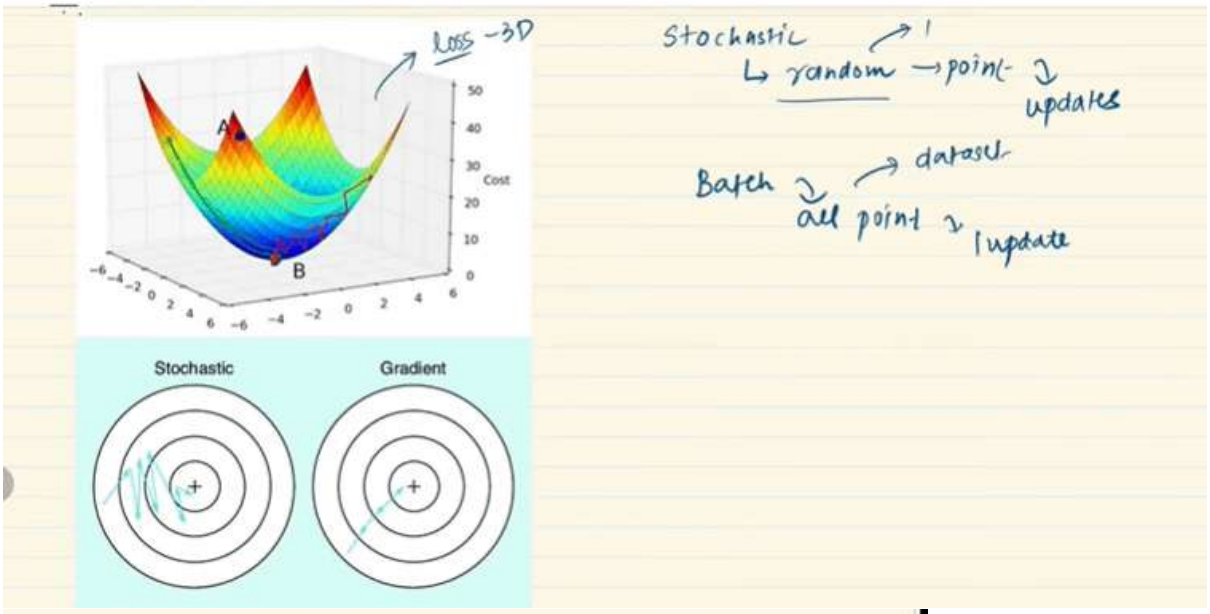
```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
start = time.time()
history = model.fit(X_scaled,y,epochs=15,batch_size=400,validation_split=0.2)
print(time.time() - start)
```

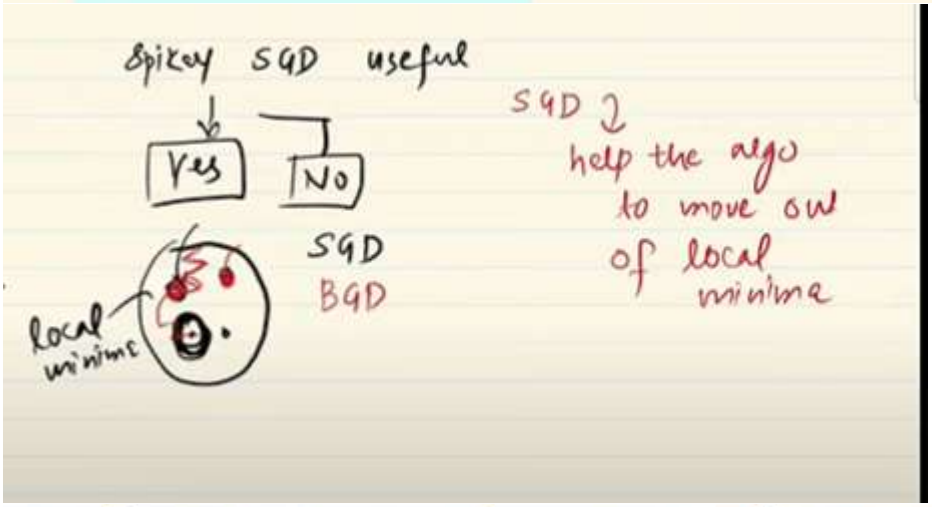
```
Epoch 1/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.8969 - loss: 0.2646 - val_accuracy: 0.9875 - val_loss: 0.1507
Epoch 2/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 102ms/step - accuracy: 0.8969 - loss: 0.2615 - val_accuracy: 0.9875 - val_loss: 0.1537
Epoch 3/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 99ms/step - accuracy: 0.8938 - loss: 0.2597 - val_accuracy: 0.9875 - val_loss: 0.1563
Epoch 4/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step - accuracy: 0.8938 - loss: 0.2582 - val_accuracy: 0.9875 - val_loss: 0.1586
Epoch 5/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 112ms/step - accuracy: 0.8938 - loss: 0.2570 - val_accuracy: 0.9875 - val_loss: 0.1607
Epoch 6/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 127ms/step - accuracy: 0.8938 - loss: 0.2559 - val_accuracy: 0.9875 - val_loss: 0.1626
Epoch 7/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - accuracy: 0.8938 - loss: 0.2550 - val_accuracy: 0.9875 - val_loss: 0.1644
Epoch 8/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 85ms/step - accuracy: 0.8938 - loss: 0.2542 - val_accuracy: 0.9875 - val_loss: 0.1661
Epoch 9/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - accuracy: 0.8969 - loss: 0.2534 - val_accuracy: 0.9875 - val_loss: 0.1678
Epoch 10/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 0.8969 - loss: 0.2526 - val_accuracy: 0.9875 - val_loss: 0.1694
Epoch 11/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step - accuracy: 0.8969 - loss: 0.2520 - val_accuracy: 0.9875 - val_loss: 0.1709
Epoch 12/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 140ms/step - accuracy: 0.8969 - loss: 0.2514 - val_accuracy: 0.9875 - val_loss: 0.1723
Epoch 13/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - accuracy: 0.8938 - loss: 0.2508 - val_accuracy: 0.9875 - val_loss: 0.1737
Epoch 14/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 141ms/step - accuracy: 0.8938 - loss: 0.2502 - val_accuracy: 0.9875 - val_loss: 0.1751
Epoch 15/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 140ms/step - accuracy: 0.8938 - loss: 0.2497 - val_accuracy: 0.9875 - val_loss: 0.1764
3.028410434722900 4
3.028410434722900 4
```

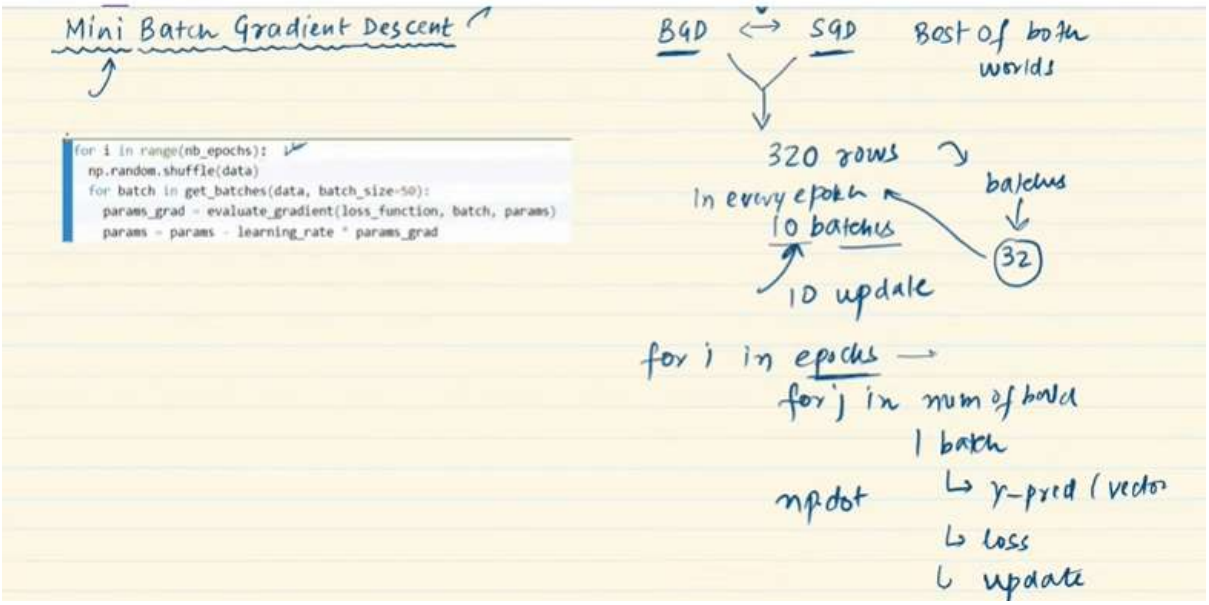
```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'])
```

```
[<matplotlib.lines.Line2D at 0x7819562bc390>]
```









Let’s say you have:

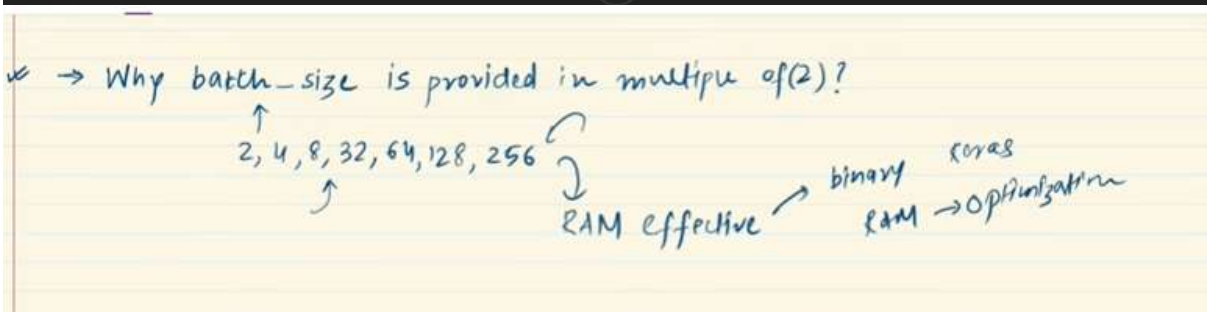
- A **dataset with 1000 samples**
- A **batch size = 50**

Step-by-step:

1. **Initialize weights and biases**.
2. **Epoch 1 starts**:
    - Shuffle the data (optional but common).
    - Split it into mini-batches of size 50 → you'll get `1000 / 50 = 20 mini-batches`.
    - For **each mini-batch** (i.e., 50 samples at a time):
        - Make predictions using current weights and biases.
        - Calculate the loss and gradients.
        - **Update weights and biases ONCE using that mini-batch**.
    - After all 20 mini-batches, epoch 1 ends

✅ Comparison Table

Aspect	Batch GD	Stochastic GD (SGD)	Mini-Batch GD
Time per Epoch	🔴 Slowest (processes all data once)	🟢 Fastest (1 sample = 1 update)	🟡 Moderate (processes batch size)
Updates per Epoch	1	= Number of samples	= Number of samples / batch size
Convergence Speed	🔴 Slow (fewer updates)	🟢 Fast per update (high frequency)	🟡 Balanced
Convergence Stability	🟢 Stable (smooth updates)	🔴 Unstable (noisy, high variance)	🟡 Moderate stability
Final Accuracy	🟢 Often good	🟡 May fluctuate, may not settle well	🟢 Often good and practical
Memory Usage	🔴 High (entire dataset in memory)	🟢 Low (1 sample at a time)	🟡 Medium (batch size in memory)
GPU Efficiency	🔴 Poor	🔴 Poor	🟢 Excellent (vectorized operations)
Practical Use	Rare in large data	Rare in pure form	🟢 Standard in practice



Vanishing Gradient Problem

Thursday, April 7, 2022 7:45 AM

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 weights receives 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, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. ~~As one example of the problem cause, traditional activation~~

1) 0.1 x 0.1 x 0.1 x 0.1 = 0.0001 → VGP

2) Deep NN →

3) Sigmoid / tanh → Af →

How to recognize?

1) Loss focus → epoch → no changes → VGP

2) weights → graph value → Tensorboard ↳ callbacks

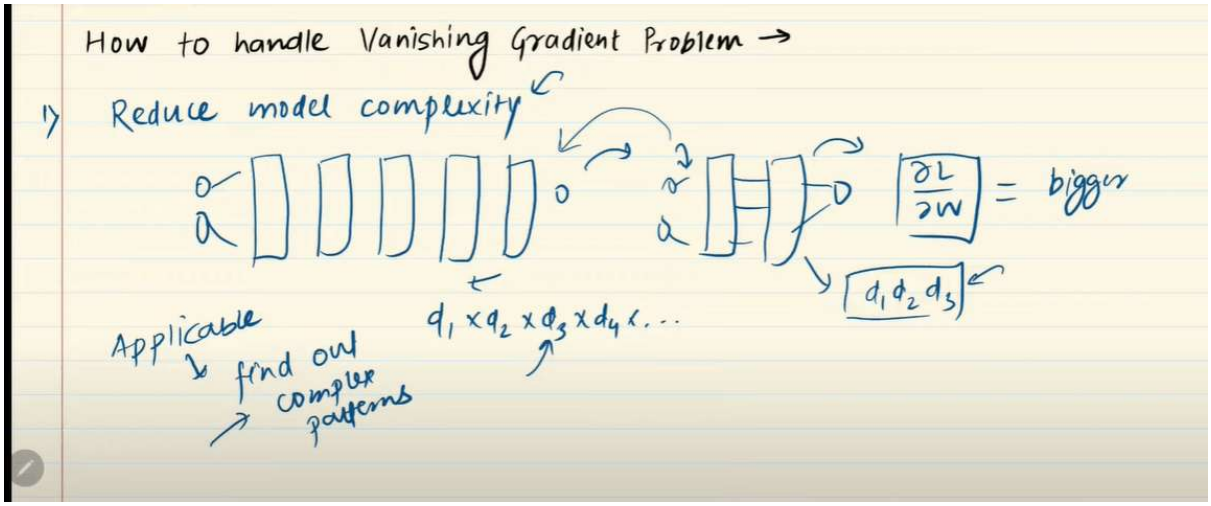
Keras

loss after epoch

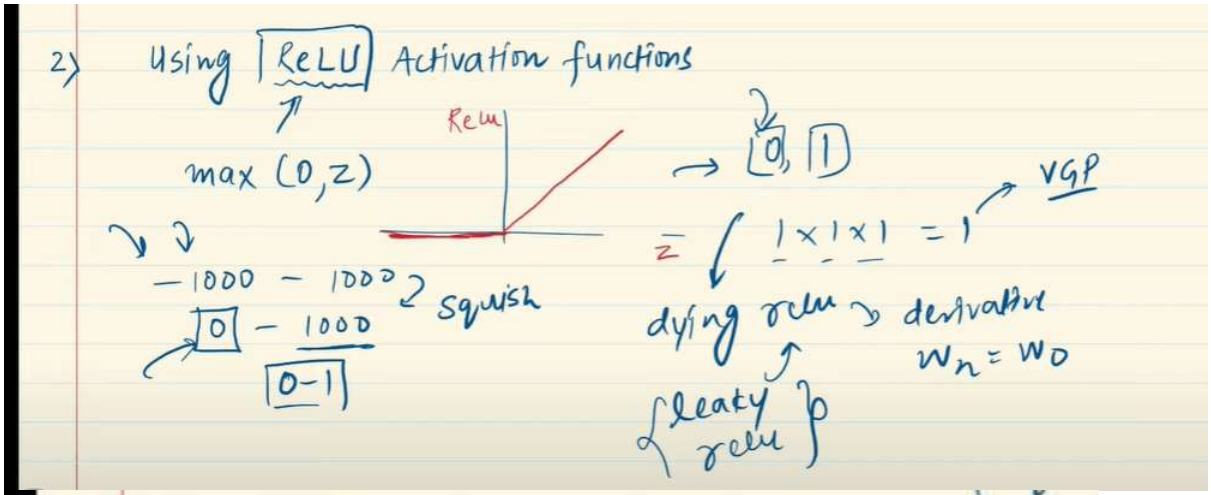
epoch

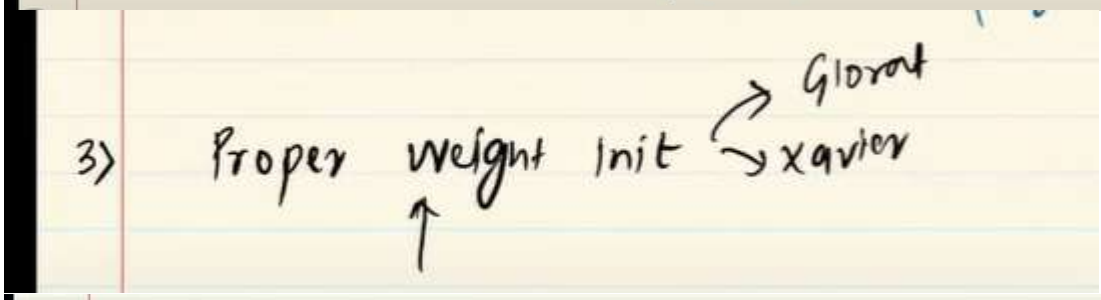
```
[75] old_weights

array([[ 0.15133941, -0.5475049 ,  0.20439553, -0.5383658 ,  0.01435351,
         0.6393233 , -0.4457206 , -0.7024723 ,  0.7043677 ,  0.26854134],
       [ 0.19128346,  0.2853908 , -0.42532492,  0.23361963,  0.48123974,
        -0.3359478 , -0.44716895, -0.57663697,  0.2556631 , -0.6959247 ]],
      dtype=float32)

new_weights

array([[ 0.15132847, -0.5475164 ,  0.20438164, -0.5383458 ,  0.01440594,
         0.639327  , -0.4457354 , -0.7025074 ,  0.7043448 ,  0.26854628],
       [ 0.19130446,  0.28541353, -0.42529657,  0.23357487,  0.48114753,
        -0.3359549 , -0.44714212, -0.5765698 ,  0.25571352, -0.6959336 ]],
      dtype=float32)
```

