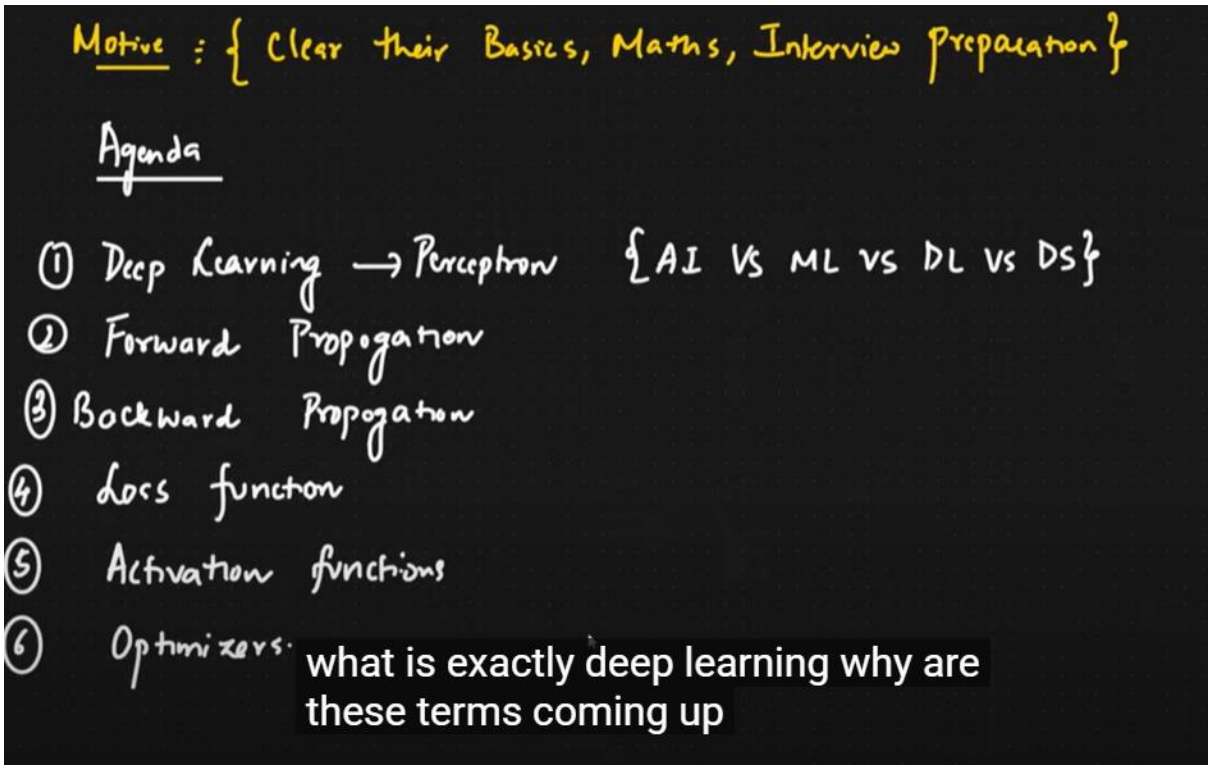
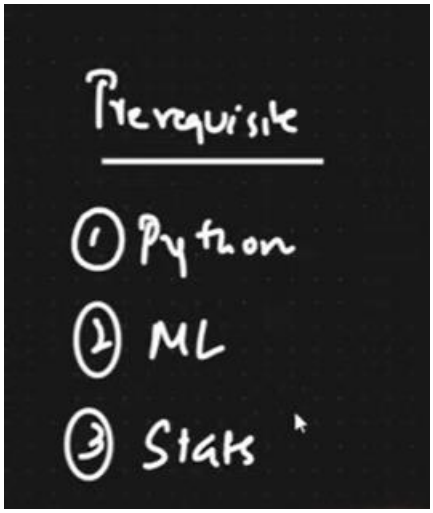
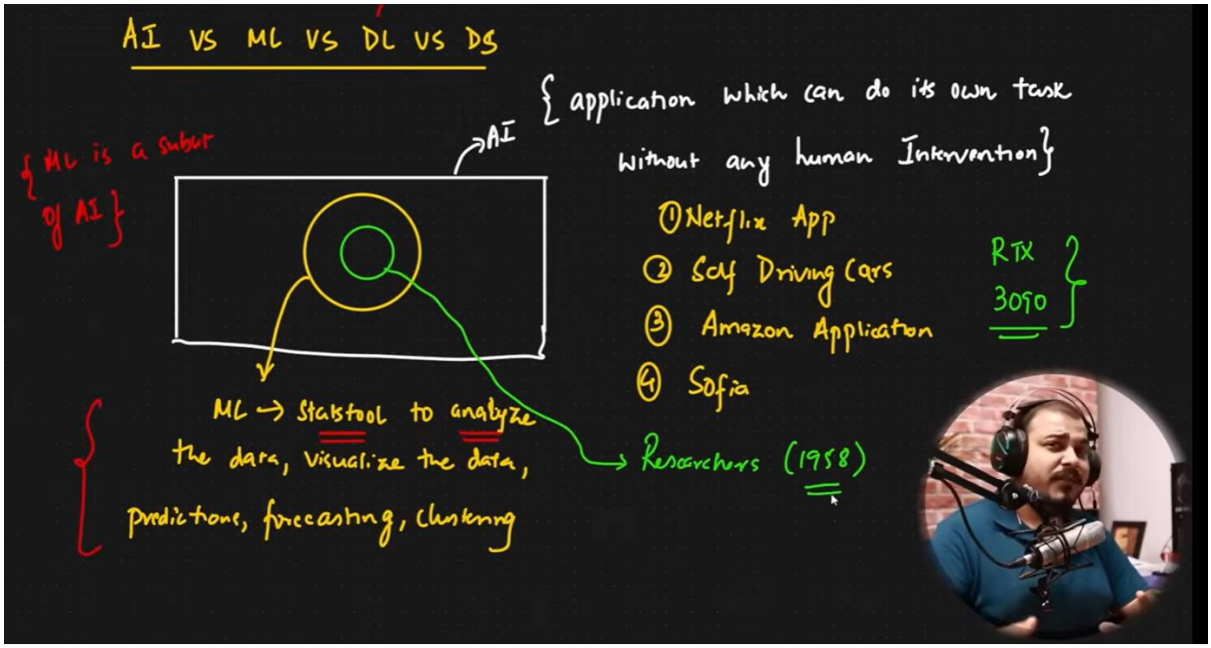
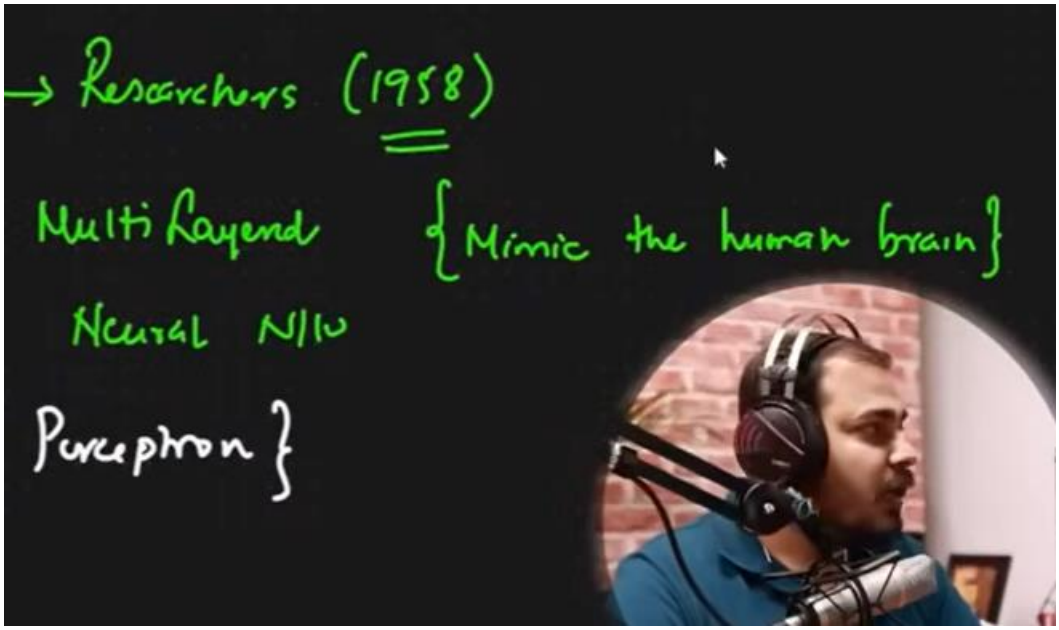
https://www.youtube.com/watch?v=d2kxUVwWWwU









① 2005 → ORKUT, FACEBOOK, Instagram, Whatsapp, LinkedIn Twitter

DATA → Exponentially ↑↑↑

2008 → {BIG DATA} → Efficiently

2013 → Company had huge amount of Data

⇓ {AI → popular}

Scanners Products

Panasonic : AC's, TV's, Refrigerator {Data}

↓ ⇓

Model → Reduce the Electricity Bills ⇓

⇓

Subscription Basis

⇓

{ Cumulate Revenues, Better Decisions }

② Hardware Advancement (NVidia) → GPU's → TRAINING THE MODEL

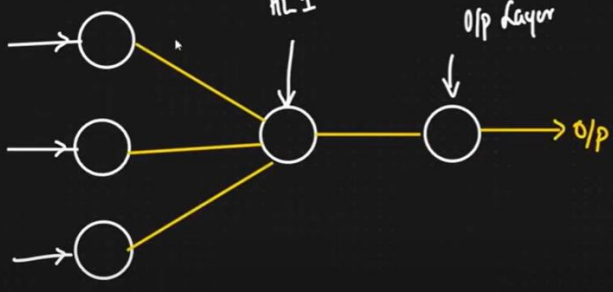
GPU's → cost ⇓⇓⇓

④ Perceptron { Single Layered Neural N/W }.