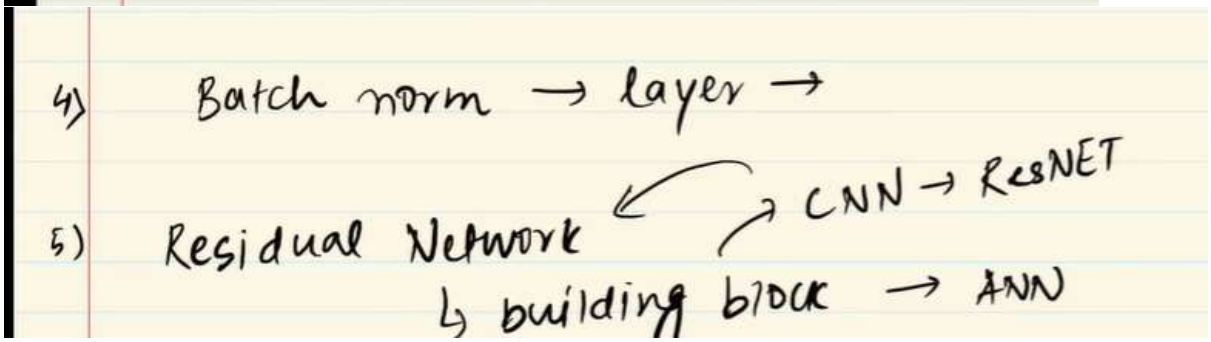
we can see after epochs there is no change in weights --- almost negligible

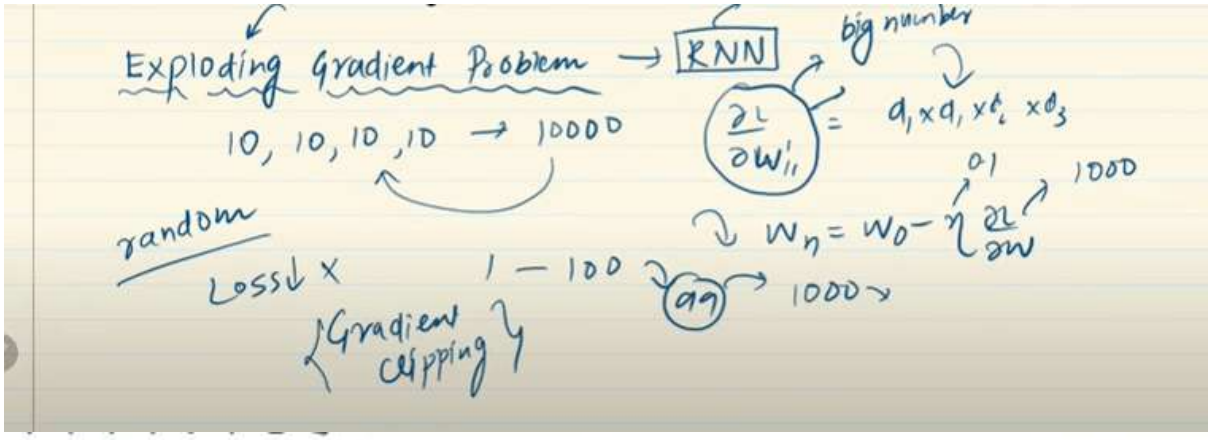


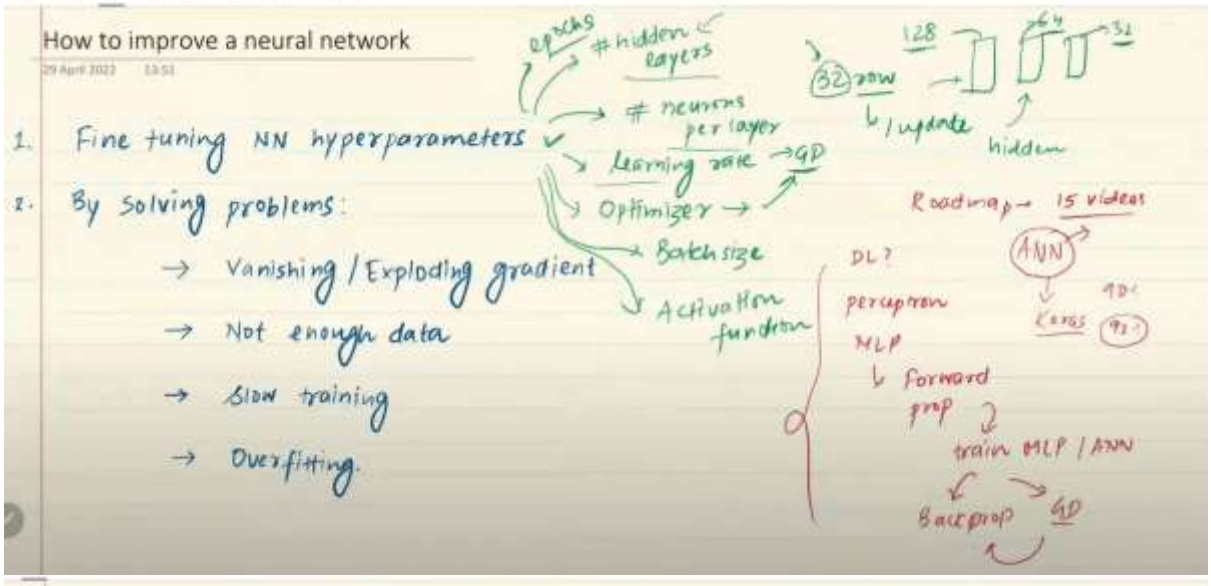
not applicable bcz nn is used for complex patterns and reducing layers would be wrong

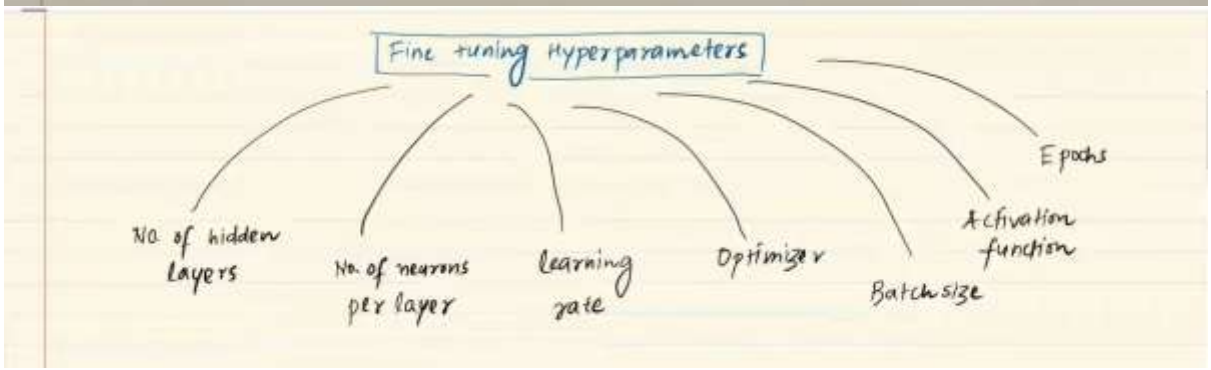








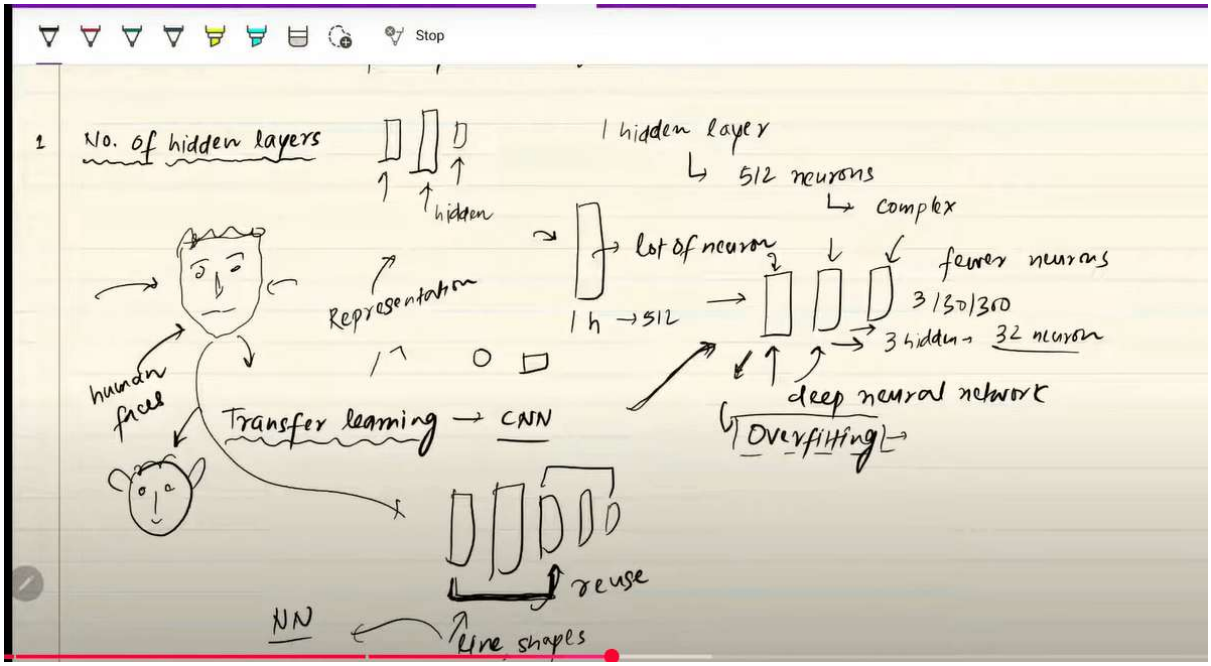




1. no of hidden layers

prefer increasing layers rather then adding more neurons in one layer

keep increasing layers till overfitting error does not occur

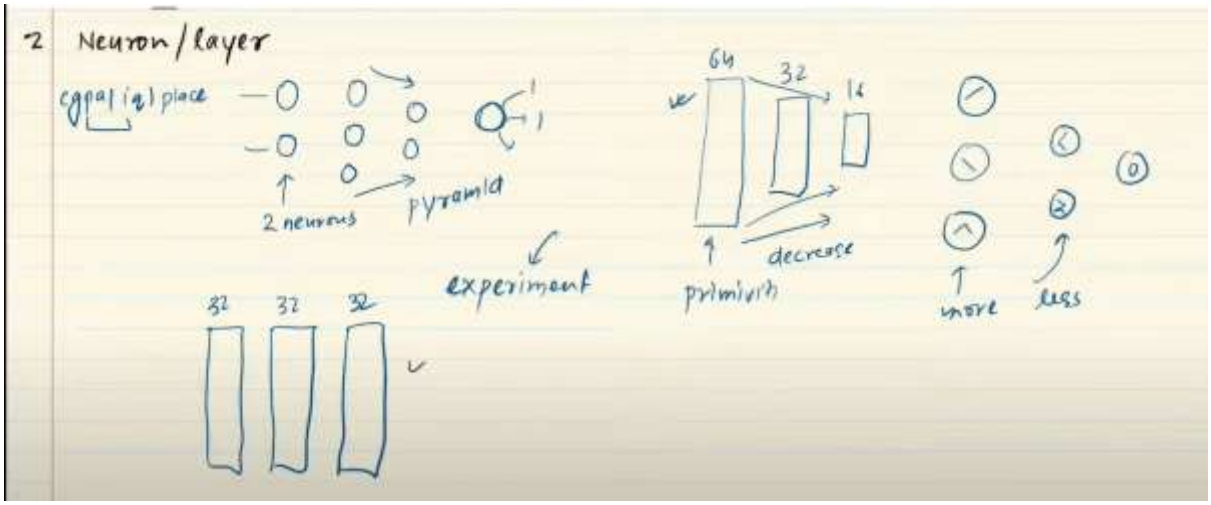


2 . NO of neurons

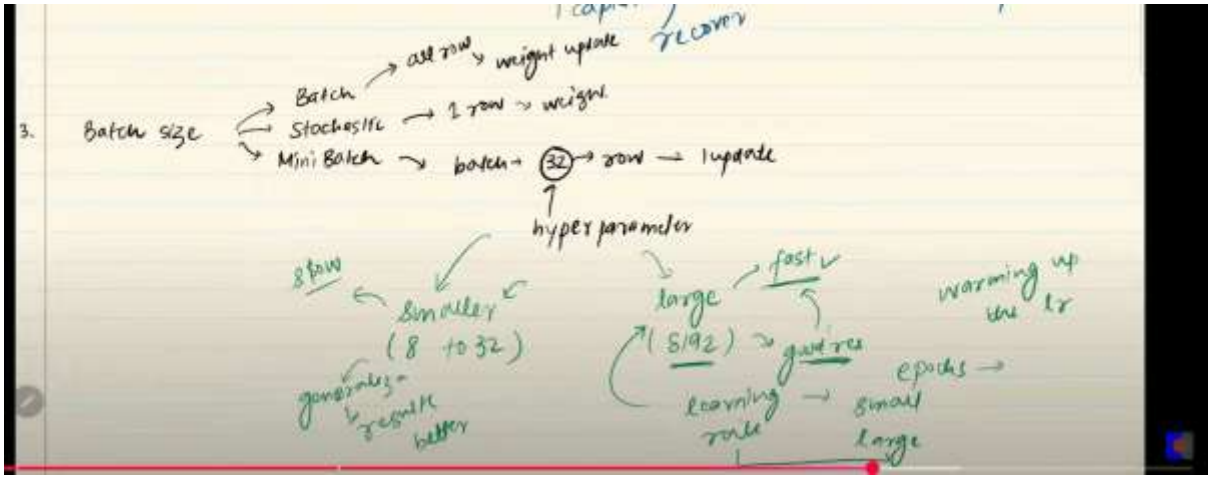
input layer => no. of neurons =no. of inputs

output layer => type of output

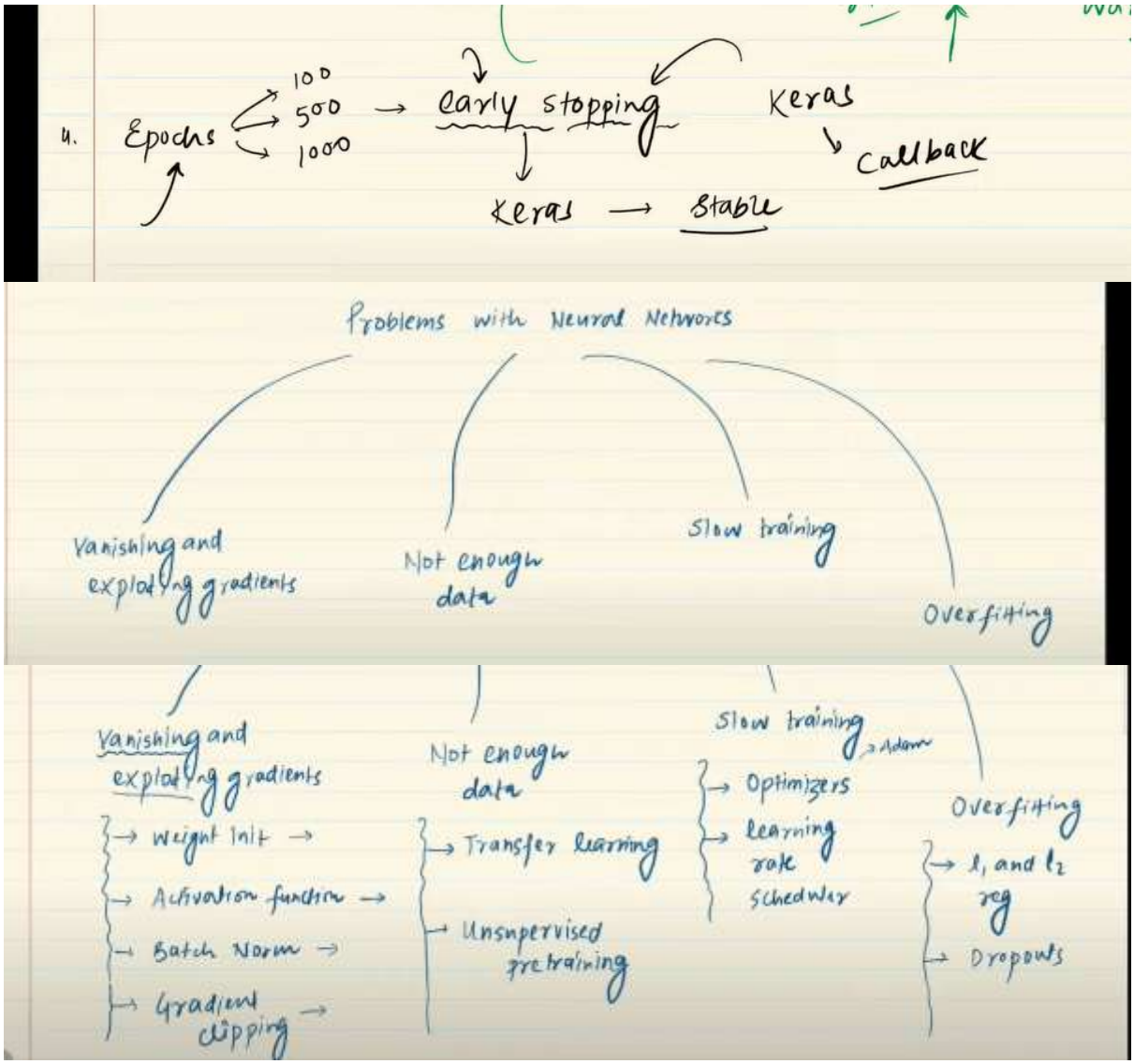
- regression - 1
- classfication - 1
- multiclass classification - depends on no. of classes to be guessed

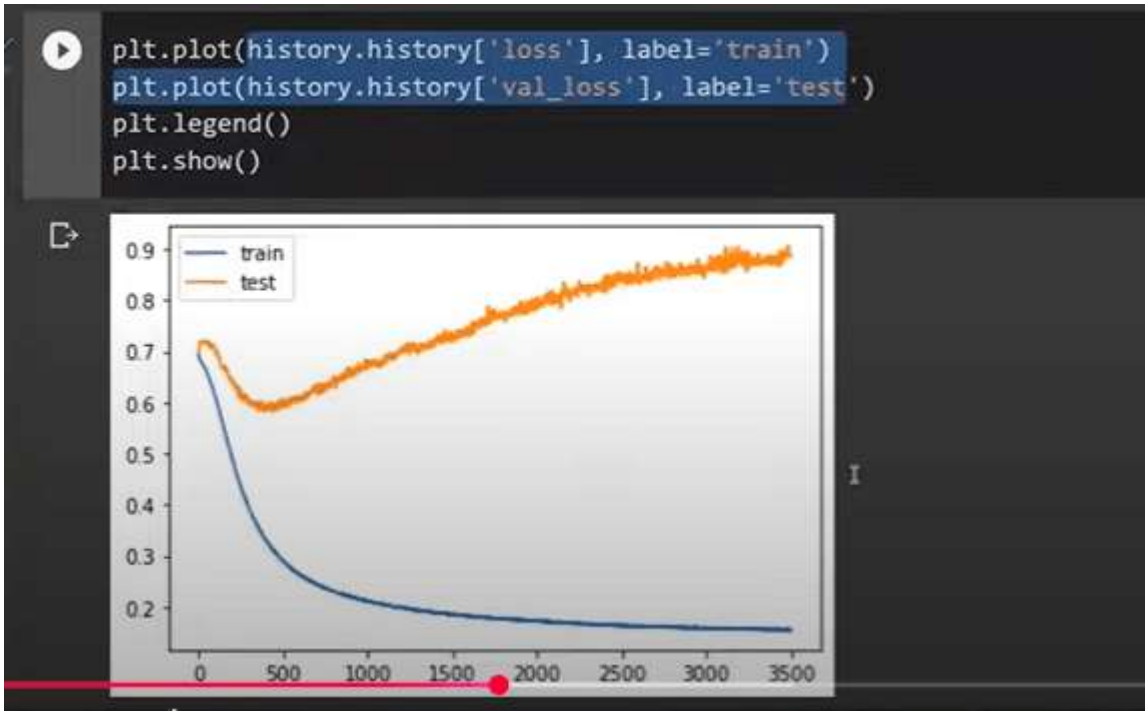


3 . batch size


3.
Batch size
Batch
Stochastic
Mini Batch
all row
weight update
1 row
weight
batch
32
row
update
hyper parameter
slow
smaller
(8 to 32)
large
(64 to 512)
fast
warming up the lr
epochs
learning rate
small
large

4 . epochs


4.
Epochs
100
500
1000
early stopping
Keras
callback
Keras → Stable
Problems with Neural Networks
Vanishing and exploding gradients
Not enough data
Slow training
Overfitting
Vanishing and exploding gradients
Weight Init
Activation function
Batch Norm
Gradient clipping
Not enough data
Transfer learning
Unsupervised pretraining
Slow training
Adam
Optimizers
learning rate schedular
Overfitting
l_1 and l_2 reg
Dropouts



- 🔹 **Blue Curve – `train` (Training Loss)**
  - Plotted using `history.history['loss']`
  - Shows how well the model is fitting the **training data**
  - **Decreases over time**, which is expected as the model learns
- 🔸 **Orange Curve – `test` (Validation Loss)**
  - Plotted using `history.history['val_loss']`
  - Shows how well the model performs on **unseen validation data**
  - **Initially decreases**, then **starts increasing** — this is a classic sign of **overfitting**

keras.io/api/callbacks/early_stopping/

Early Stopping

```
[24]  model = Sequential()

      model.add(Dense(256, input_dim=2, activation='relu'))
      model.add(Dense(1, activation='sigmoid'))
```

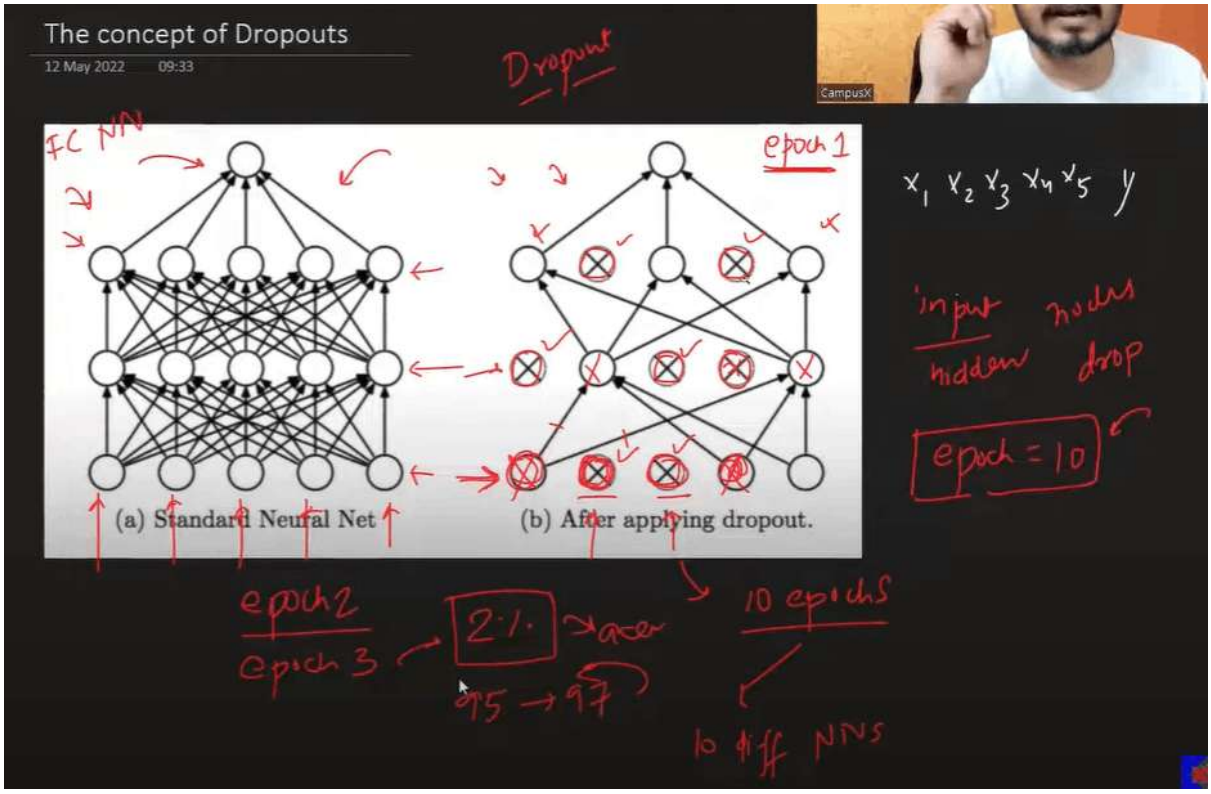
```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

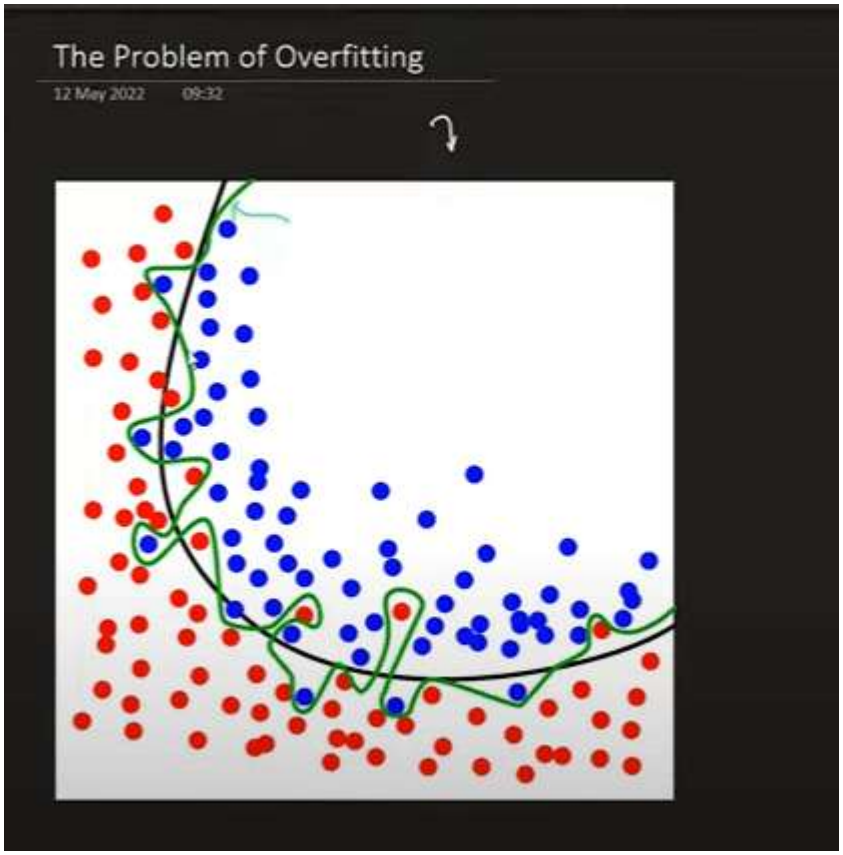
```
[25]  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
callback = EarlyStopping(
    monitor="val_loss", #Quantity to be monitored. Defaults to "val_loss".
    min_delta=0.00001, #Minimum change in monitored metric to qualify as an actual improvement.
    patience=20, #Number of epochs with no improvement before training is stopped.
    verbose=1, # Controls whether to display early stopping messages (0 = silent, 1 = show).
    mode="auto",#Defines improvement direction — 'min' for decreasing, 'max' for increasing, 'auto'
    infers from metric name.
    baseline=None,#: Minimum baseline value; training stops if no improvement beyond it is achieved
    restore_best_weights=True #If True, restores model weights from the epoch with best monitored
    value.
)
```