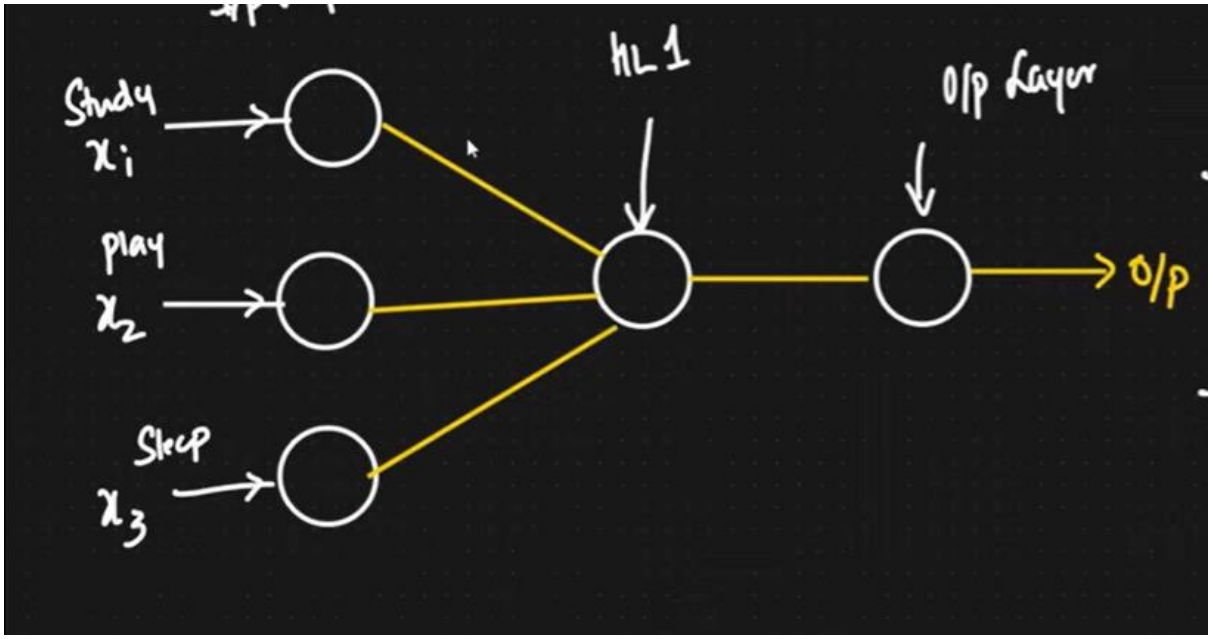
Binary classification

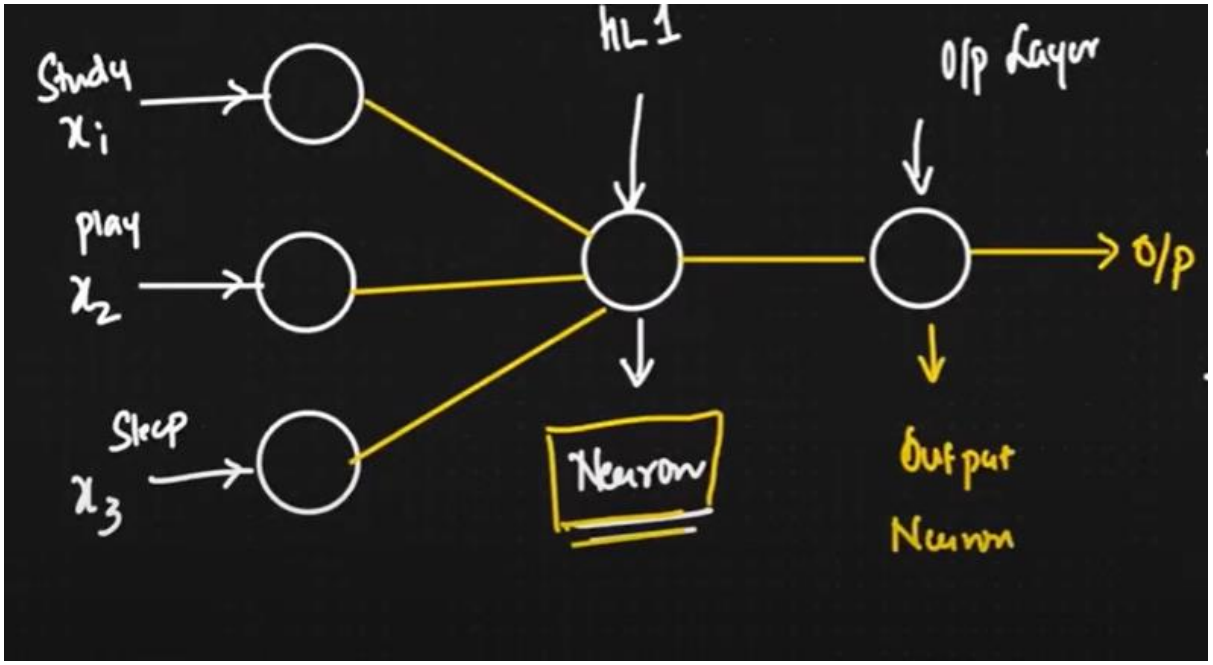
Dataset ↓

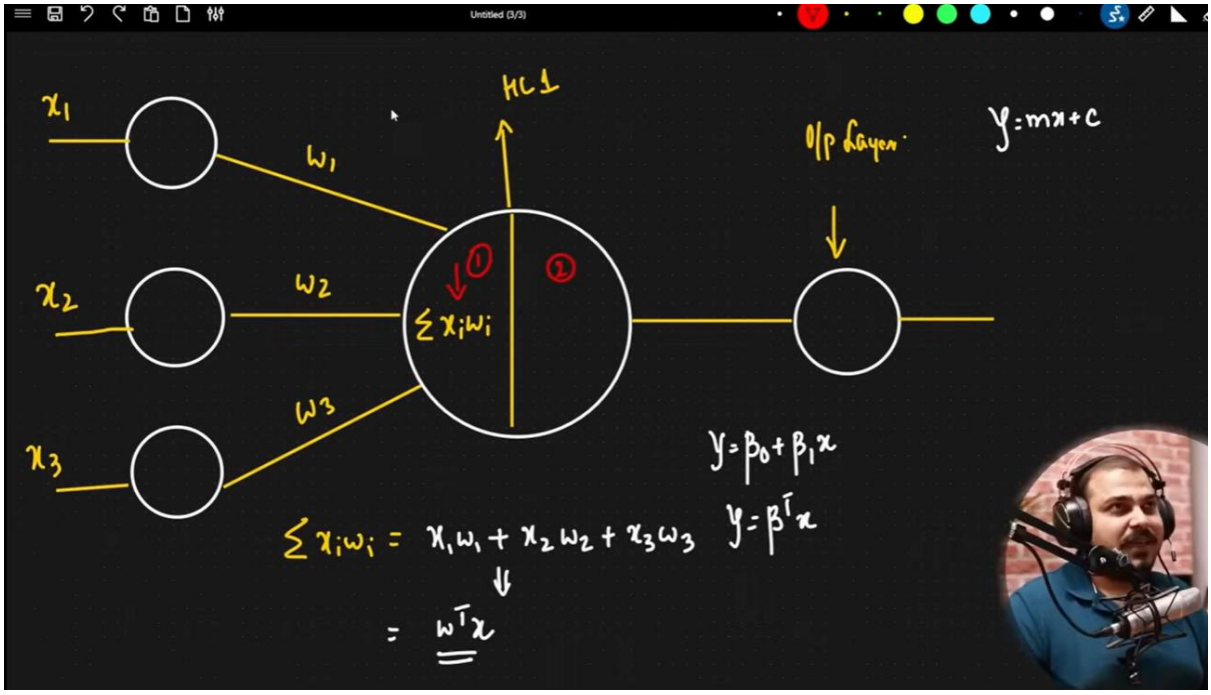
I/p layers HL 1 O/p Layer



Study	Play	Sleep	Pass/Fail
→ 7	3	7	1
O/p → 2	5	8	0
→ 4	3	7	1


HL 1
O/p Layer
Study
x_1
play
x_2
Sleep
x_3
O/p

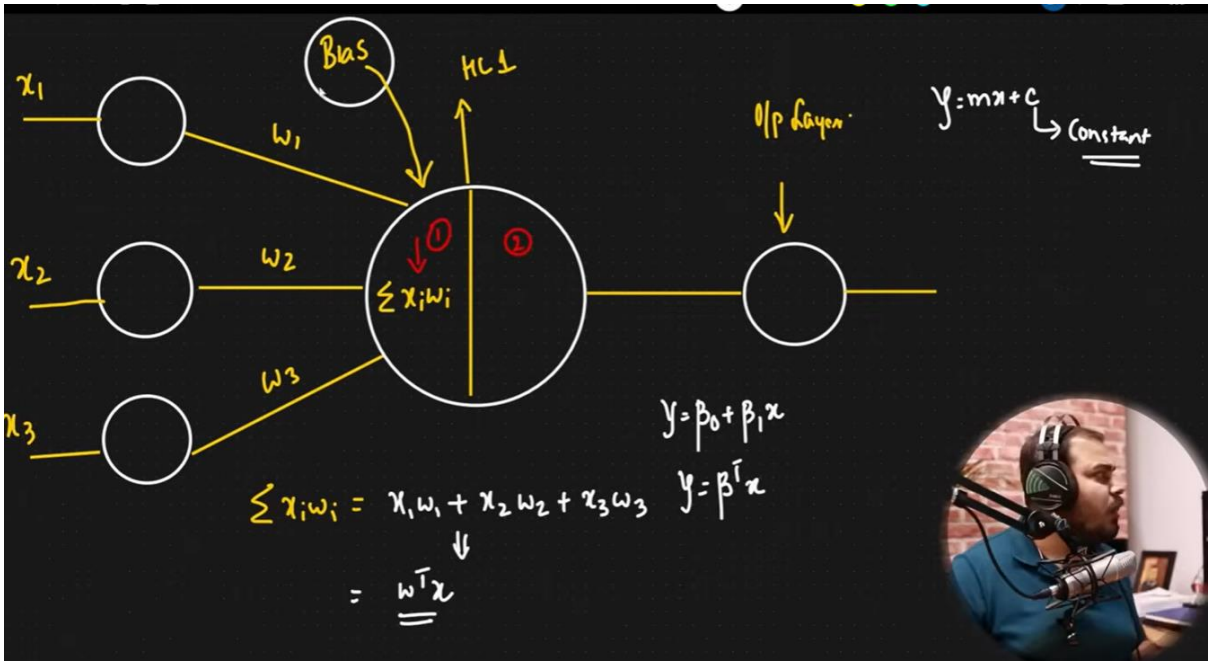

HL 1
O/p Layer
Study
x_1
play
x_2
Sleep
x_3
O/p
Neuron
Output
Neuron

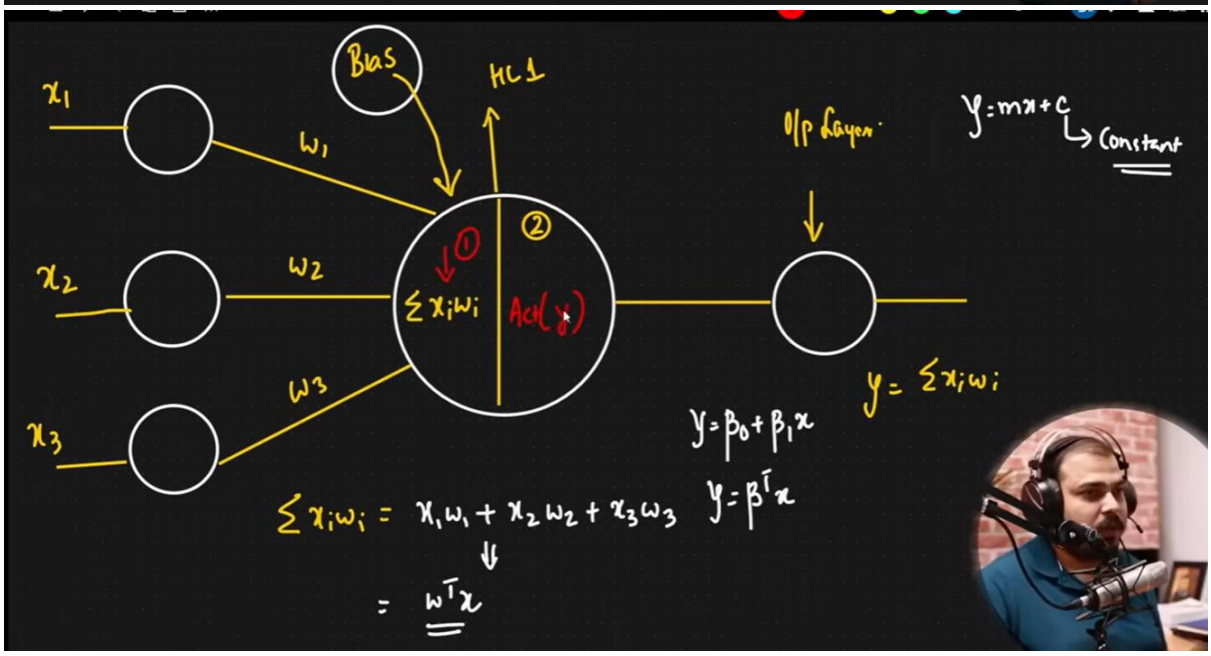


Neruons will tell or give signal at which level should get activated eg when you place a hot object in hand neurons will give signal to the brain to get activate to remove the object

During training of nn we need to decided which values to get assign as weight to the neurons, so neurons will signal based on value to which level it should get activated

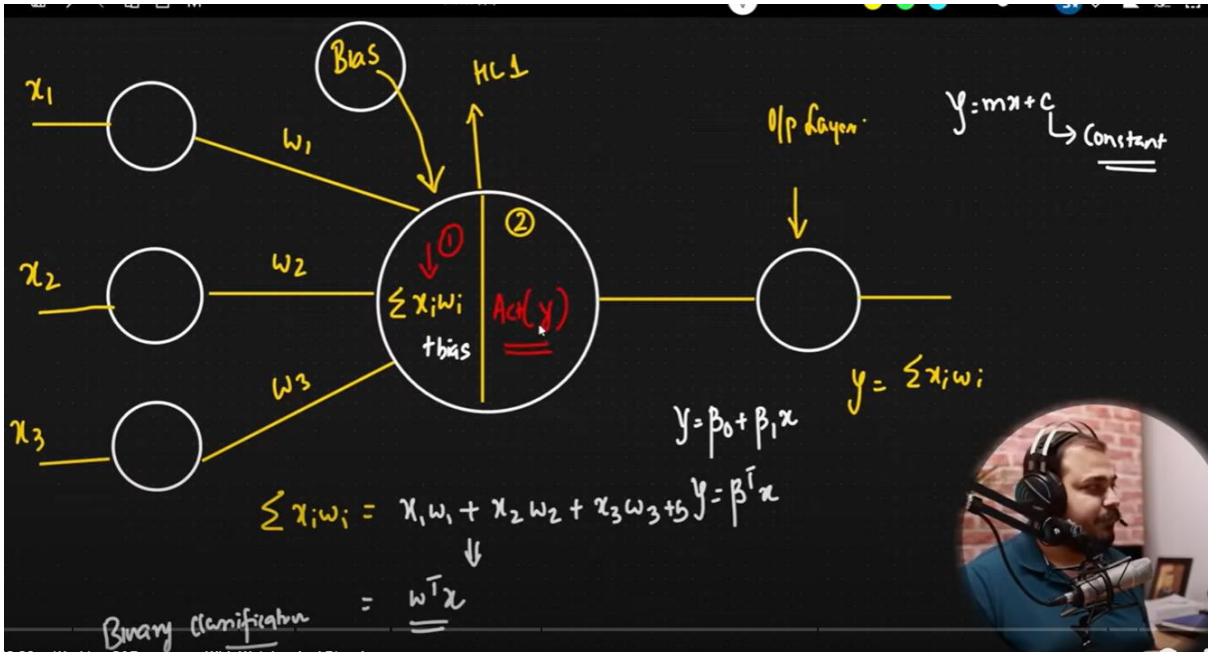
Weight should not be zero so in order to overcome lets add parameter called bias