```
history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3500,
callbacks=callback)
```

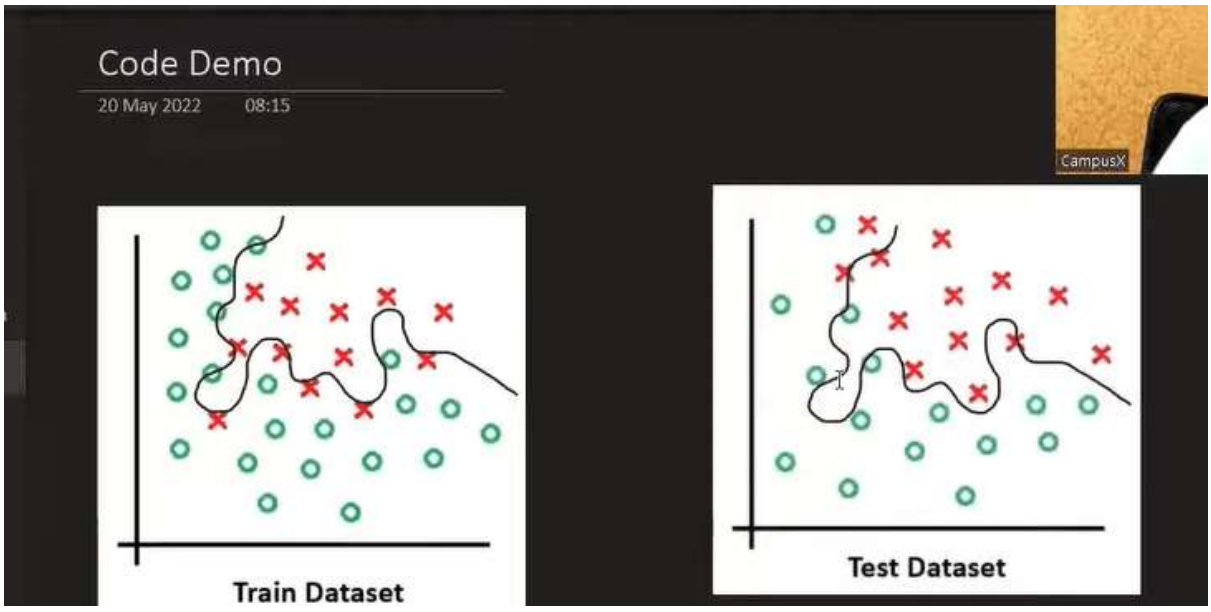
OVERFITTING

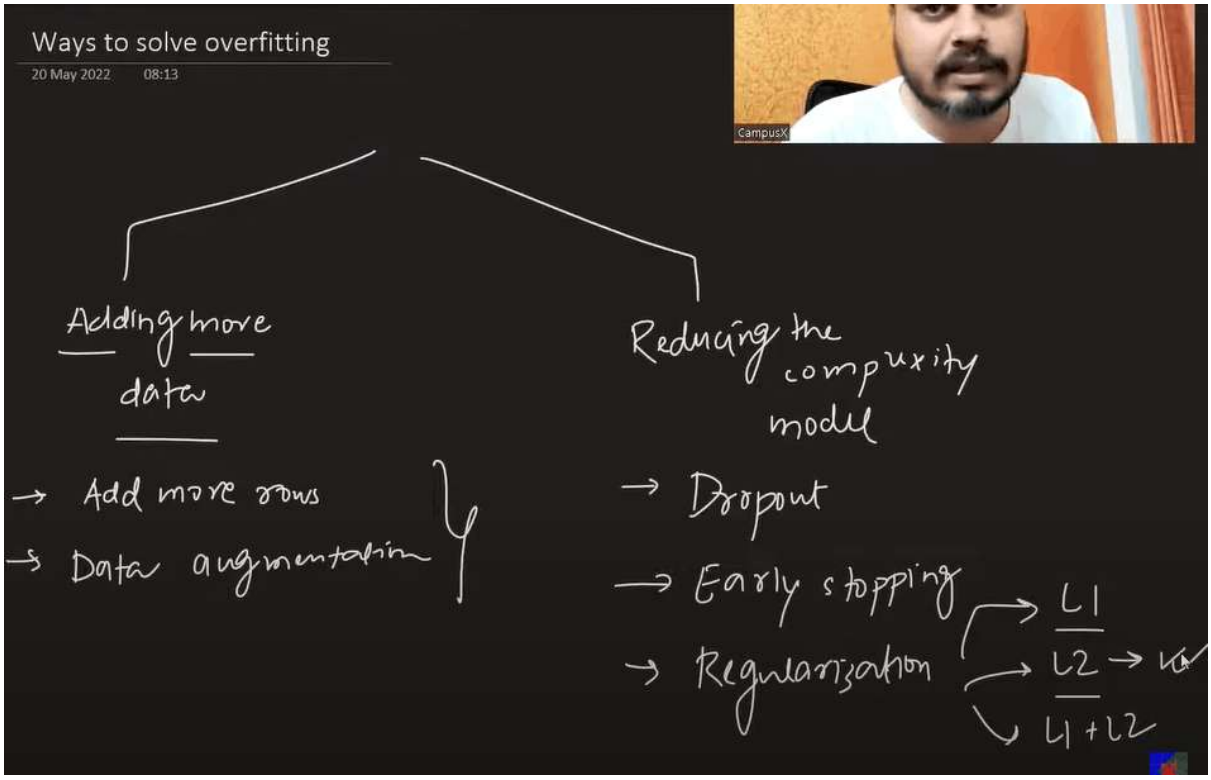




here green NN is trying to memorize and is very specific from training data perspective and performs bad with testing data

whereas the black NN is perfect

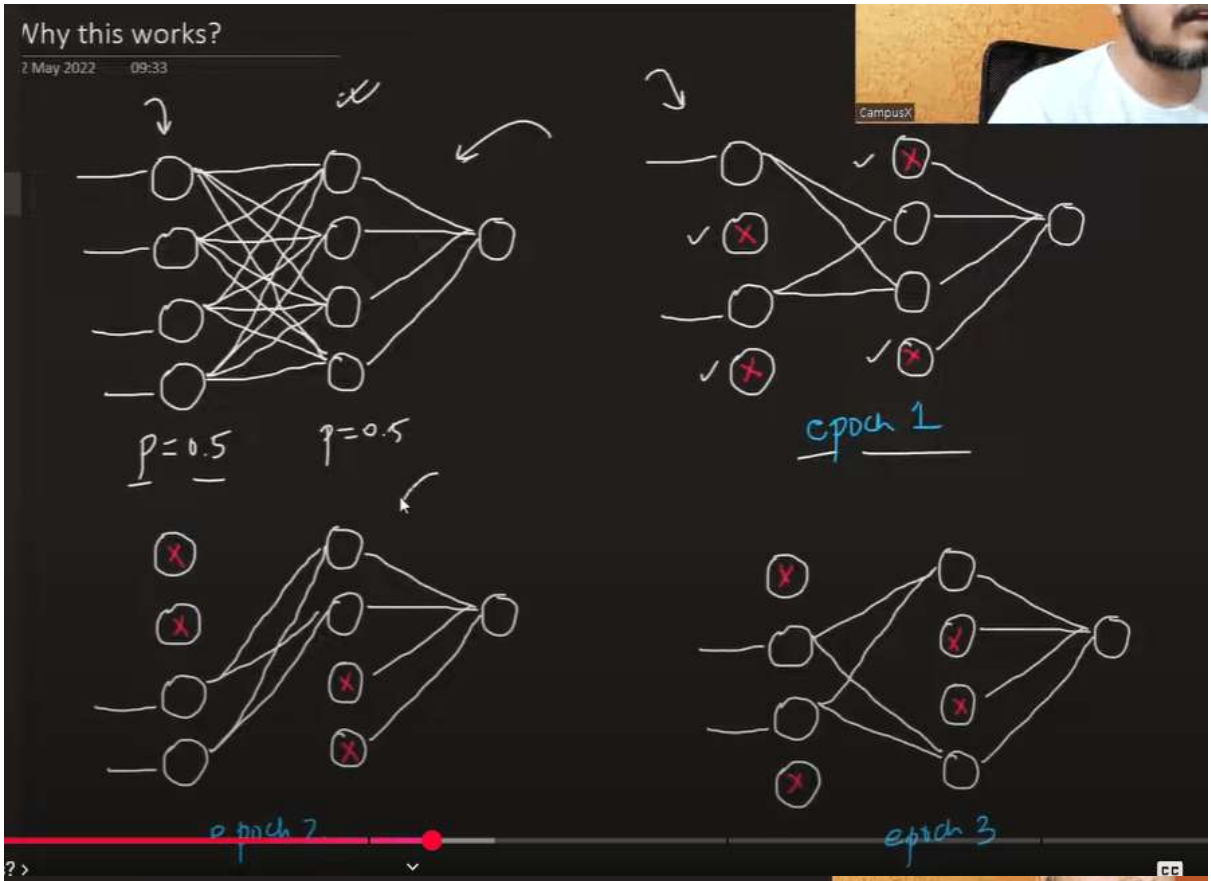


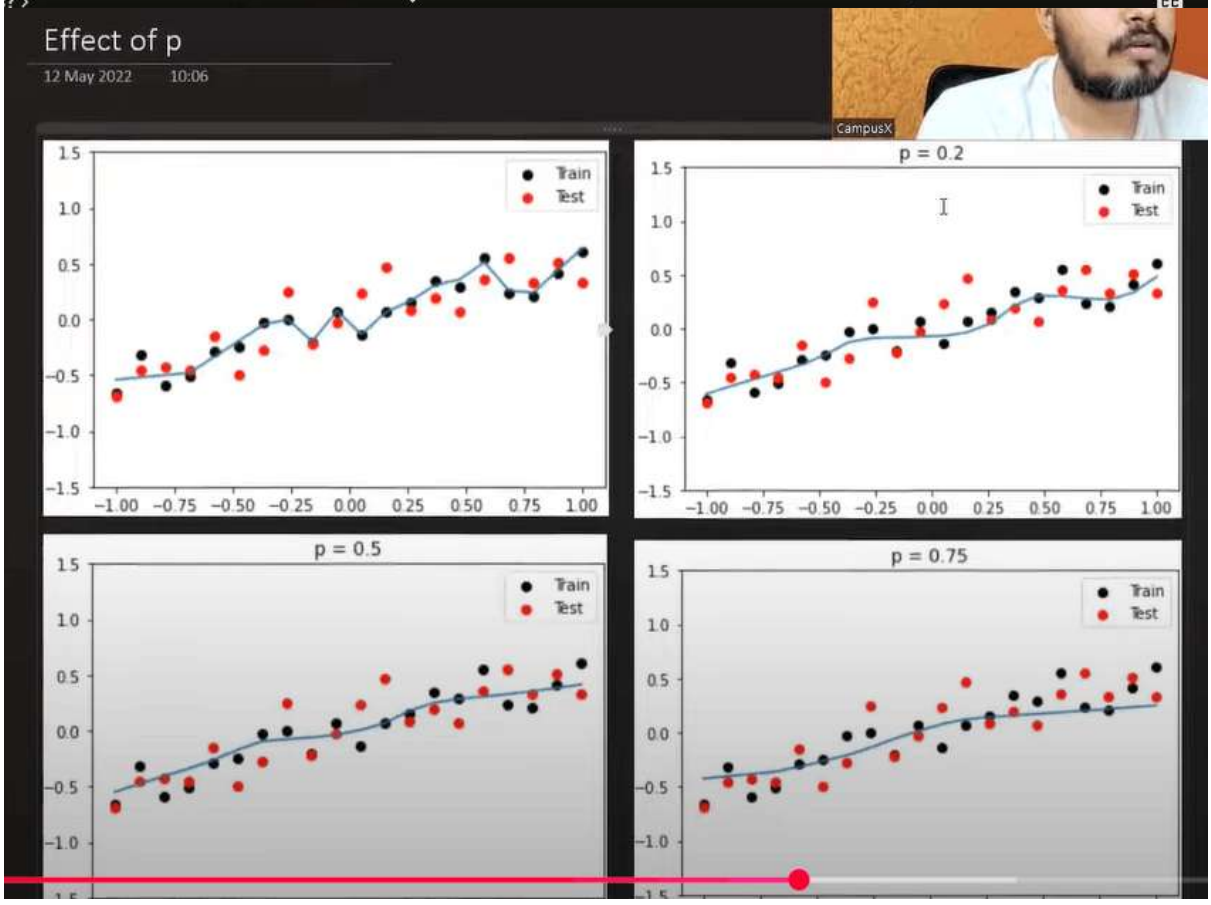


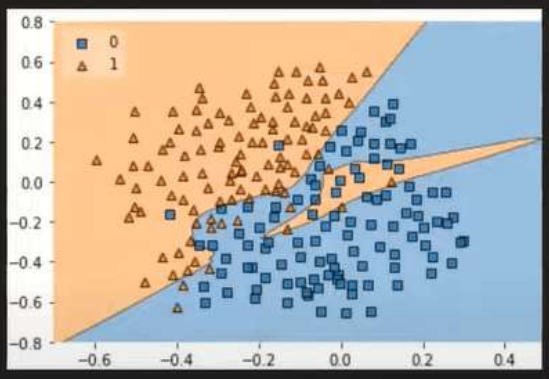
DROPOUTS

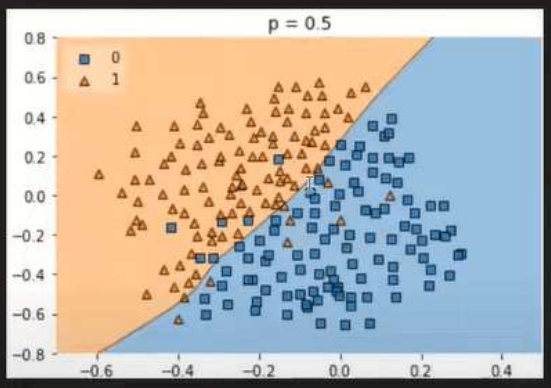
10 epoch - 10 diff NN

randomly few neurons are shut down


Why this works?
2 May 2022 09:33
CampusX
p=0.5
p=0.5
epoch 1
epoch 2
epoch 3


Effect of p
12 May 2022 10:06
CampusX
Train
Test
p = 0.2
p = 0.5
p = 0.75



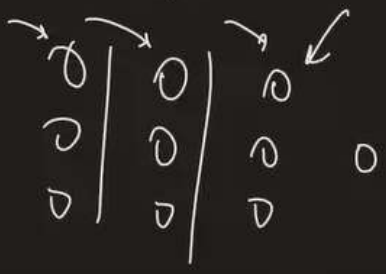


Practical Tips and Tricks

12 May 2022 09:34

1) Overfitting P↑, underfitting P↓

2) last layer → dropout

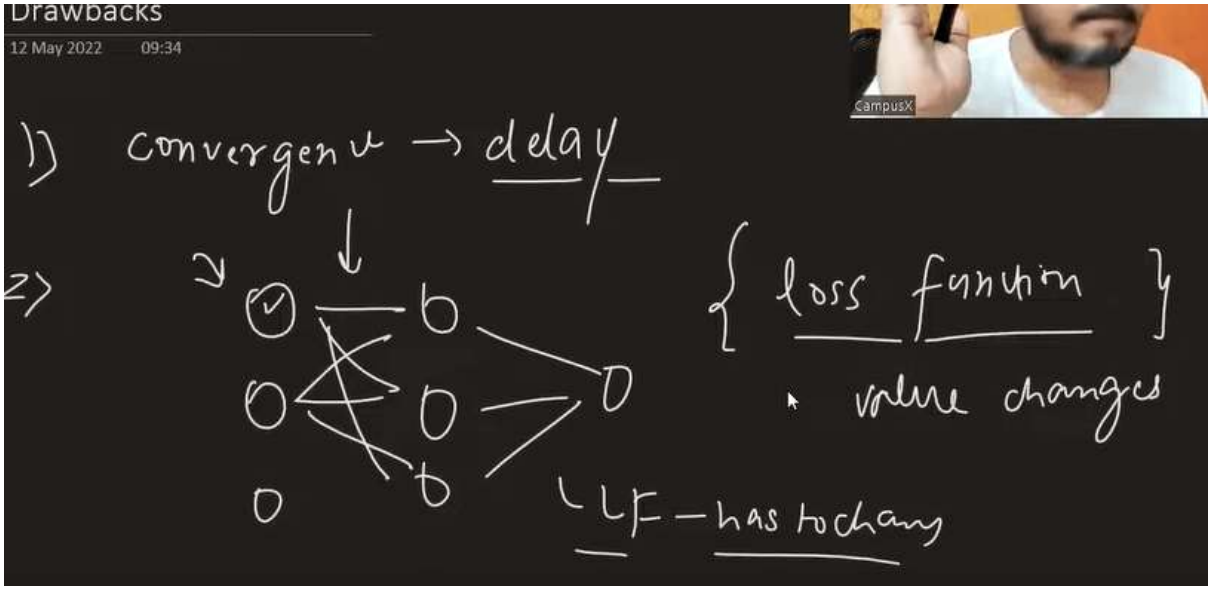


3) CNN → 40-50 · (P) → ✓

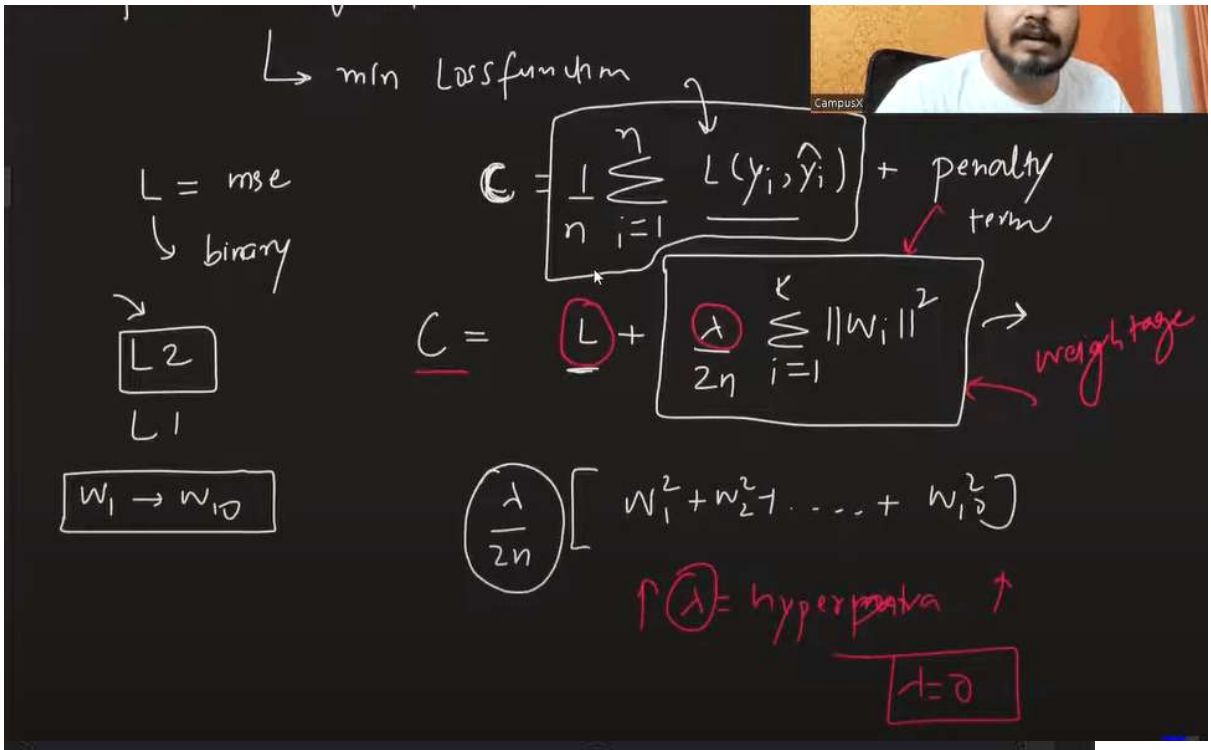
20-30 RNN

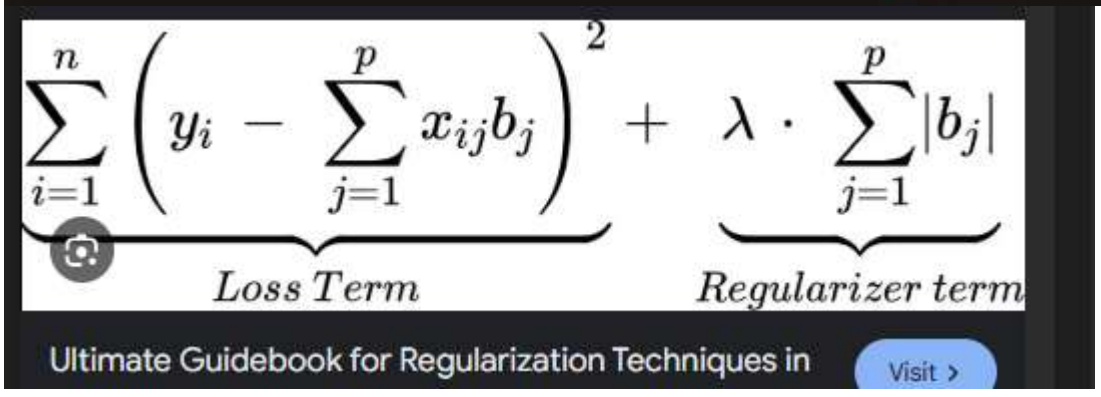
50>

ANN → 10-50

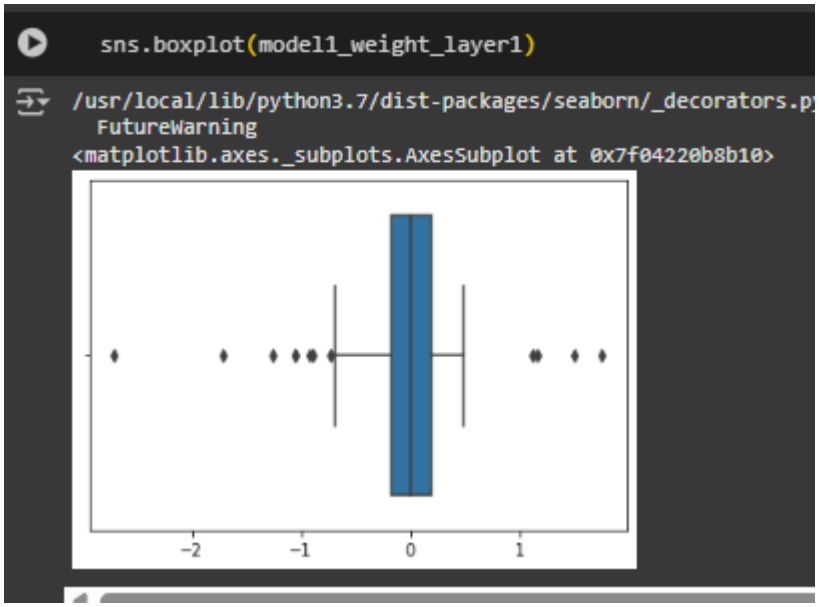


L1 L2 regularization





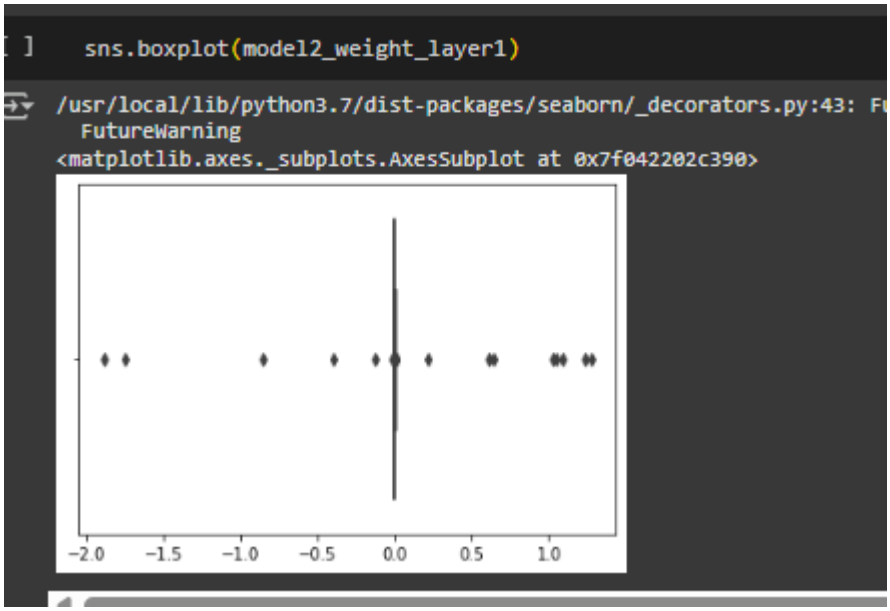
without regulaizaton model 1



we can see weights are mainly between -0.2 and 0.2 with outliers till -2.5

whereas regularization will decay weights and bring them smaller

so in model2 all weights are in between 0 range and outliers go till -2.0 max