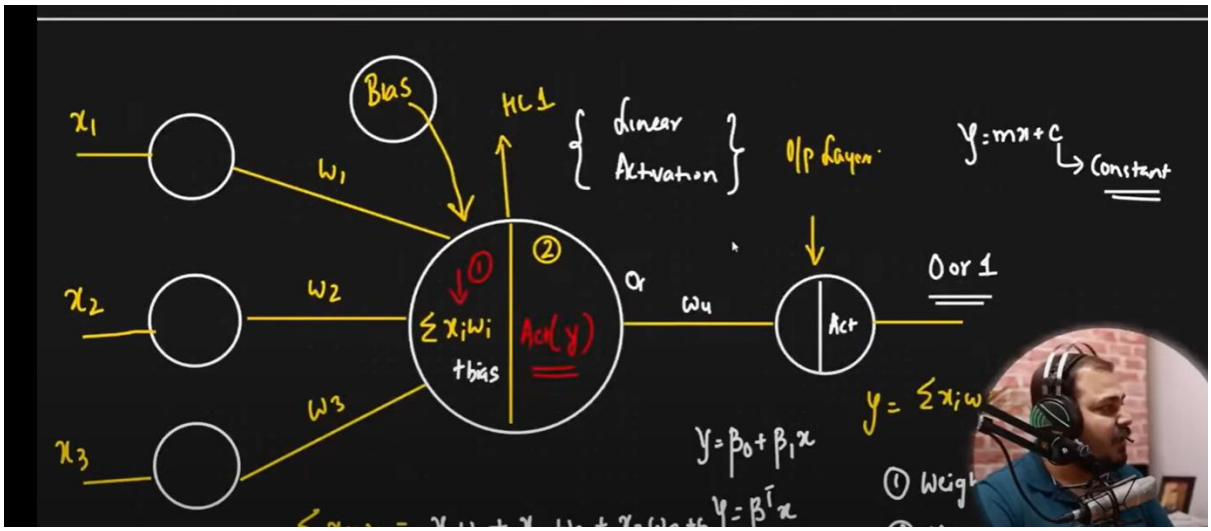


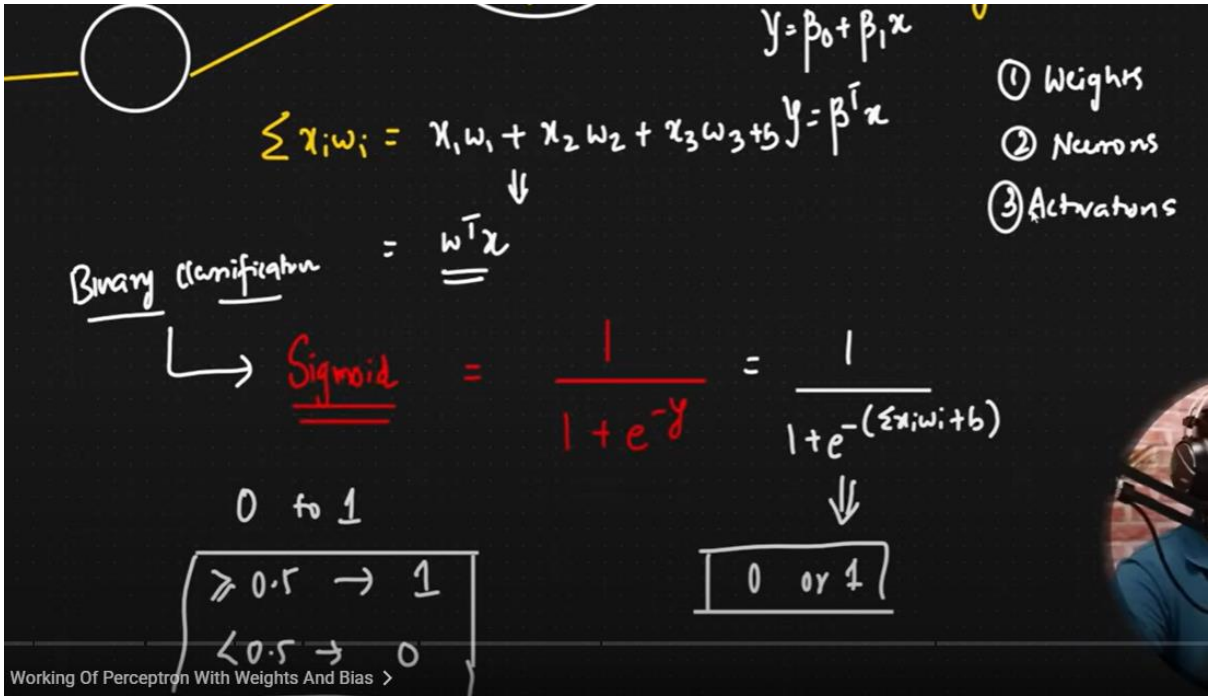


We will pass on activation function on top of y

Sigmoid activation function


Bias
HL1
x_1
w_1
x_2
w_2
x_3
w_3
① $\sum x_i w_i$ + bias
② Act(y)
O/p Layer
$y = mx + c$ → Constant
$y = \sum x_i w_i$
$y = \beta_0 + \beta_1 x$
$\sum x_i w_i = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$
$y = \beta^T x$
$= w^T x$
Binary classification


Bias
HL1
{ linear Activation }
O/p Layer
$y = mx + c$ → Constant
x_1
w_1
x_2
w_2
x_3
w_3
① $\sum x_i w_i$ + bias
② Act(y)
or
w_4
Act
0 or 1
$y = \beta_0 + \beta_1 x$
$y = \beta^T x$
$y = \sum x_i w$
① Weig

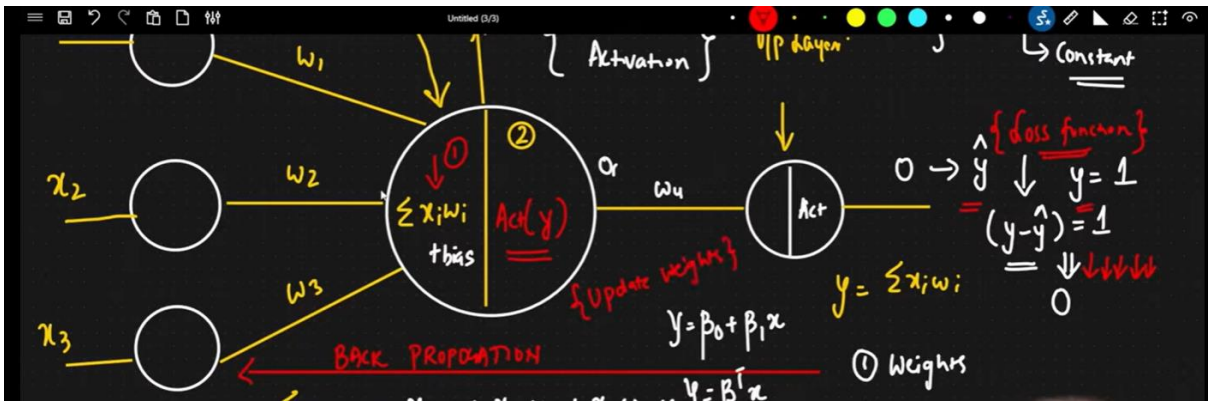


Forward propagation

We can take a input we take a input multiplied by weights add a bias and then activate neurons get the outputs it is called forward propagation

Backward propagation

In order to minimize the error (loss function) we need to update the weights then it is back propagation

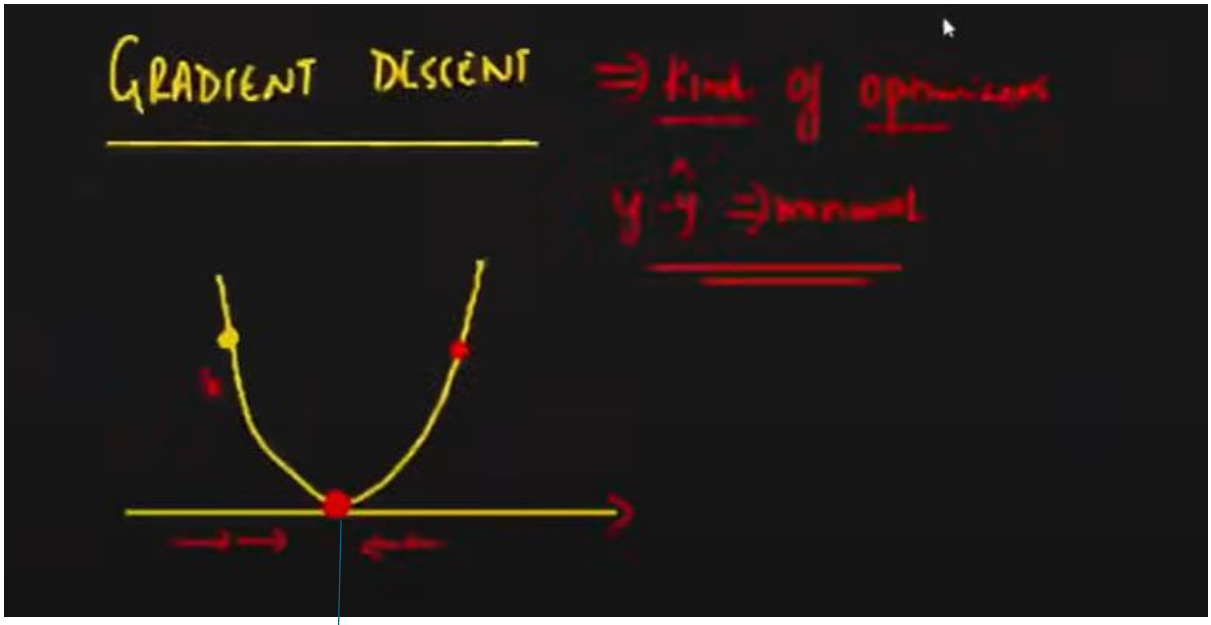


Optimizers

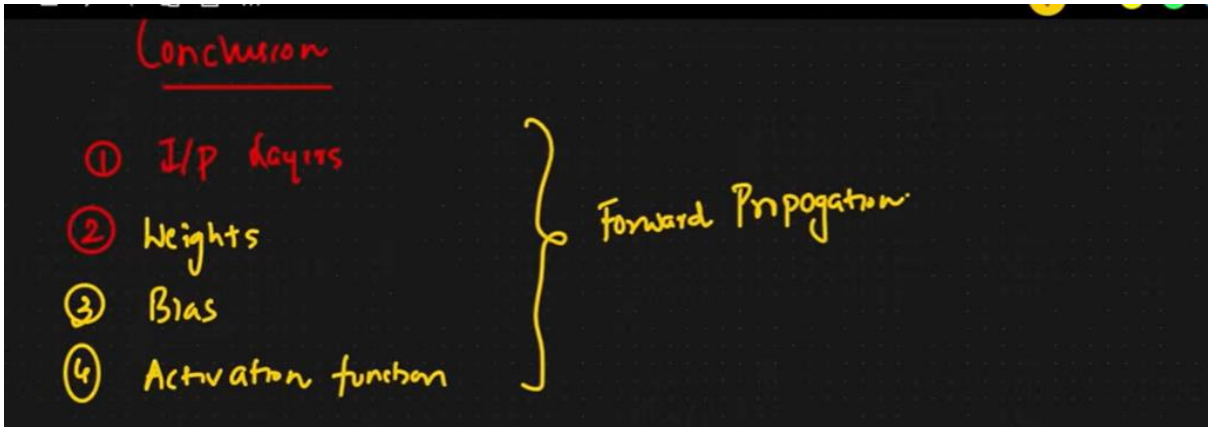
How we update the weights that is done by optimizers

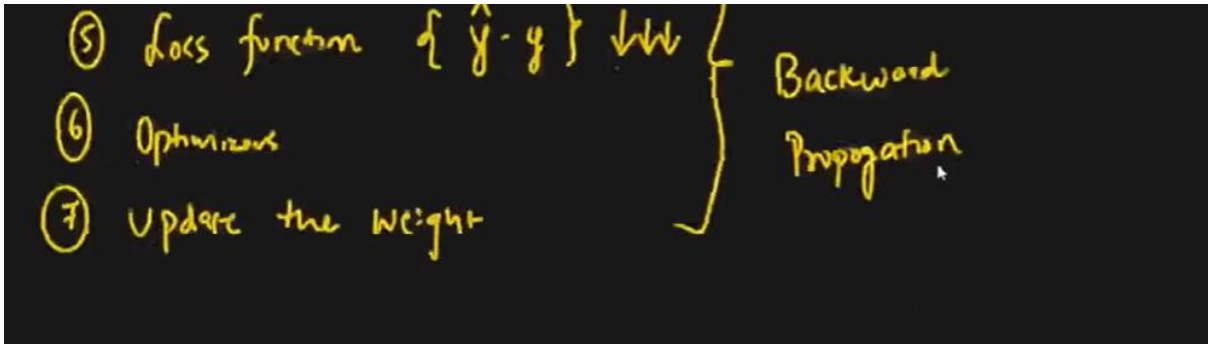
Gradient descent is a optimizers

It is combination of both forward and backward in linear regression backwards weights will get updated and in forward weights will get multiplied with the coefficients