```
[48] model1_weight_layer1.min()
     -2.7185578

     model2_weight_layer1.min()
     -0.57482177
```

ACTIVATION FUNCTION

activation function is a gate which tell whether any input given to a neuron will pass or not

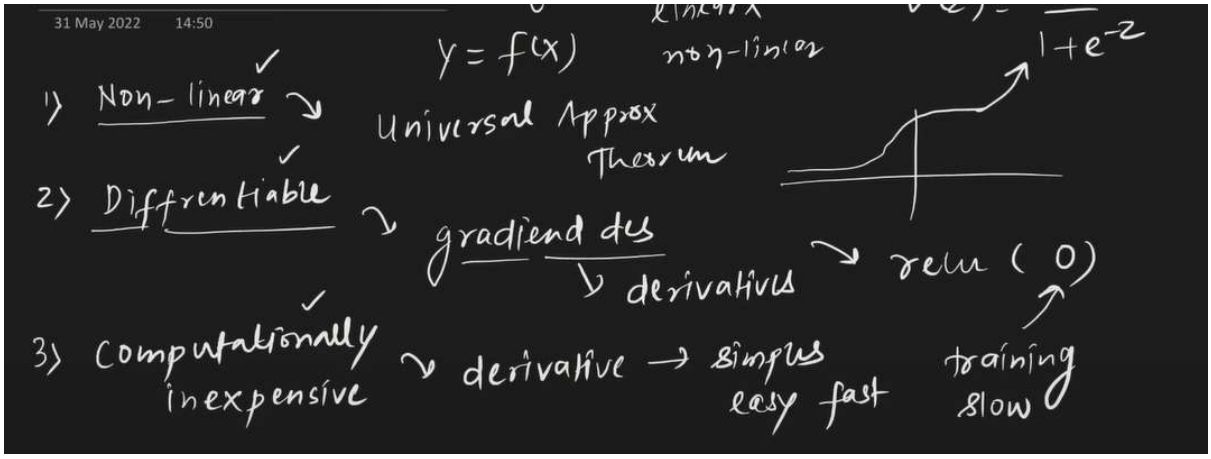
and if it passes

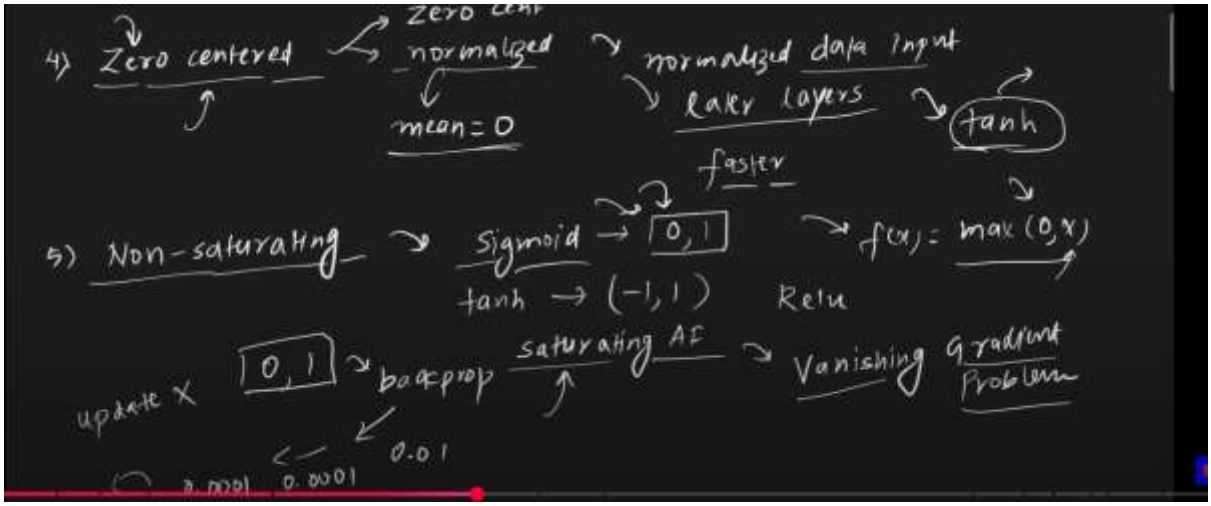
it decides

- whether a neuron will be activated or not
- if activated then how much activated will it be

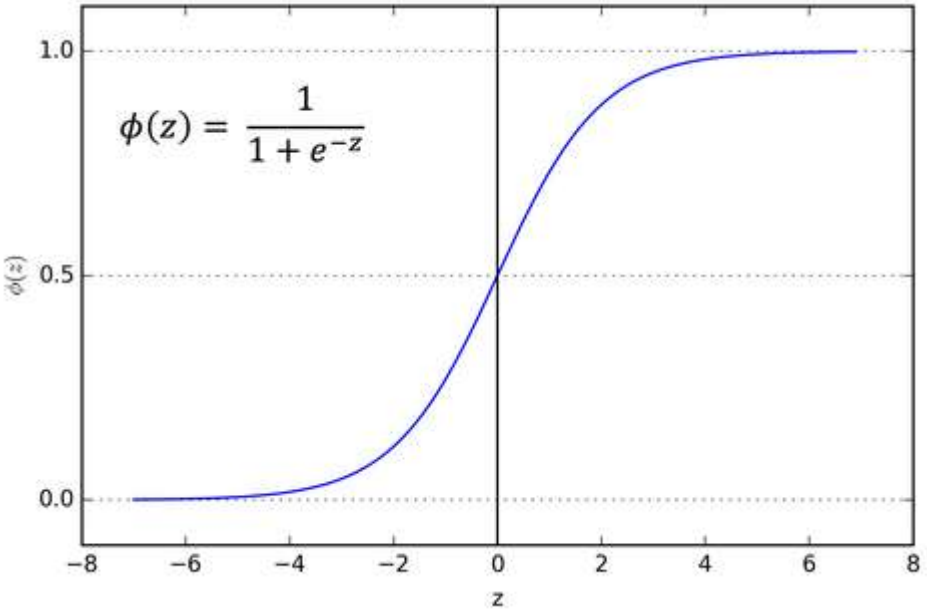
without activation function ANN will work like linear regg and will not catch no linearity

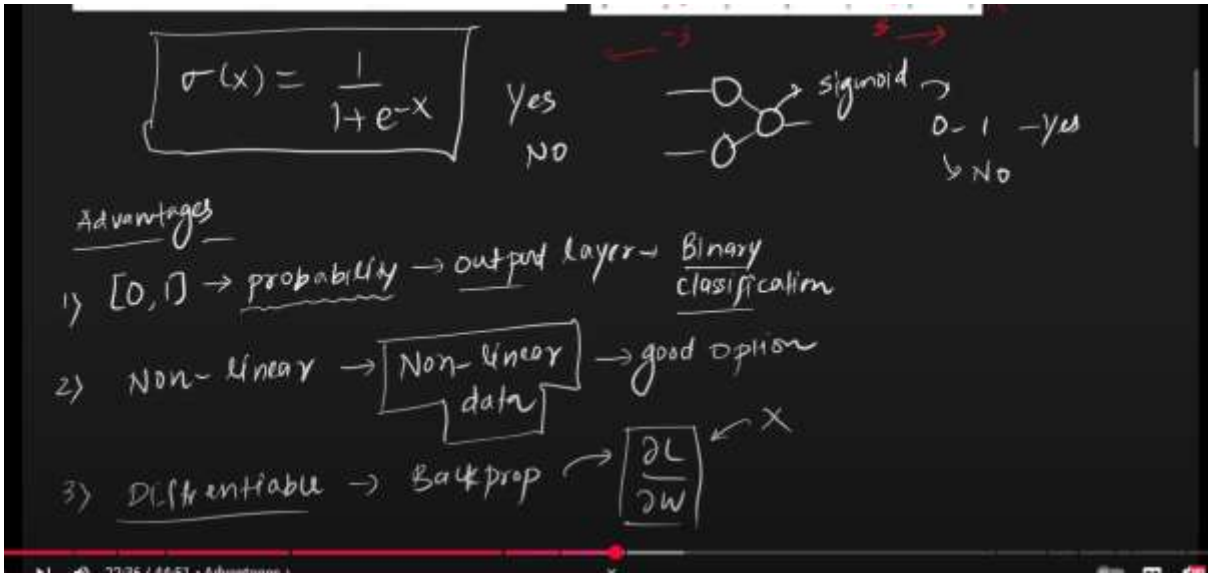
qualities of a activation func




4) Zero centered
Zero
normalized
mean = 0
normalized data input
later layers
tanh
faster
5) Non-saturating
Sigmoid → [0,1]
tanh → (-1,1)
f(x) = max(0,x)
Relu
update X
[0,1]
backprop
saturating AF
Vanishing Gradient Problem
0.01
0.001
0.0001

sigmoid

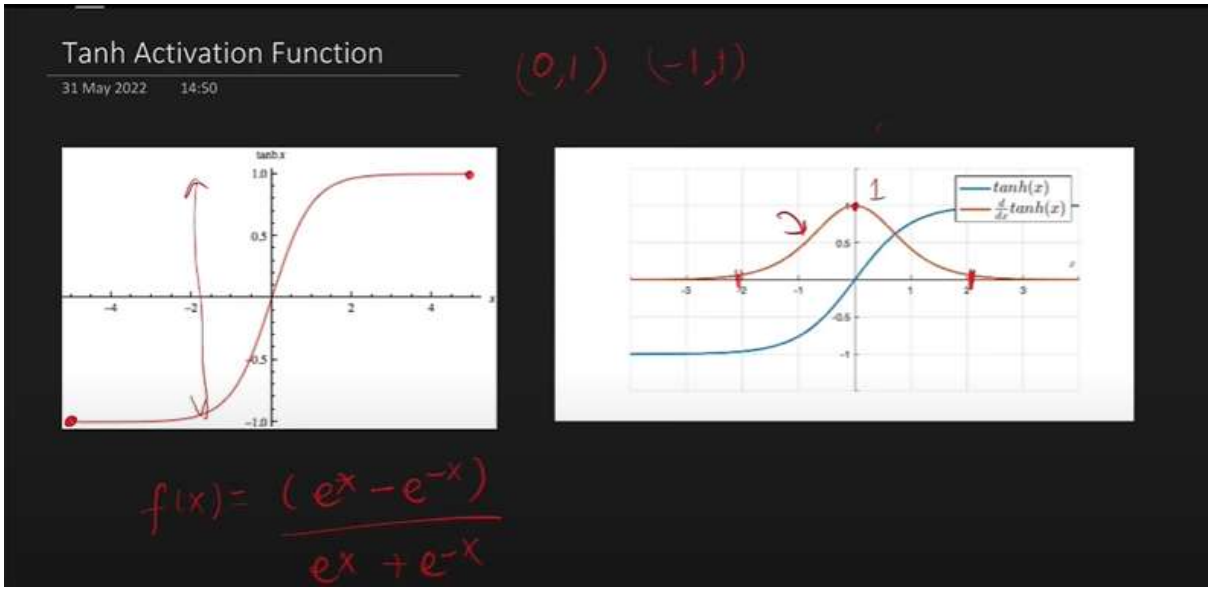

$\phi(z) = \frac{1}{1+e^{-z}}$
1.0
0.5
0.0
$\phi(z)$
−8
−6
−4
−2
0
2
4
6
8
z

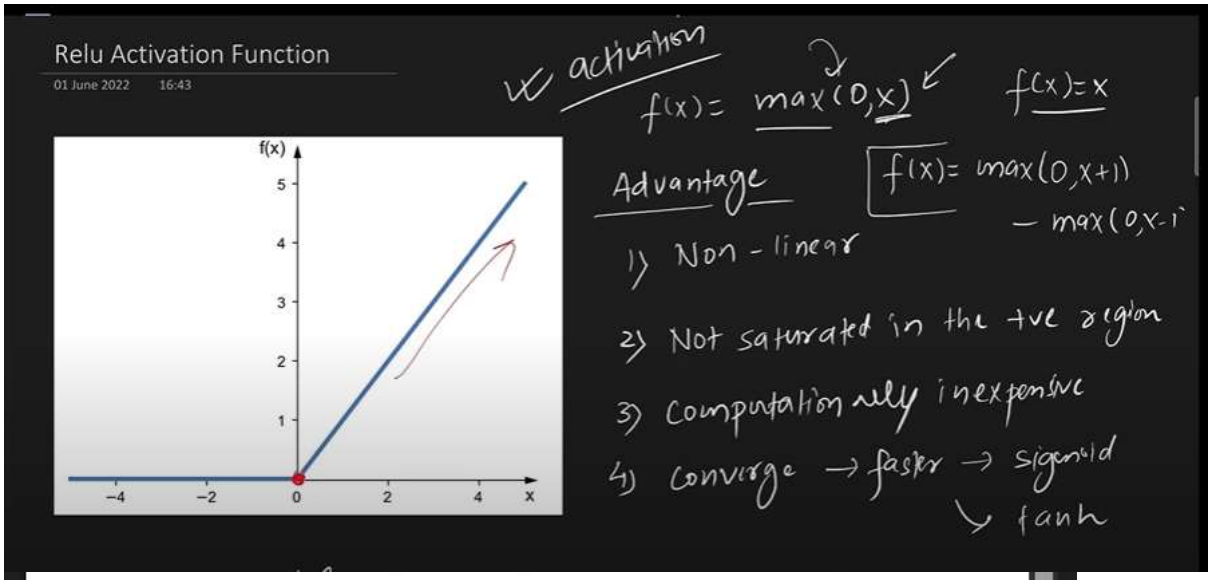


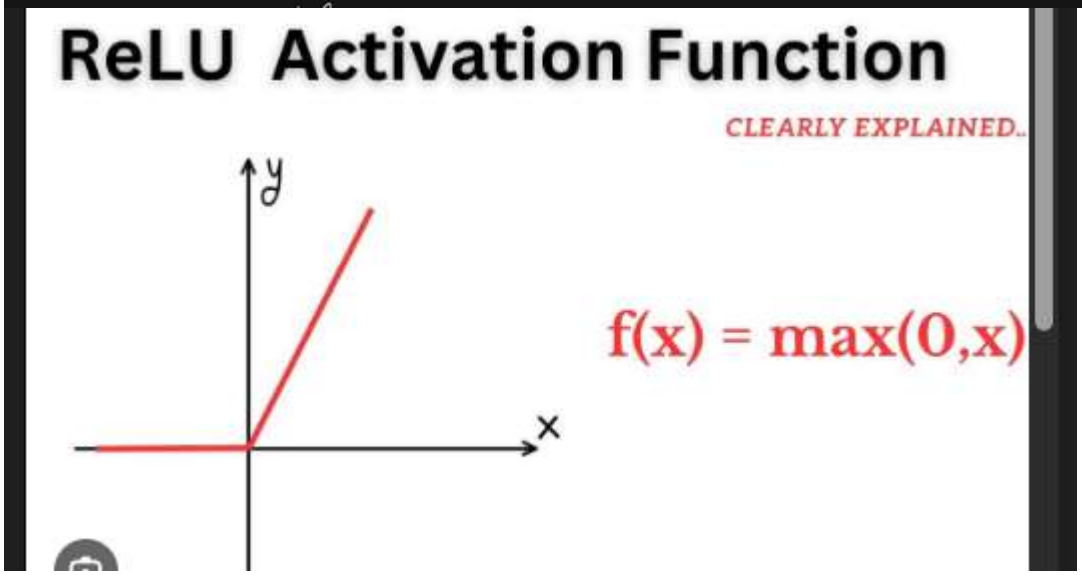
sigmoid is not used in hidden layers

used in output layer in case of binary classification problem

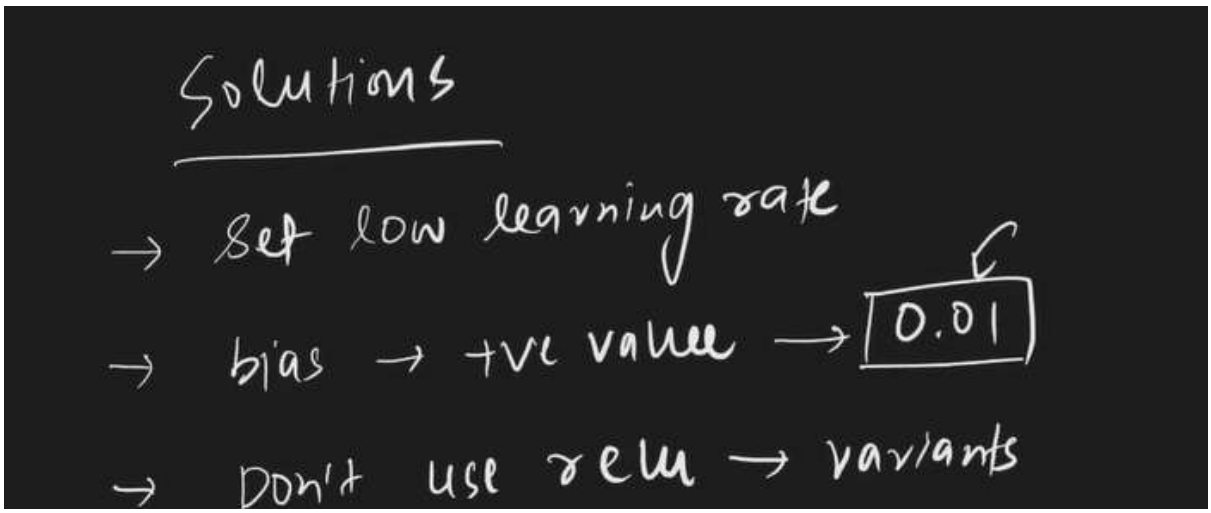
Tanh

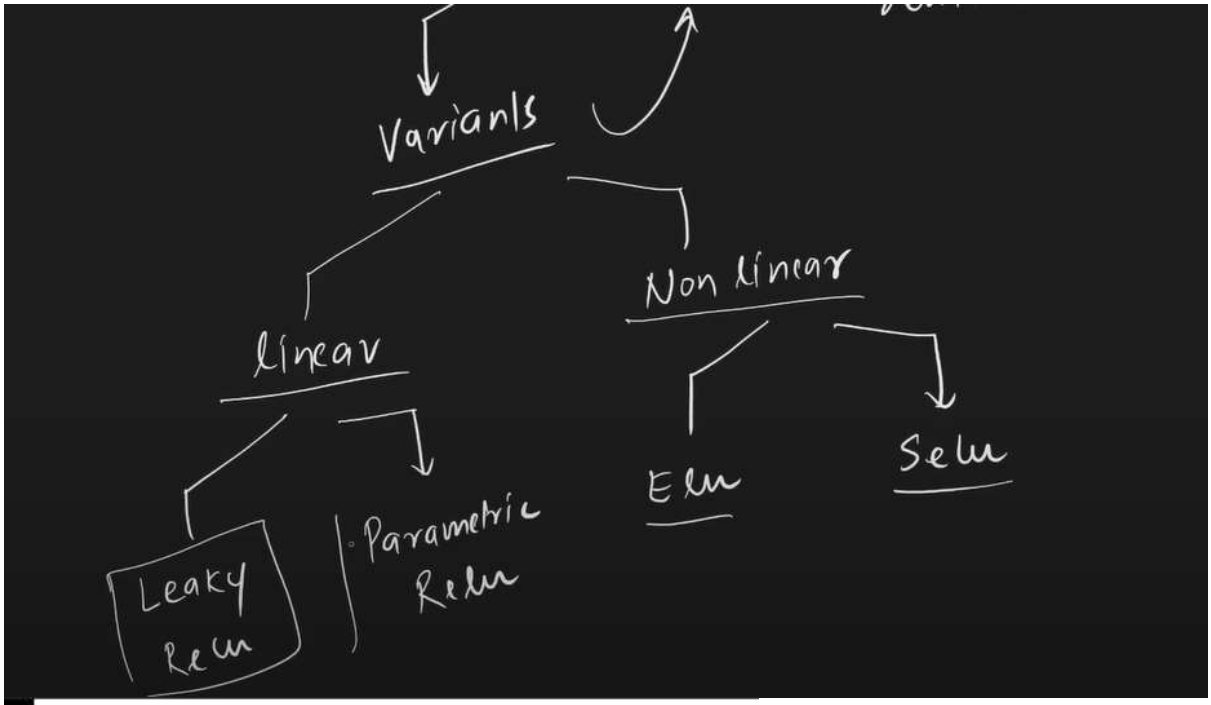



Relu Activation Function
01 June 2022 16:43
f(x)
5
4
3
2
1
−4
−2
0
2
4
x
activation
f(x)= max(0,x)
f(x)=x
Advantage
f(x)= max(0,x+1)
− max(0,x-1)
1) Non - linear
2) Not saturated in the +ve region
3) Computationally inexpensive
4) Converge → faster → sigmoid
↳ tanh

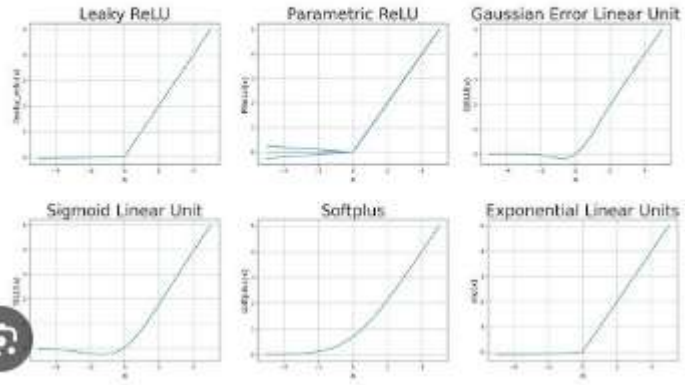

ReLU Activation Function
CLEARLY EXPLAINED..
y
x
f(x) = max(0,x)

dying relu


Solutions
→ Set low learning rate
→ bias → +ve value → 0.01
→ Don't use relu → variants

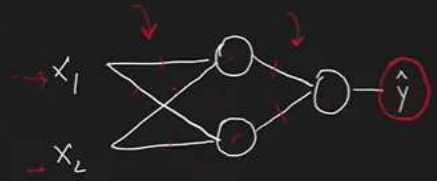


ReLU Variants



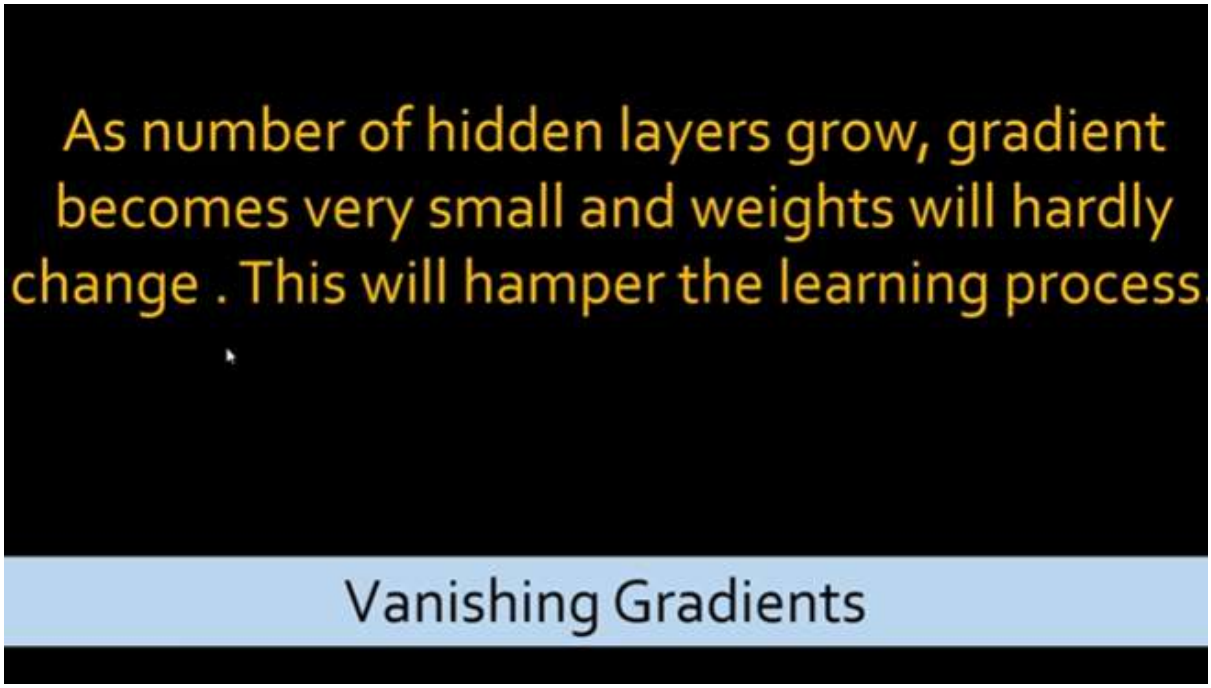
Why Weight Initialization is Important?

22 June 2022 12:48

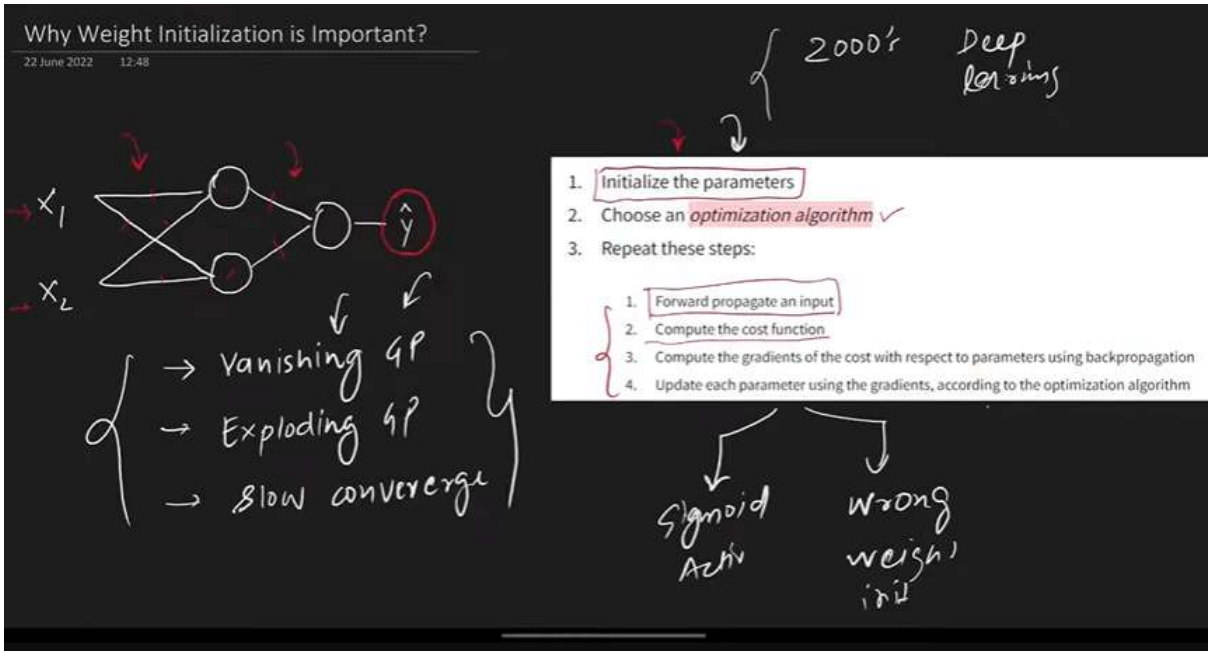


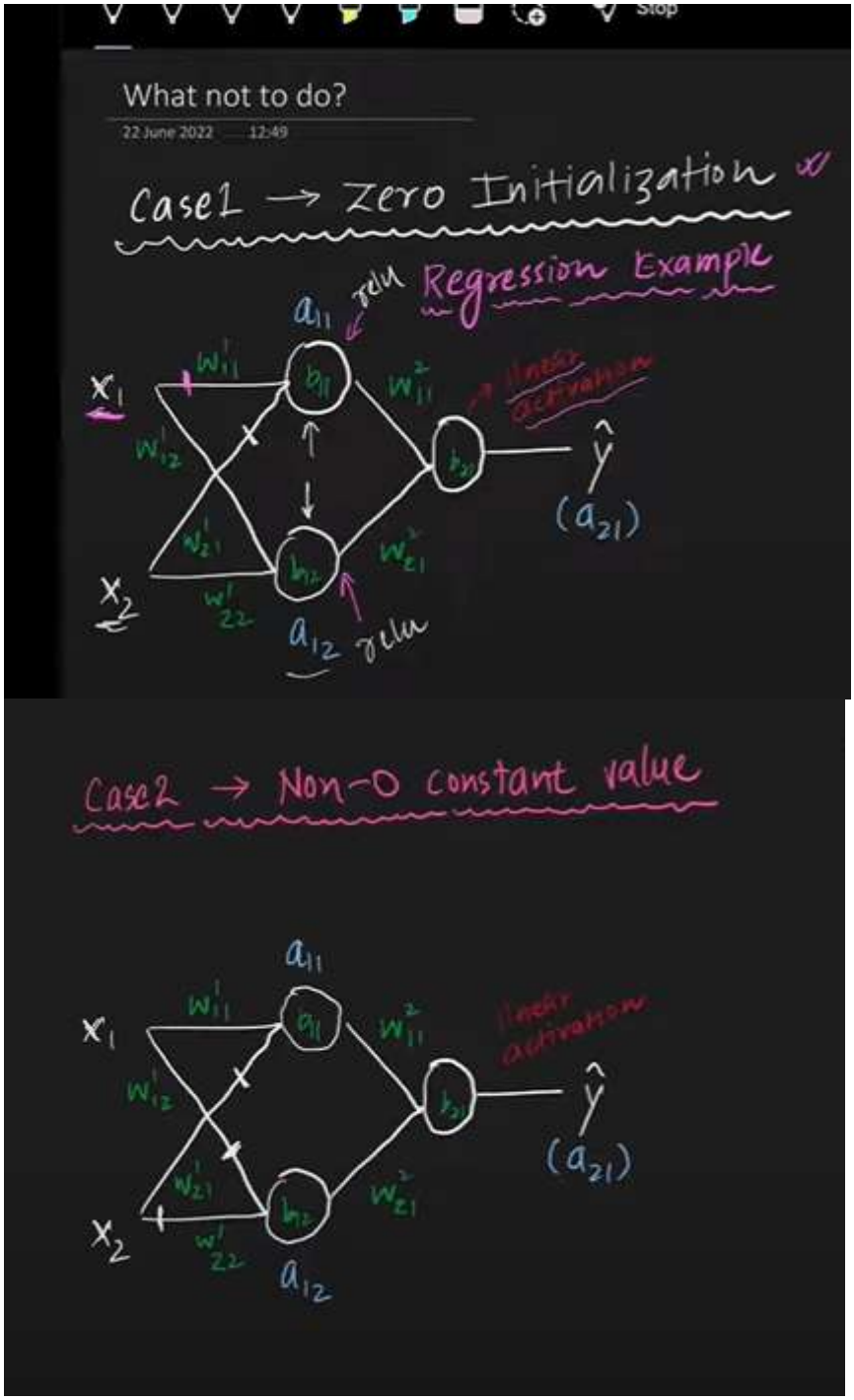
→ Vanishing GP
→ Exploding GP
→ Slow convergence

1. Initialize the parameters
2. Choose an *optimization algorithm* ✓
3. Repeat these steps:
    1. Forward propagate an input
    2. Compute the cost function
    3. Compute the gradients of the cost with respect to parameters using backpropagation
    4. Update each parameter using the gradients, according to the optimization algorithm

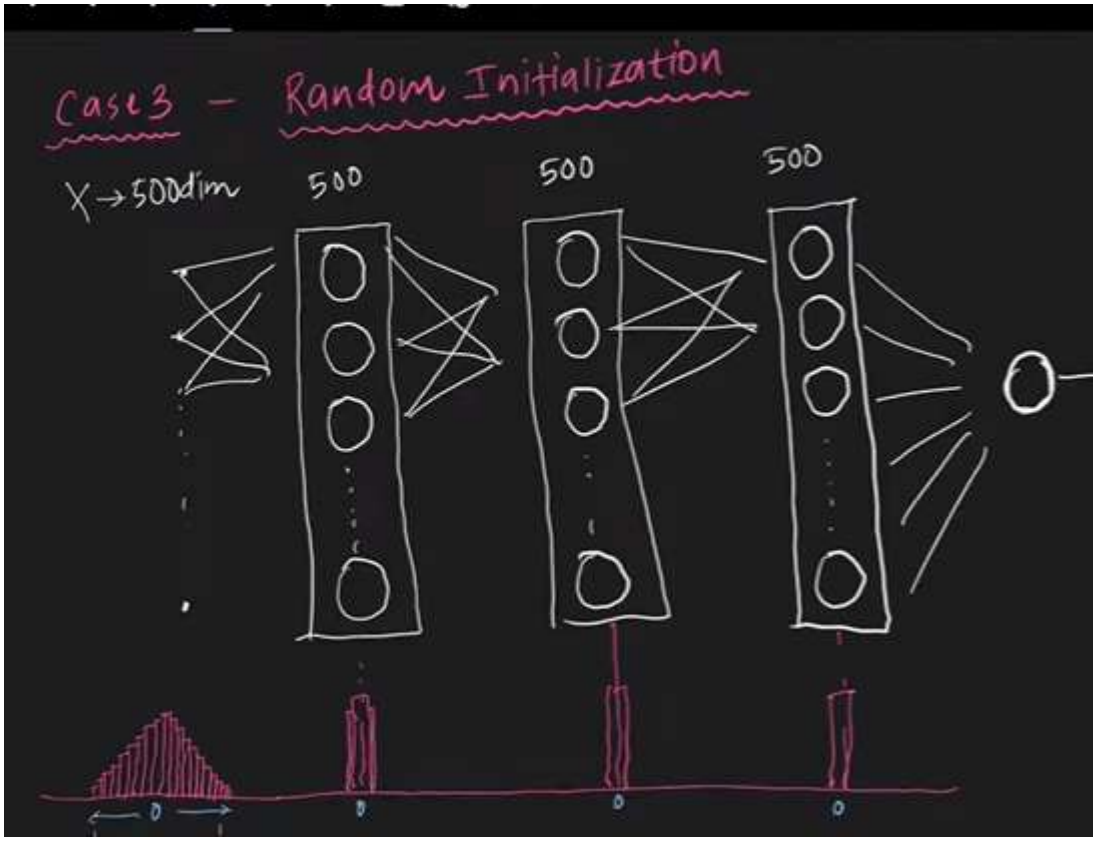


WEIGHT INITIALIZATION TECHNIQUES

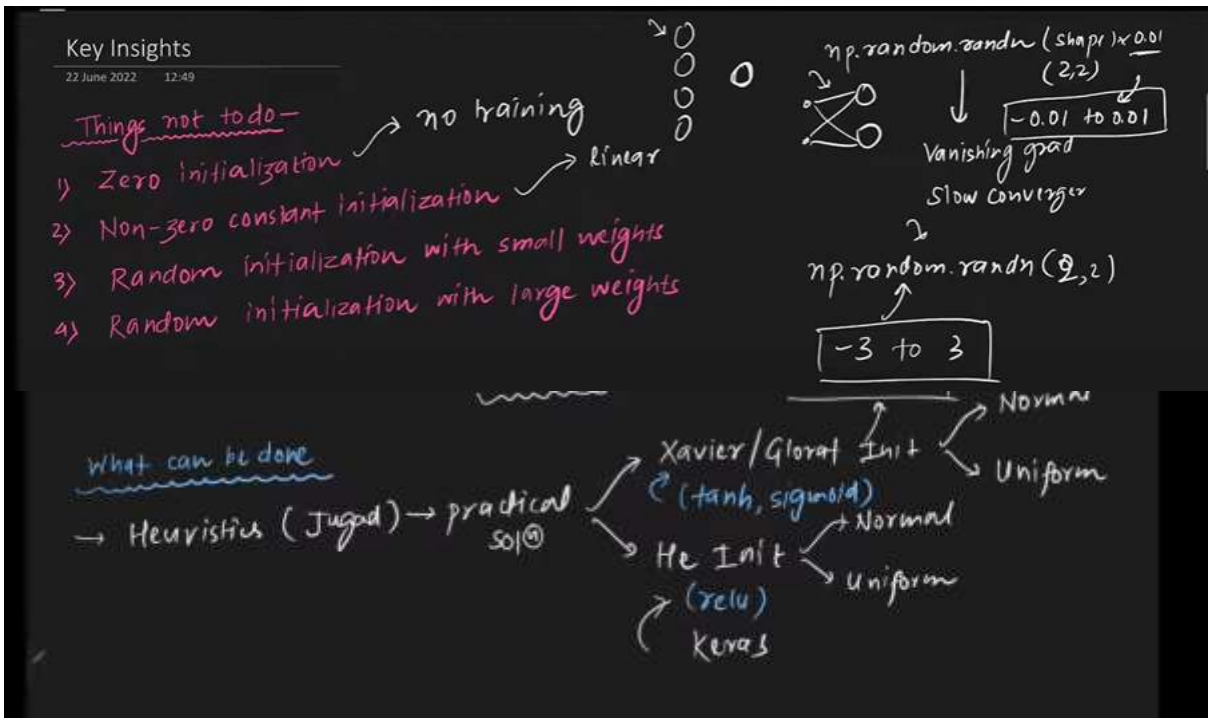




all weights and biases given same value



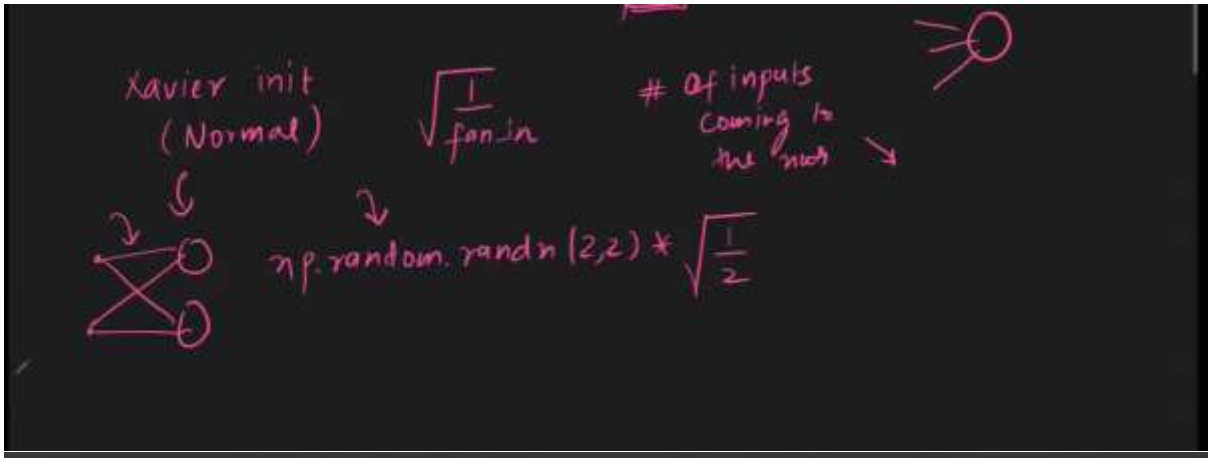
all below 4 techiques are wrong



we know weights can only be initialized by random values

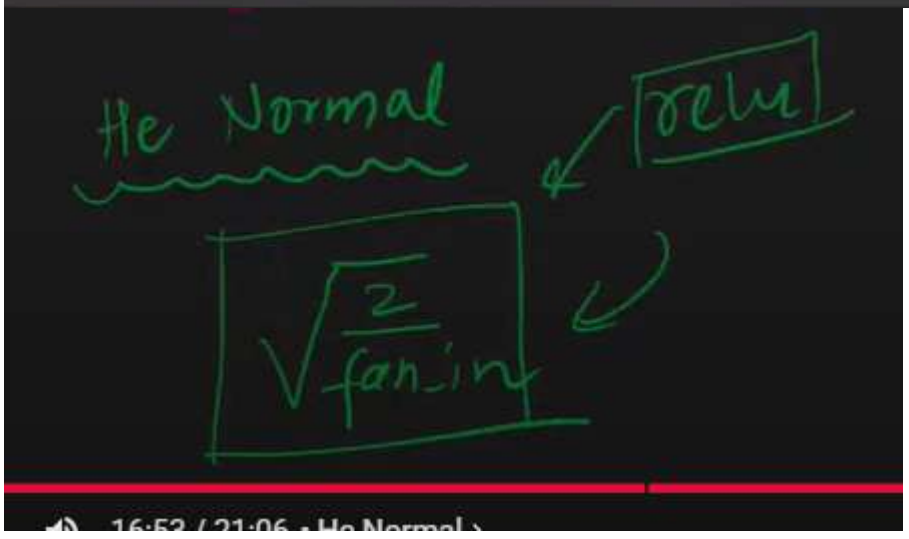
but problem is that they cant be very small values nor it can be very big values

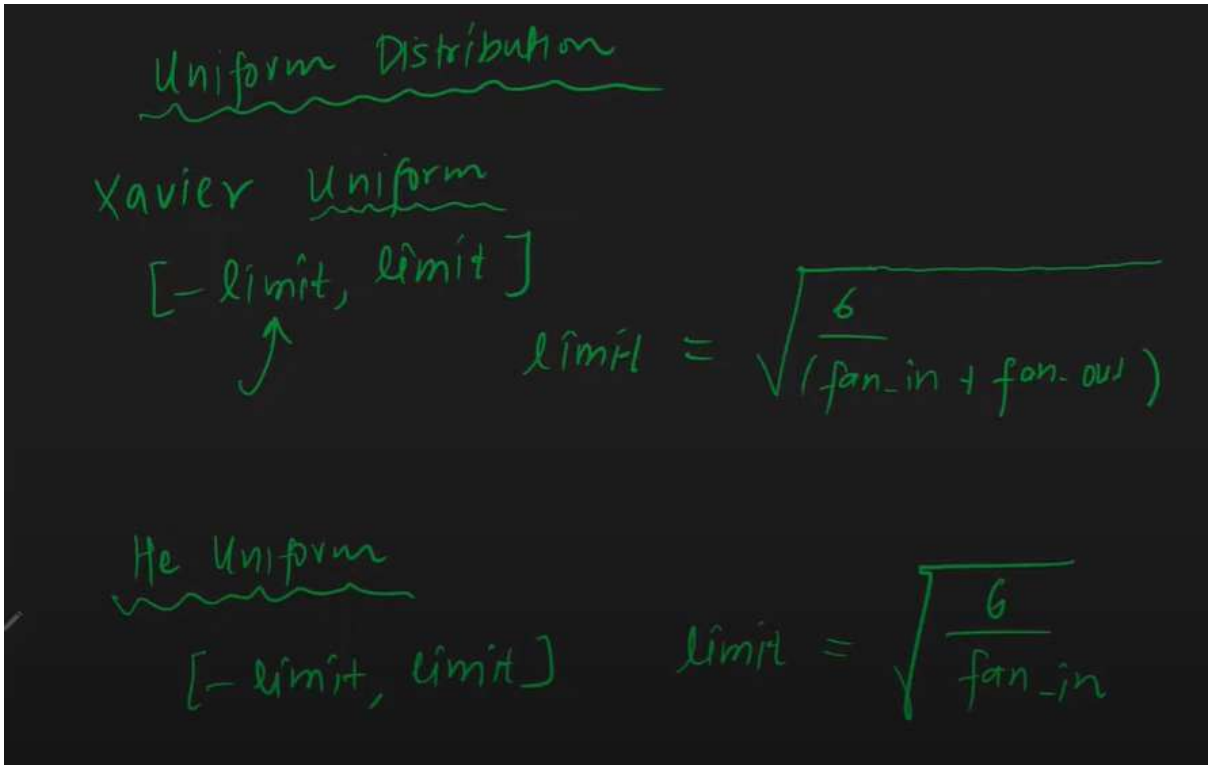
so we should have some appropriate range of values from which random w and b are picked