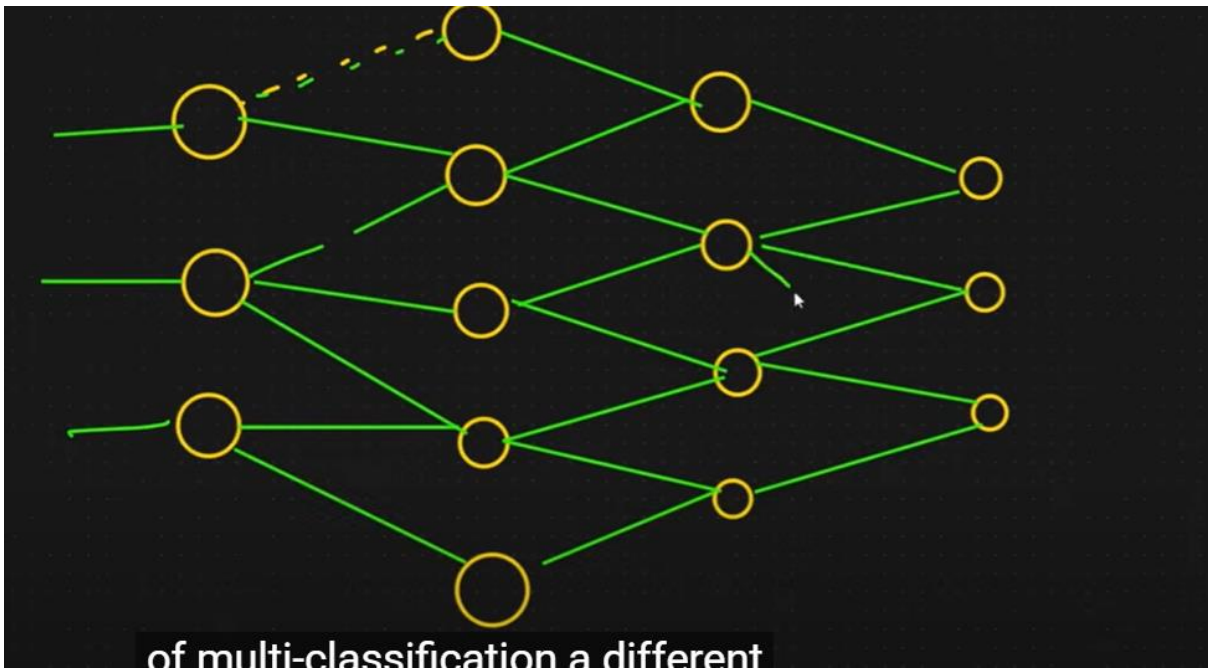


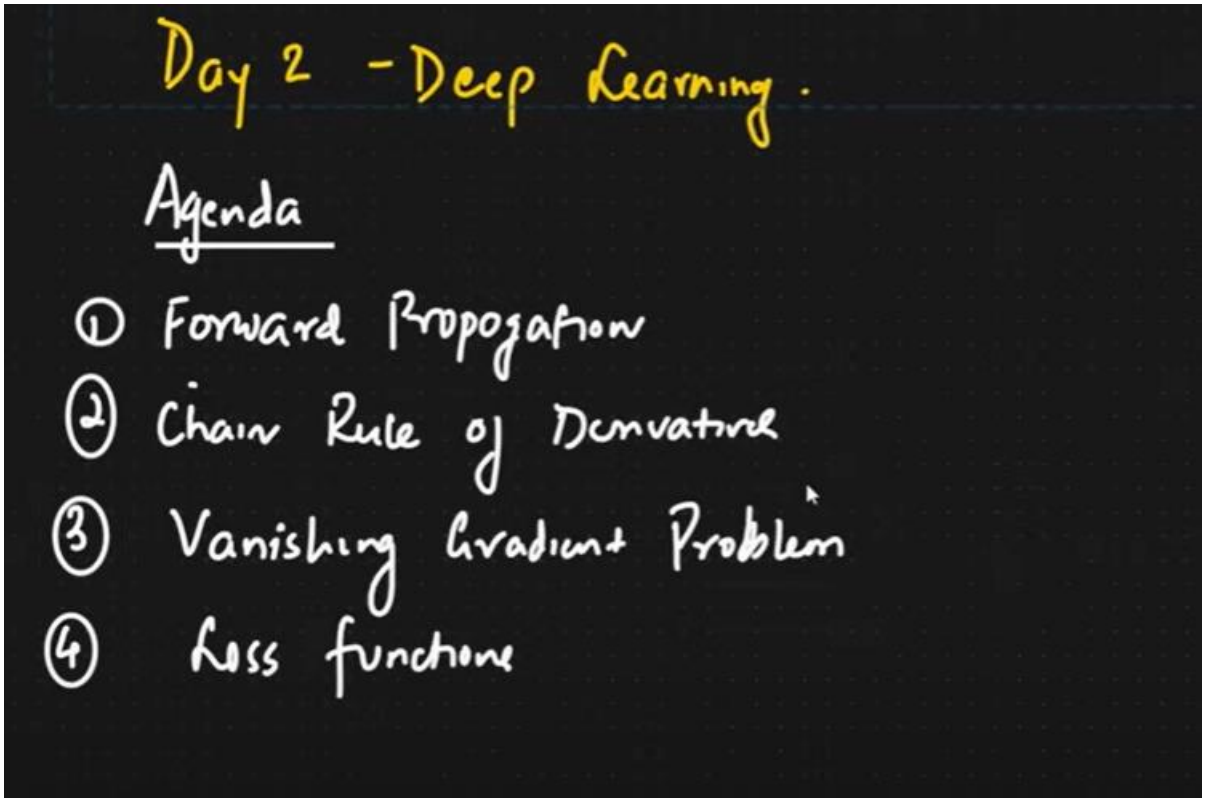
Global minima

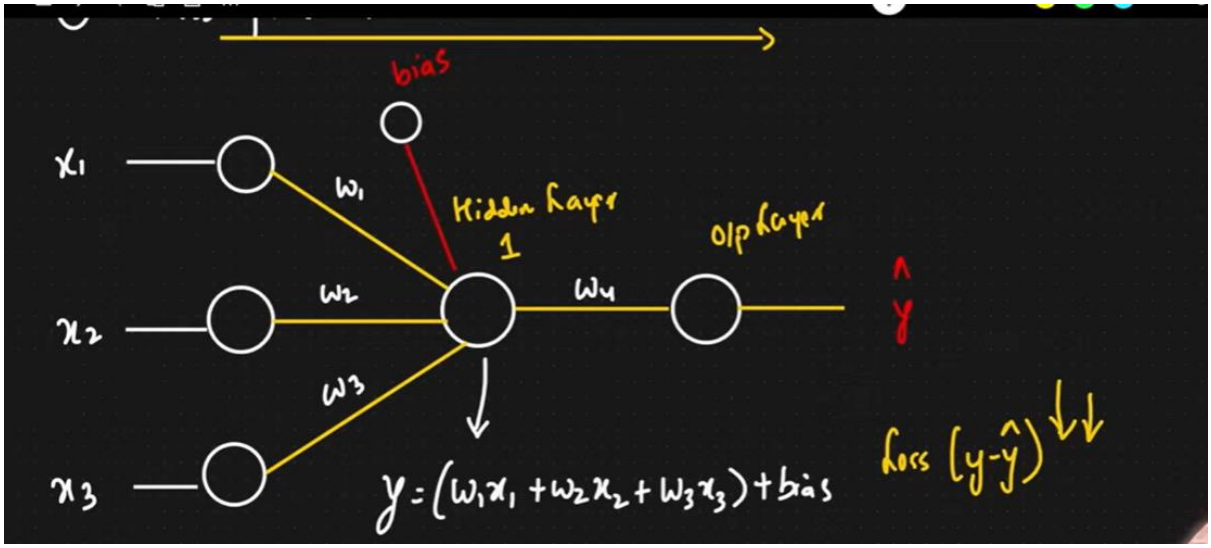


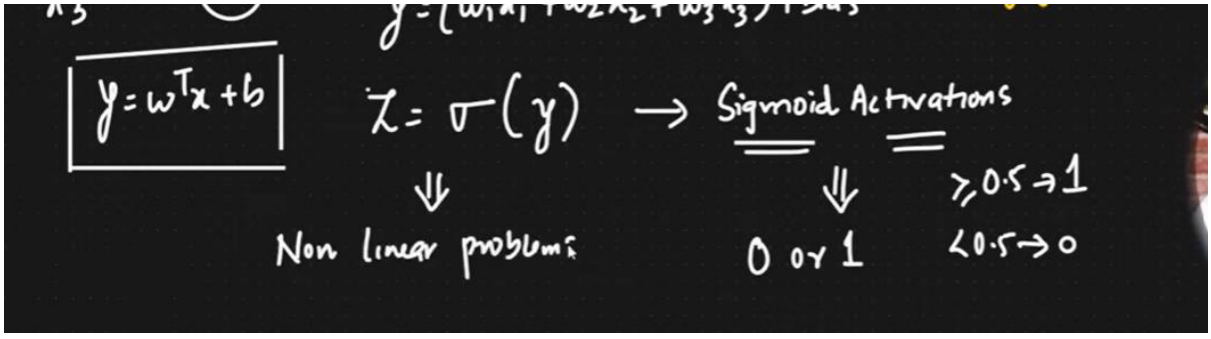


Multilayer neural network





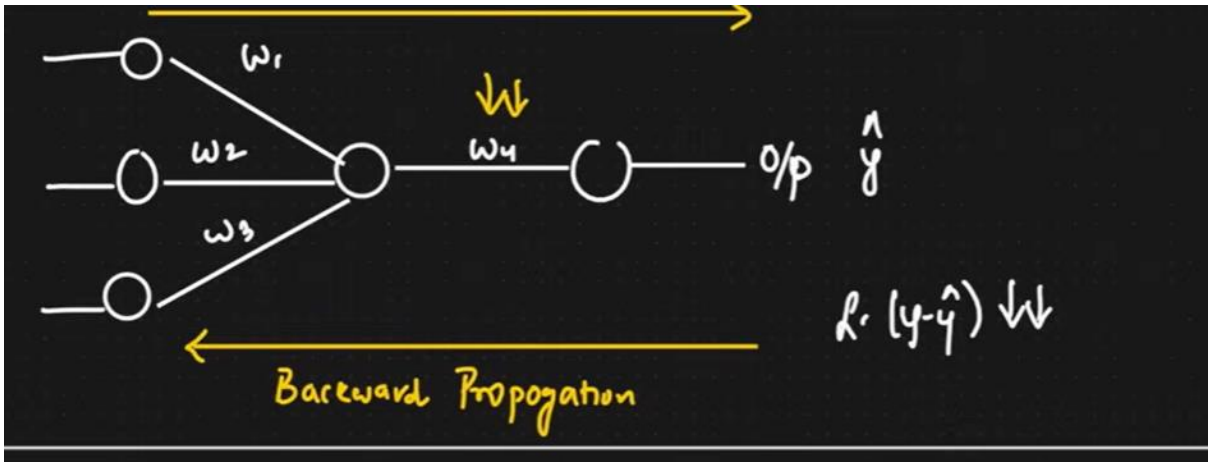


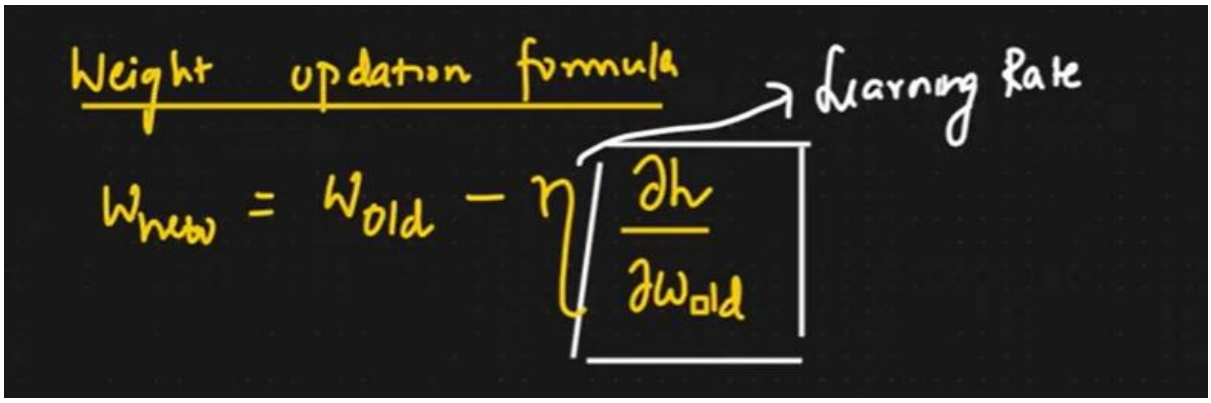


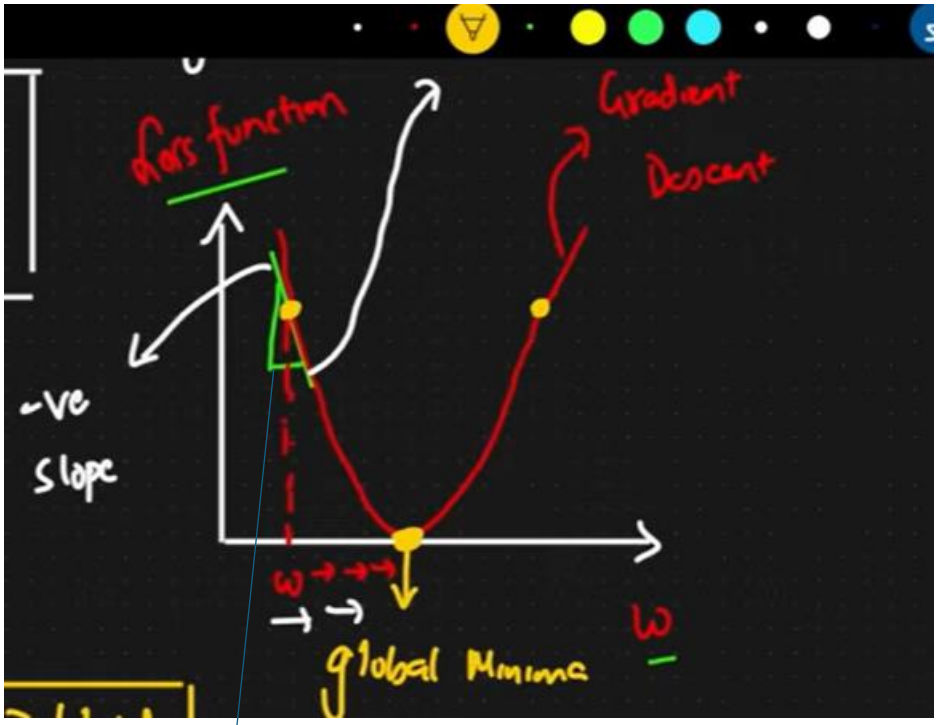
Backpropagation

Updation of weights happens in backpropagation