```
initial_weights[0] = np.random.randn(2,10)*np.sqrt(1/2)
initial_weights[1] = np.zeros(model.get_weights()[1].shape)
initial_weights[2] = np.random.randn(10,10)*np.sqrt(1/10)
initial_weights[3] = np.zeros(model.get_weights()[3].shape)
initial_weights[4] = np.random.randn(10,10)*np.sqrt(1/10)
initial_weights[5] = np.zeros(model.get_weights()[5].shape)
initial_weights[6] = np.random.randn(10,10)*np.sqrt(1/10)
initial_weights[7] = np.zeros(model.get_weights()[7].shape)
initial_weights[8] = np.random.randn(10,1)*np.sqrt(1/10)
initial_weights[9] = np.zeros(model.get_weights()[9].shape)
```

```
[48] model.set_weights(initial_weights)
```





`class GlorotNormal` : The Glorot normal initializer, also called Xavier normal initializer.

`class GlorotUniform` : The Glorot uniform initializer, also called Xavier uniform initializer.

`class HeNormal` : He normal initializer.

`class HeUniform` : He uniform variance scaling initializer.

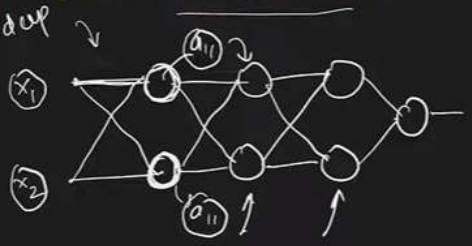
```
model = Sequential()

model.add(Dense(10,activation='relu',input_dim=2,kernel_initializer='he_uniform'))
model.add(Dense(10,activation='relu',kernel_initializer='he_normal'))
model.add(Dense(10,activation='relu',kernel_initializer='he_normal'))
model.add(Dense(10,activation='relu',kernel_initializer='he_normal'))
model.add(Dense(1,activation='sigmoid'))

model.summary()
```

What is Batch Norm?

27 June 2022 11:00

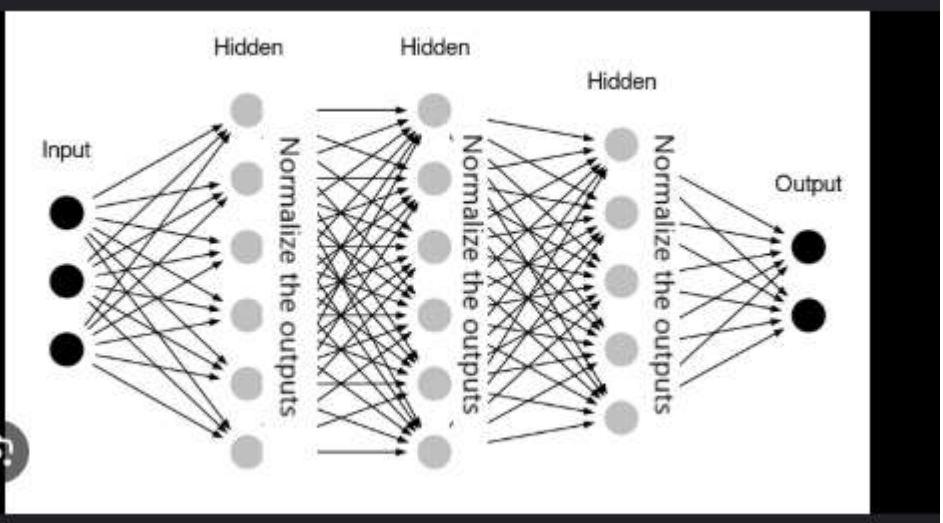
Batch-Normalization (BN) is an algorithmic method which makes the training of Deep Neural Networks (DNN) faster and more stable.

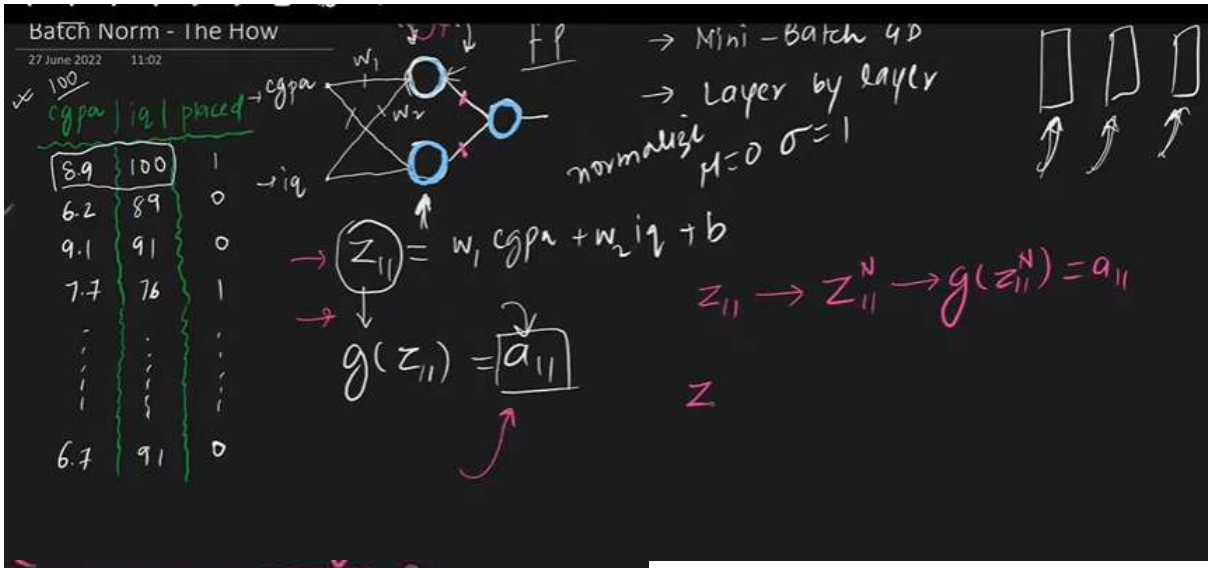
It consists of normalizing activation vectors from hidden layers using the mean and variance of the current batch. This normalization step is applied right before (or right after) the nonlinear function.

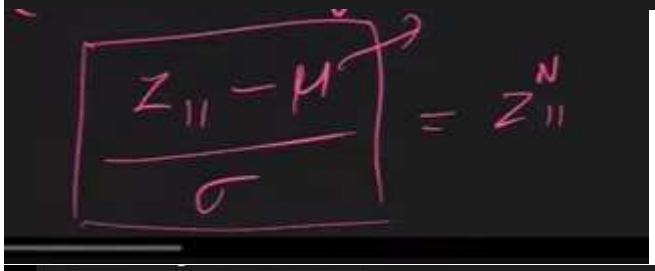


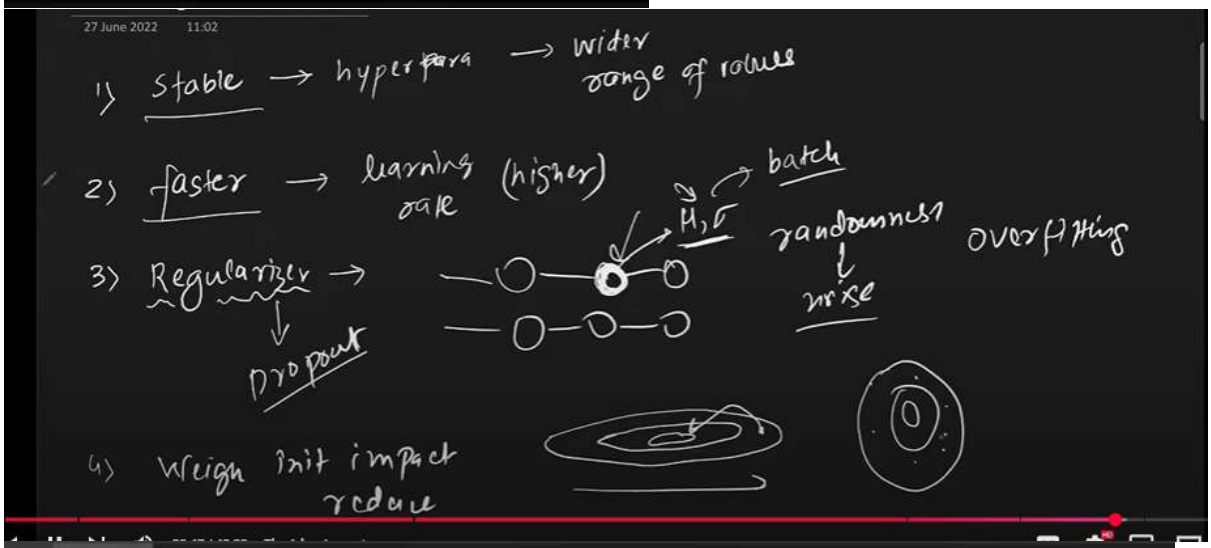
normalized
$m=0$
$sd=1$

normalize
$m=0$ $sd=1$



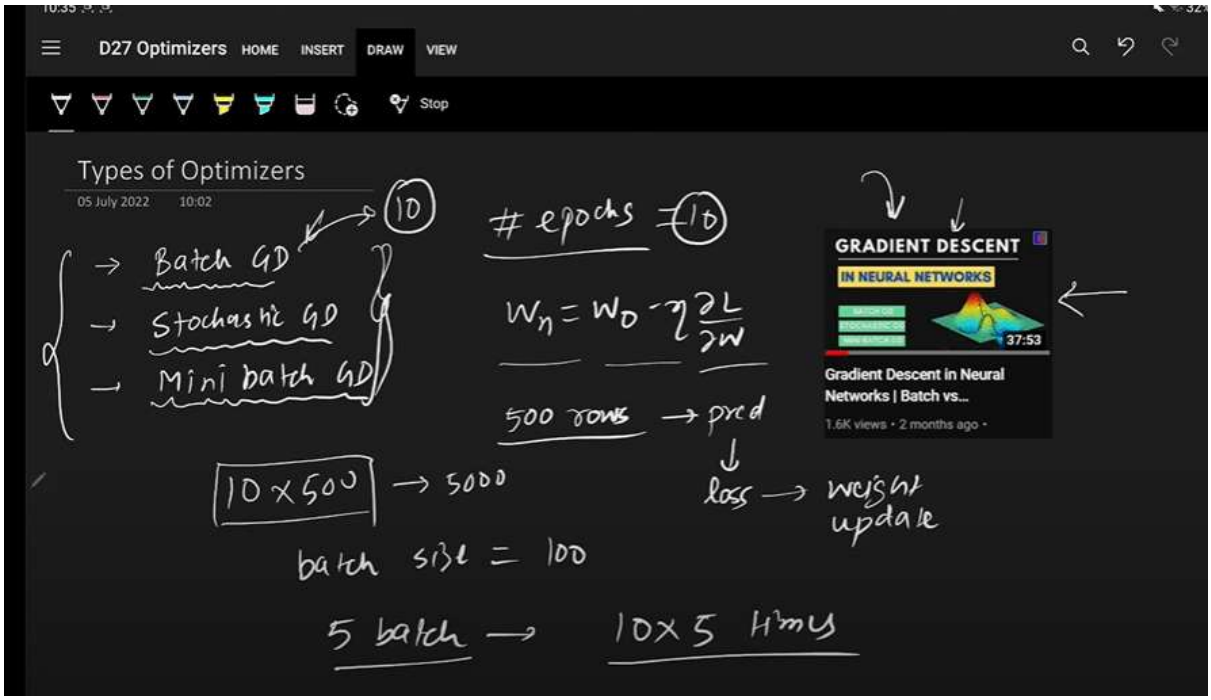




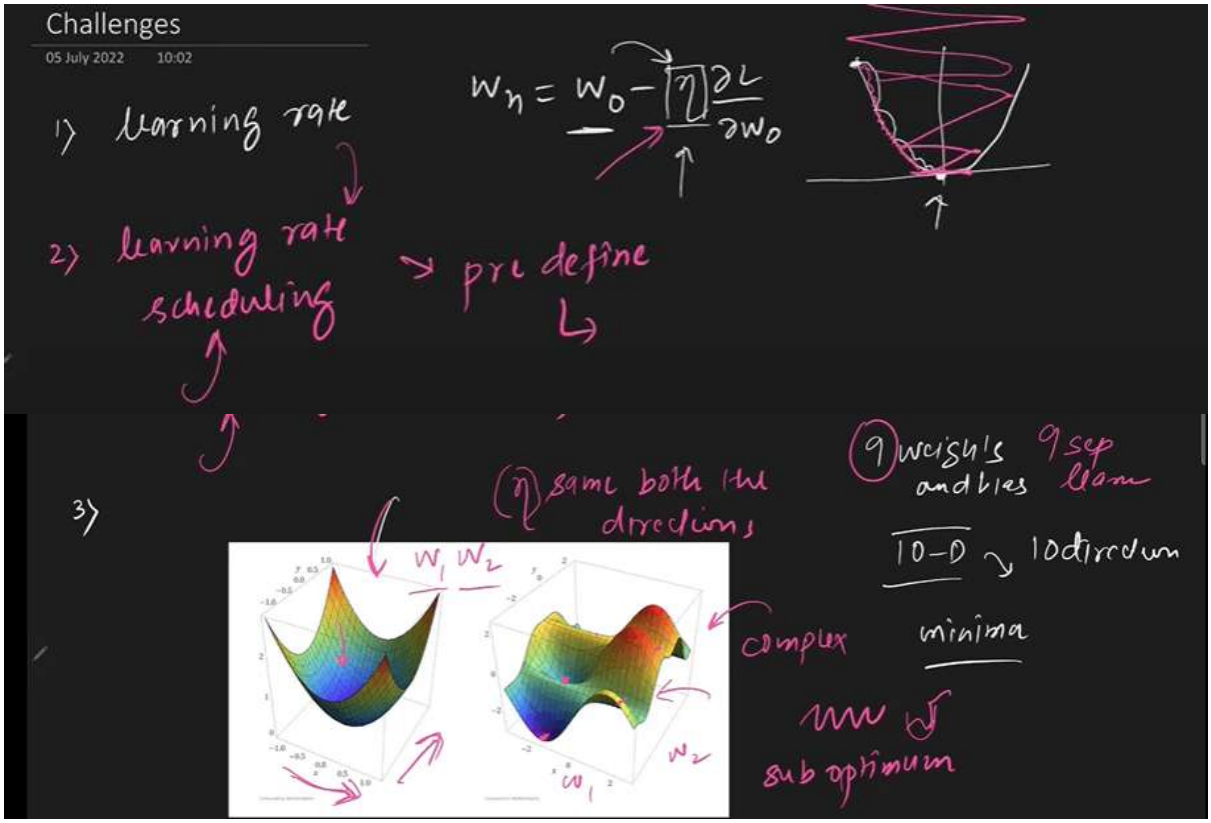


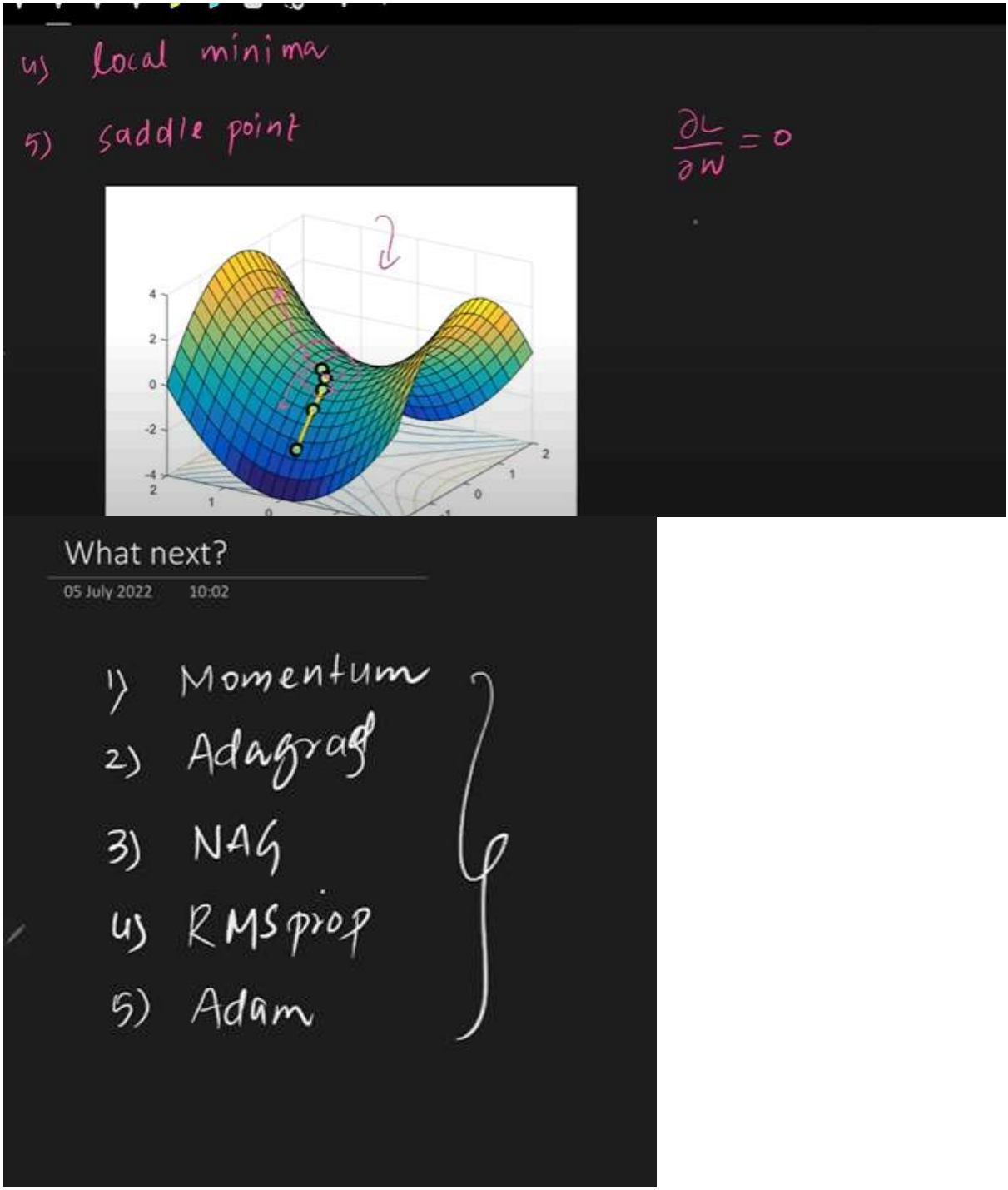