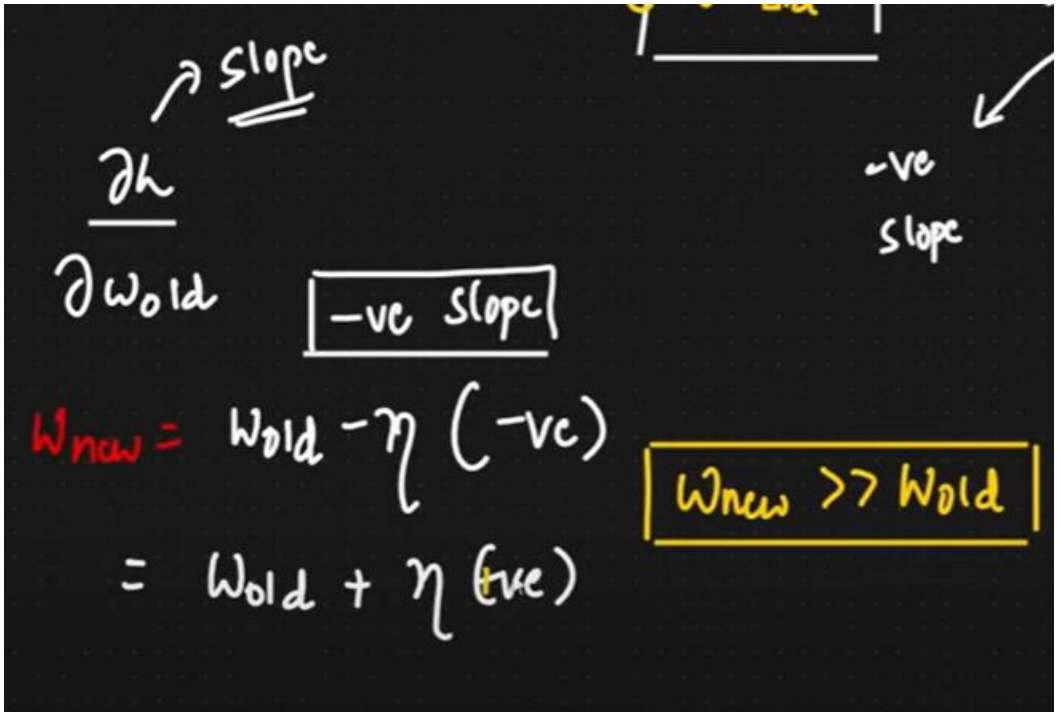
Chain rule of differentiations

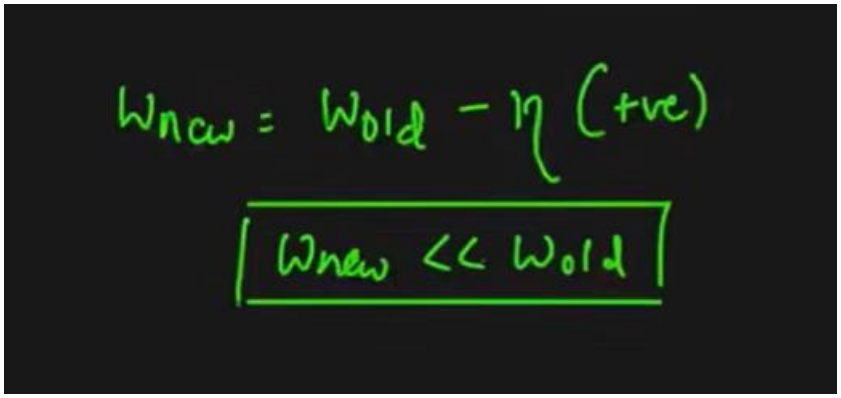


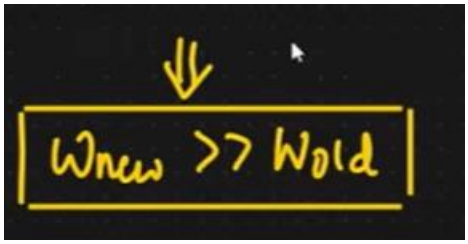


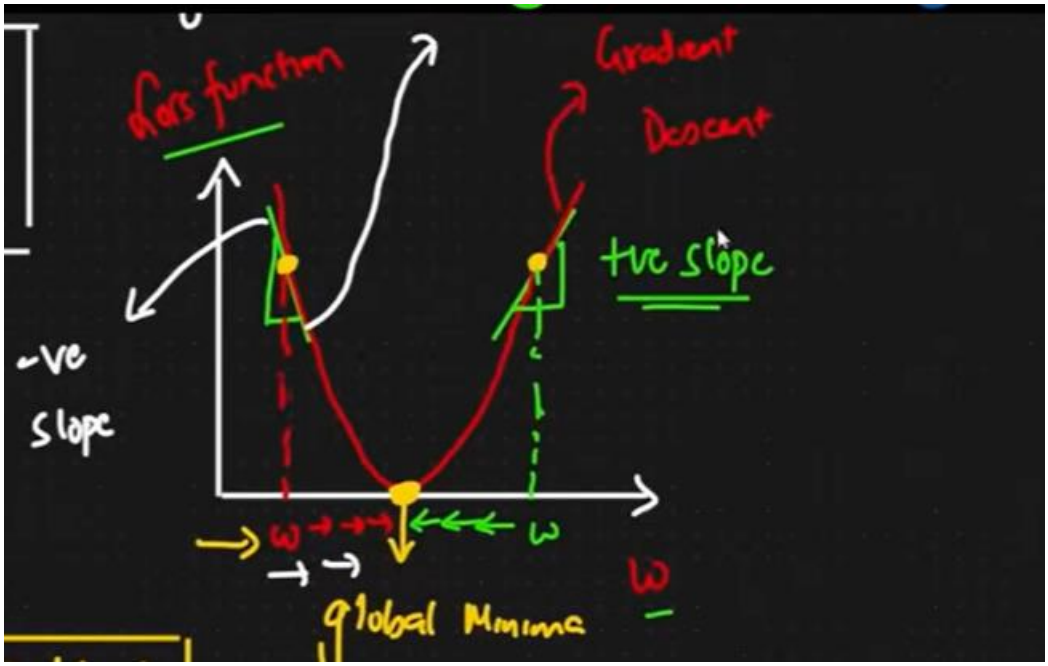


Tangent line to find slope if the line going downwards then it is negative slope we need to increase the weights to achieve global minima (i.e., minimal loss function)