```
model = Sequential()

model.add(Dense(3,activation='relu',input_dim=2))
model.add(BatchNormalization())
model.add(Dense(2,activation='relu'))
model.add(BatchNormalization())
model.add(Dense(1,activation='sigmoid'))

model.summary()
```

OPTMIZERS

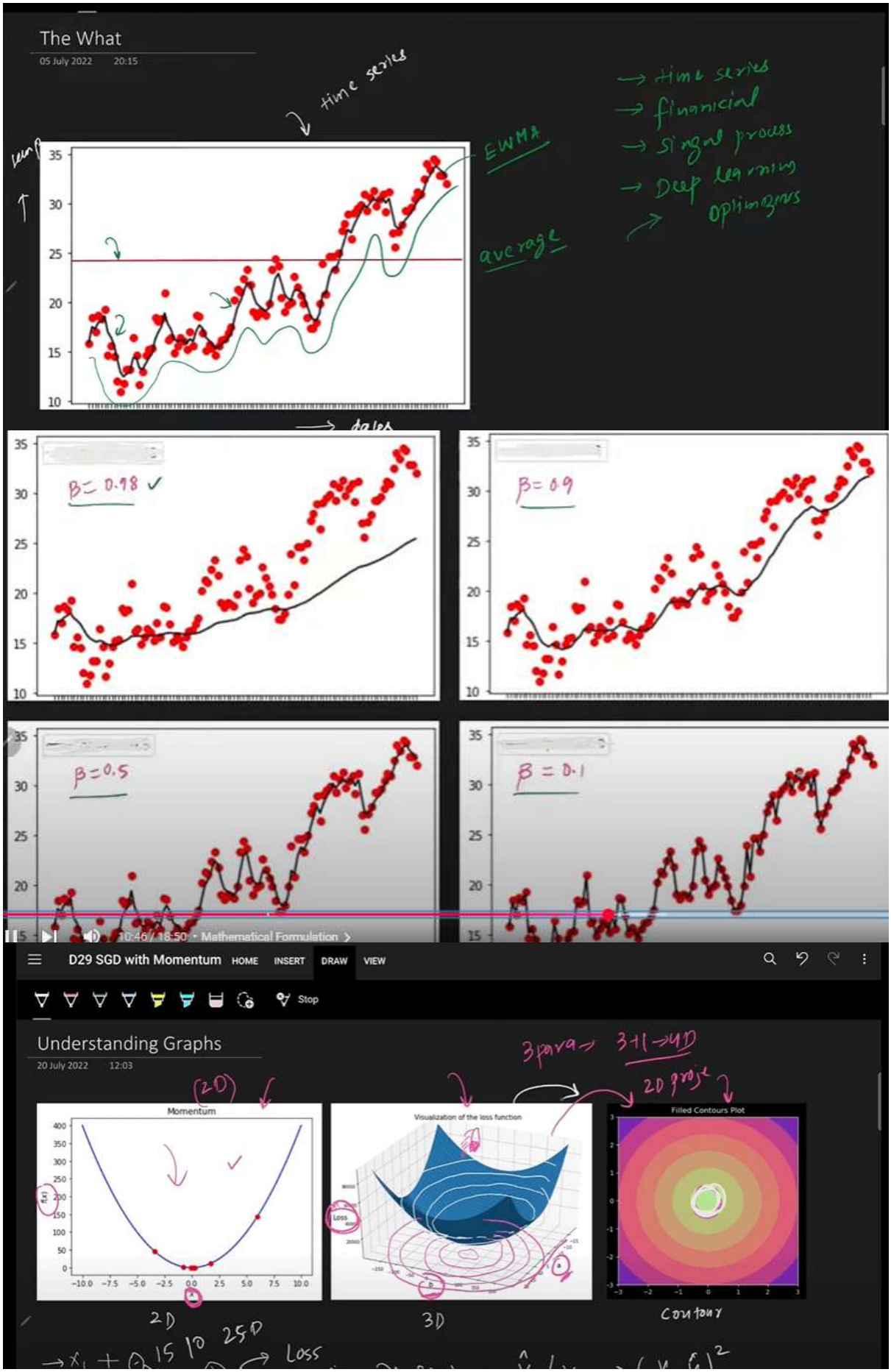


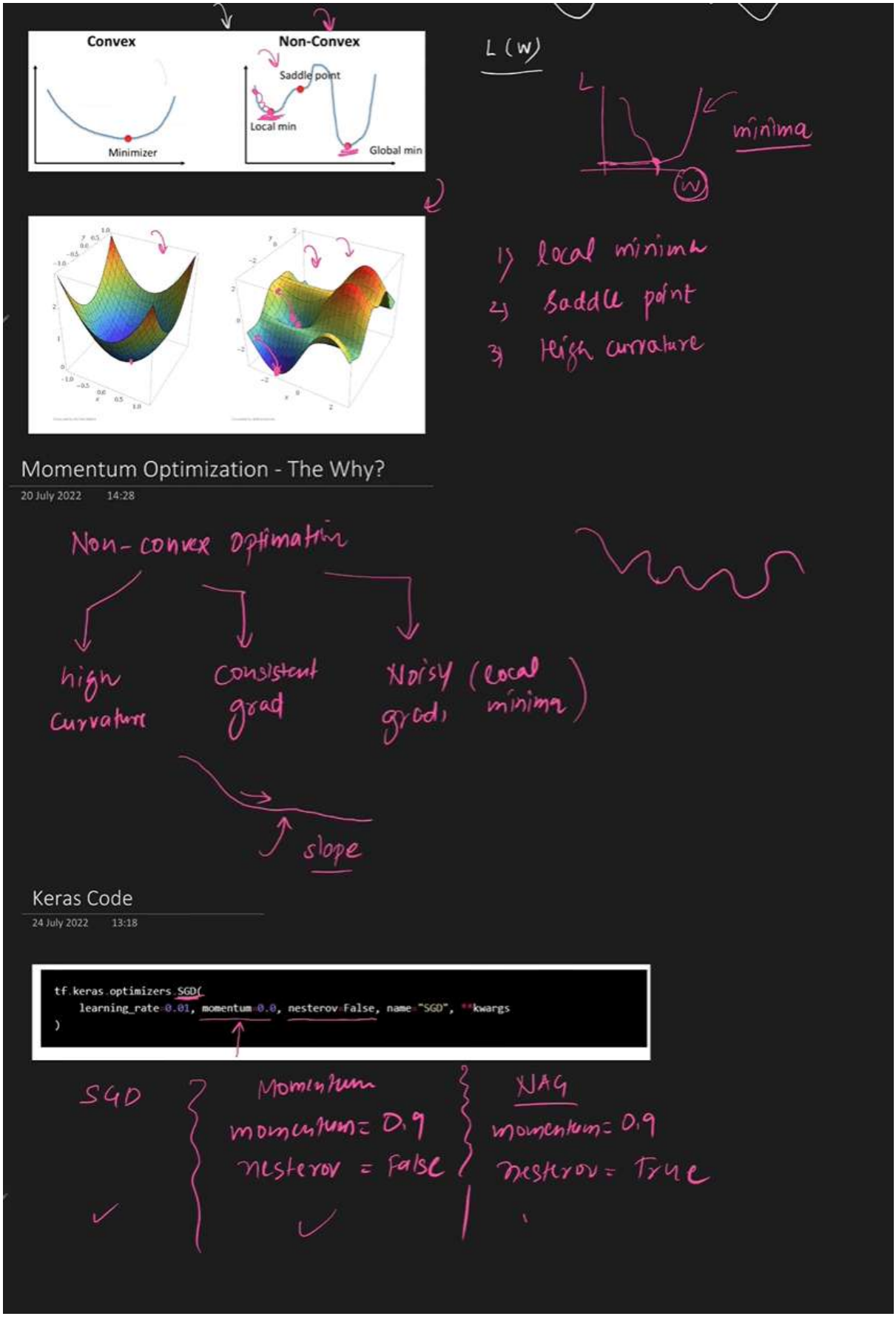
challengers with GD





EWMA Exponentially Weighted Moving Average or Exponential Weighted Average


The What
05 July 2022 20:15
time series
EWMA
average
→ time series
→ financial
→ signal process
→ Deep Learning
Optimizers
β = 0.98
β = 0.9
β = 0.5
β = 0.1
10:46 / 18:50 • Mathematical Formulation
D29 SGD with Momentum
HOME
INSERT
DRAW
VIEW
Stop
Understanding Graphs
20 July 2022 12:03
3para → 3+1 → 4D
2D proje
Momentum
Visualization of the loss function
Filled Contours Plot
2D
3D
Contour
Loss



batch gradient descent problem is speed

momentum uses past gradients for current update , but problem is oscillations

NAG overcame oscillations

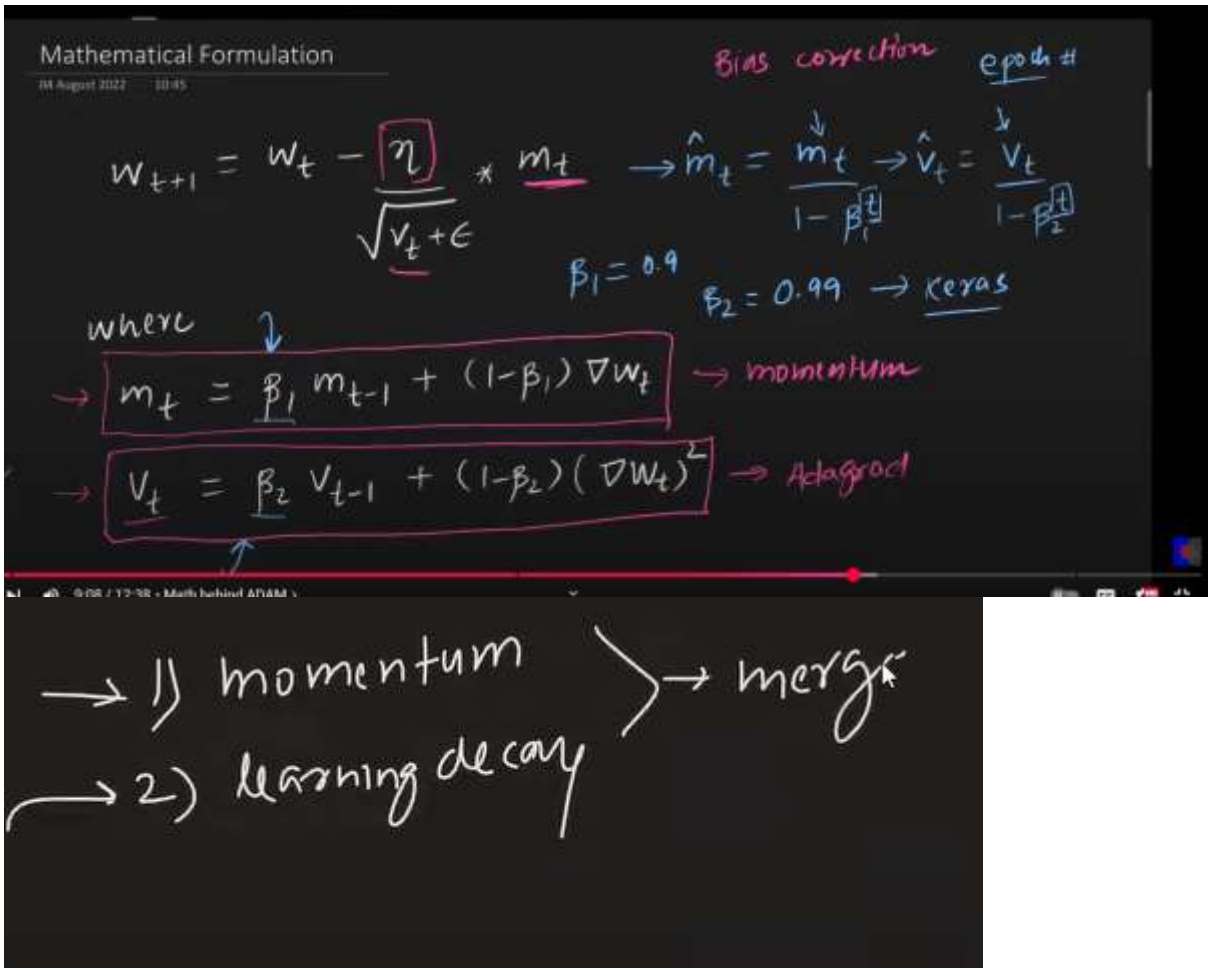
but still problem was with sparse column

Adagrad solve sparse problem by decaying learning rate , but problem is it was not able to reach minima

RMS prop solved everything , no disv

so overall two optimzation mindsets

either momentum or LR decay , ADAM merged both



ADAM IS BEST

THEN RMS PROP

KERAS TUNER

```python
from keras.layers import Dropout
def build_model(hp):
    model = Sequential()

    # Input layer + First hidden layer: Tune the number of neurons
    model.add(Dense(units=hp.Int('input_units', min_value=8, max_value=128, step=8),
                    activation='relu', input_dim=8))

    # Add a tunable dropout layer after the first hidden layer
    model.add(Dropout(hp.Choice('dropout_input', values=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.
    9])))


    # Add a tunable number of hidden layers between 0 and 10
    for i in range(hp.Int('num_layers', min_value=0, max_value=10)):
        # Tune the number of neurons in each hidden layer
        model.add(Dense(units=hp.Int(f'layer_{i}_units', min_value=8, max_value=128, step=8),
                        activation='relu'))
        # Add a tunable dropout layer after each hidden layer
        model.add(Dropout(hp.Choice('dropout' + str(i), values=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
        0.8, 0.9])))


    # Output layer
    model.add(Dense(1, activation='sigmoid'))


    # Tune the optimizer
    optimizer = hp.Choice('optimizer', values=['adam', 'sgd', 'rmsprop', 'adadelta'])

    model.compile(
        optimizer=optimizer,
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return model
```