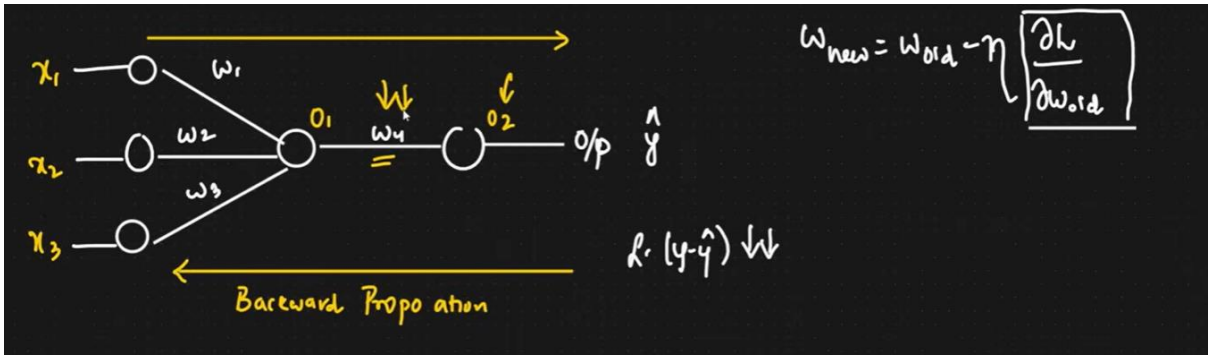
We need to decrease the weight to achive global minima when slope is positive

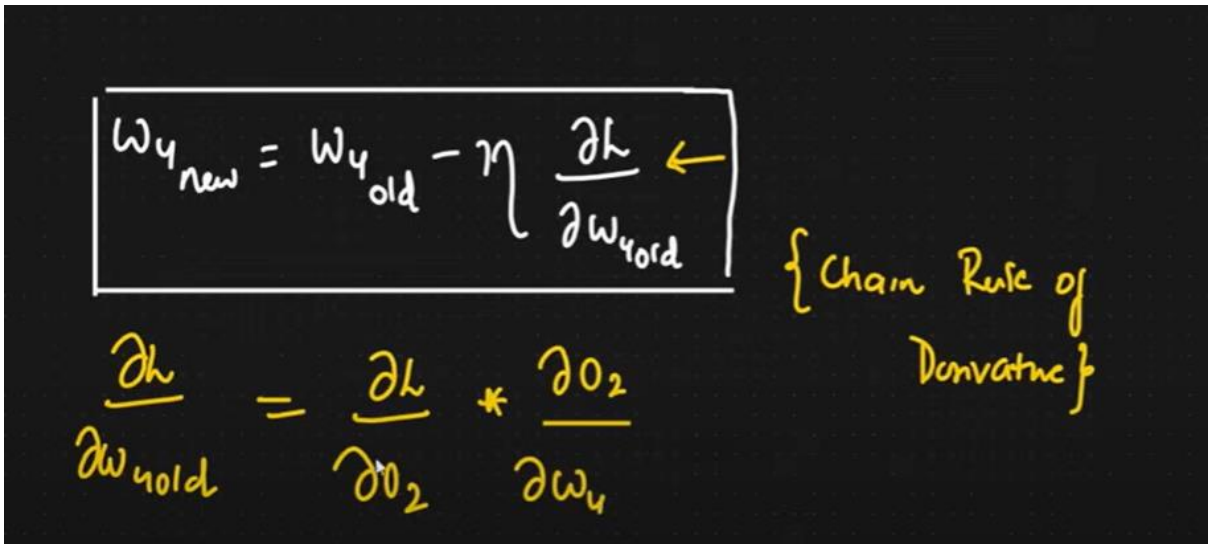
Learning rate should be always be a small number to slowly achieve global minima

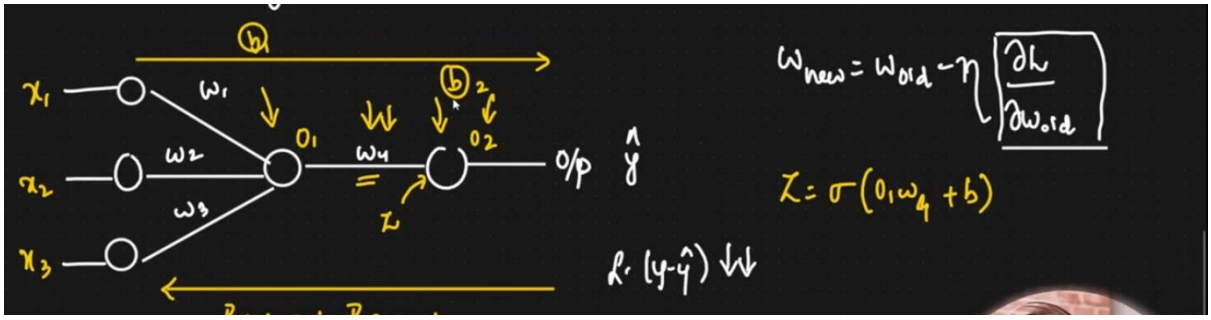
Learning rate can be round =0.01

Bigger number can take higher steps and vary here and after

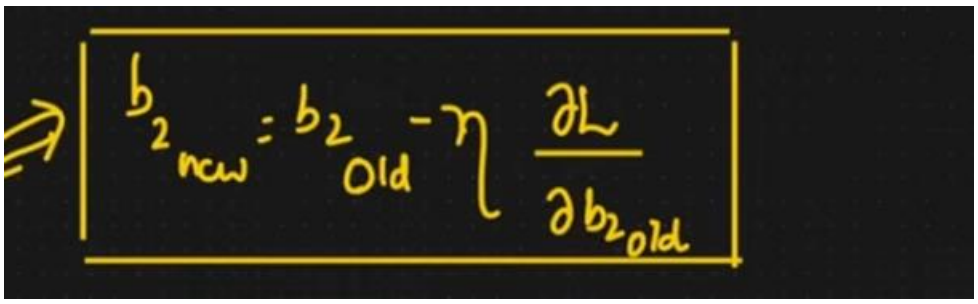
Chain rule of differentiation

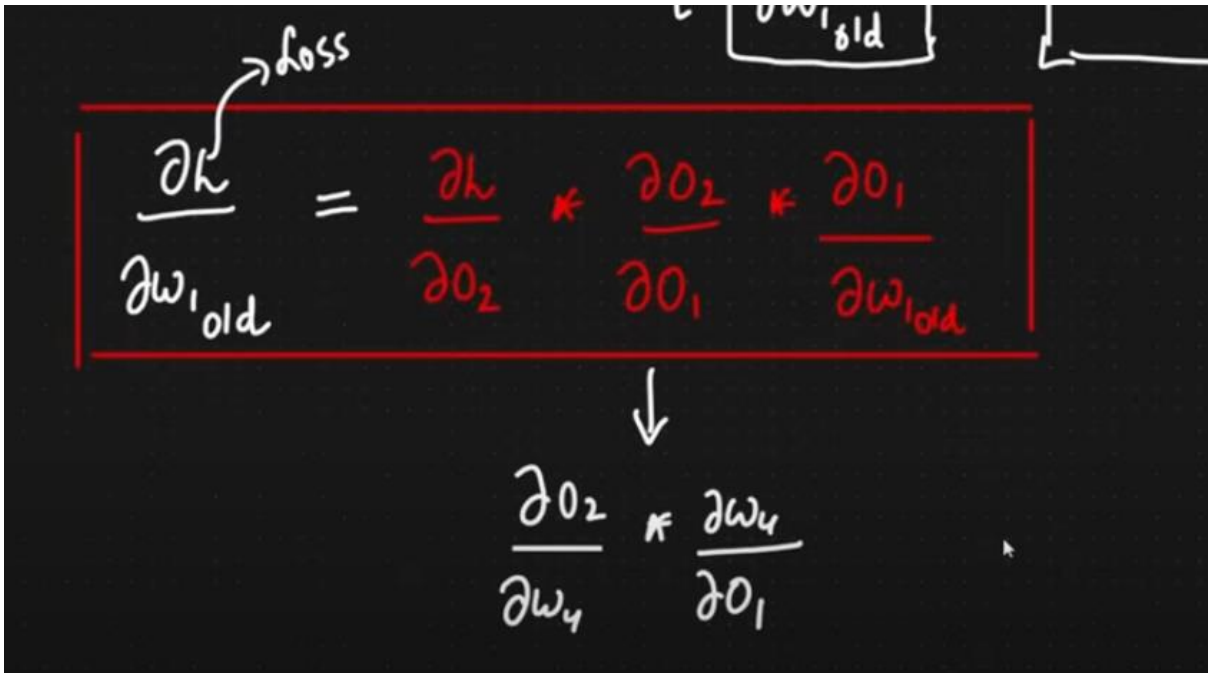


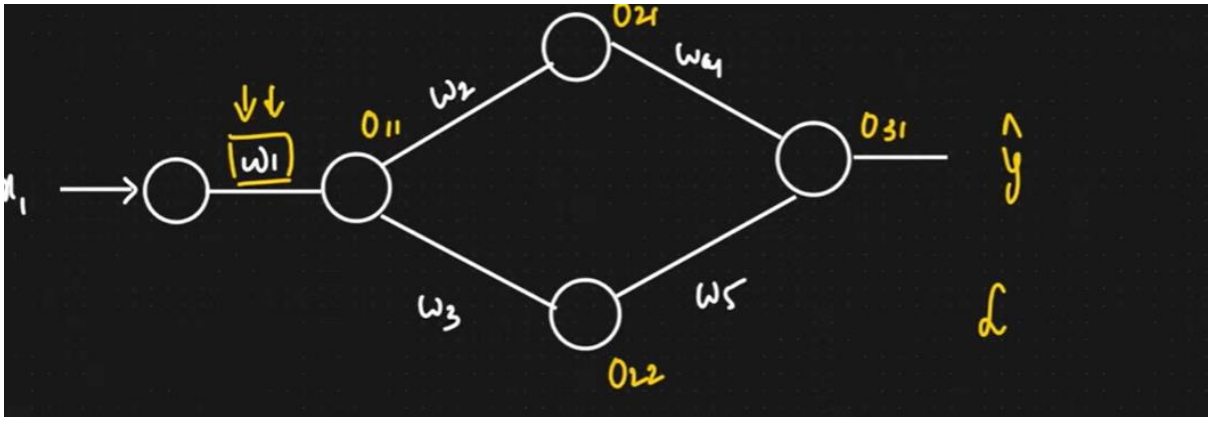




Derivative of loss function is dependent on o2 and o2 is dependent on o1 and o1 is dependent on w1 etc, it is a chain kind of thing




loss
$\frac{\partial L}{\partial w_{1_{old}}} = \frac{\partial L}{\partial O_2} * \frac{\partial O_2}{\partial O_1} * \frac{\partial O_1}{\partial w_{1_{old}}}$
$\frac{\partial O_2}{\partial w_4} * \frac{\partial w_4}{\partial O_1}$

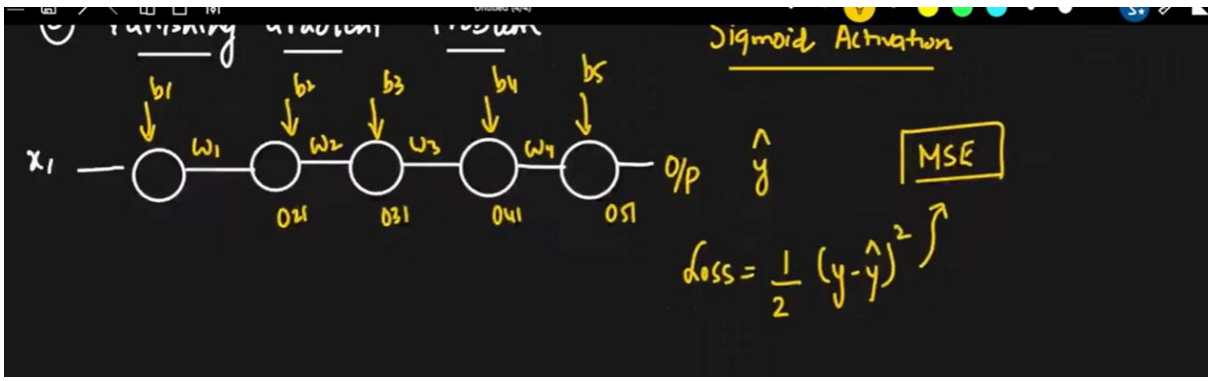

O_{21}
w_4
w_2
O_{11}
w_1
O_{31}
$\hat{y}$
x_1
w_3
w_5
L
O_{22}

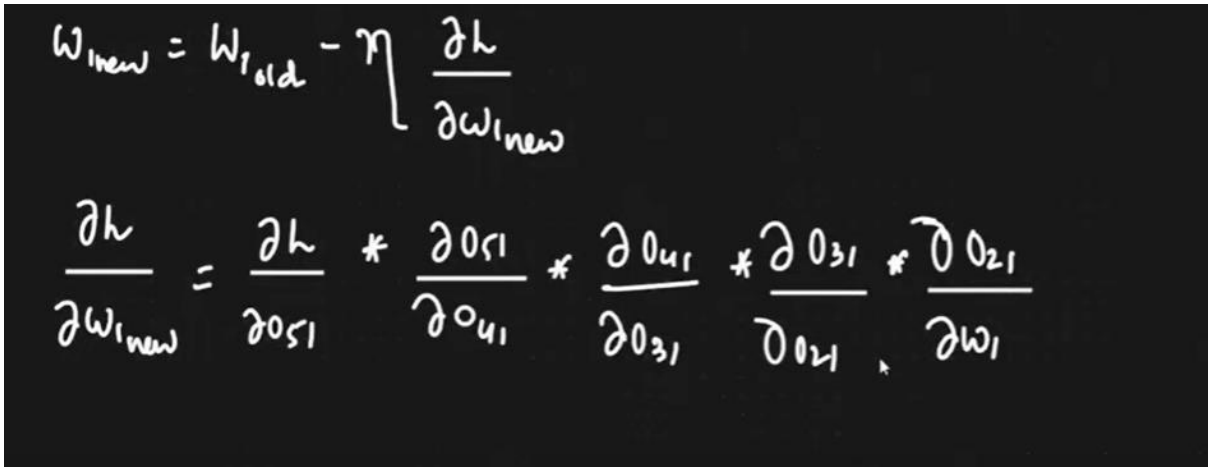
$$W_{1\,new} = W_{1\,old} - \eta \frac{\partial L}{\partial W_{1\,old}}$$

$$\frac{\partial L}{\partial W_{1\,old}} = \left[\frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial O_{21}} * \frac{\partial O_{21}}{\partial O_{11}} * \frac{\partial O_{11}}{\partial W_{1\,old}}\right] + \left[\frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial O_{22}} * \frac{\partial O_{22}}{\partial O_{11}} * \frac{\partial O_{11}}{\partial W_{1\,old}}\right]$$

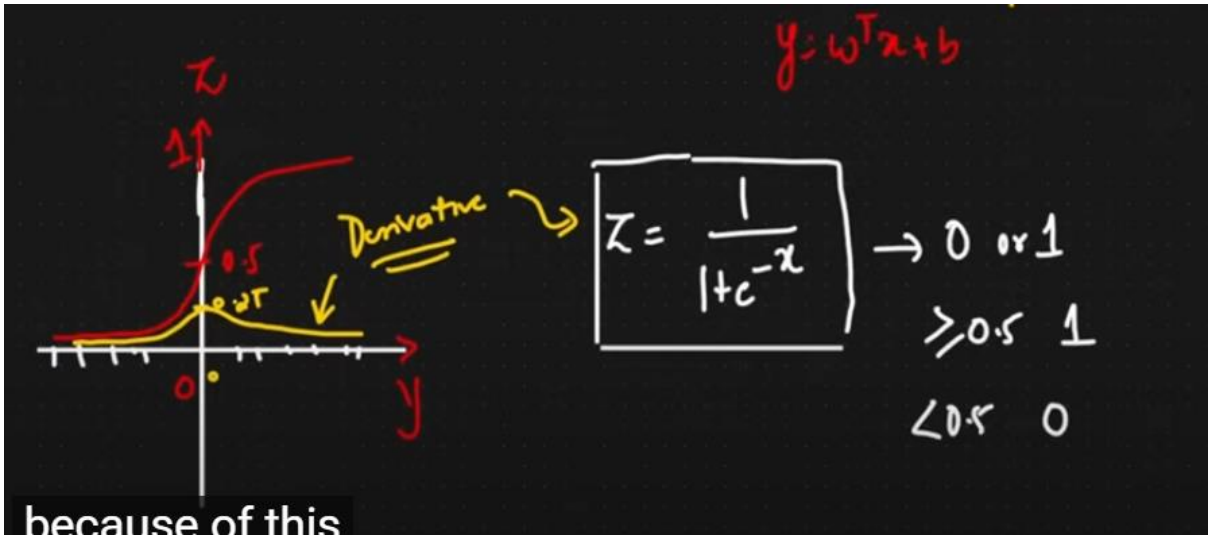
Chain rule of derivates

Vanishing gradient descent programe

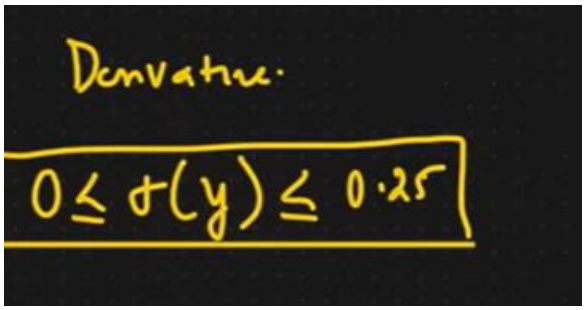


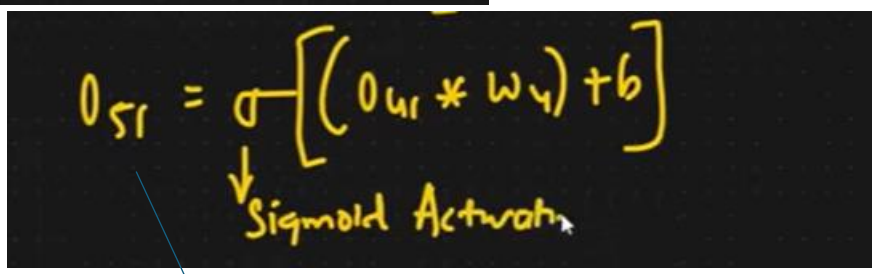


Sigmoid activation function

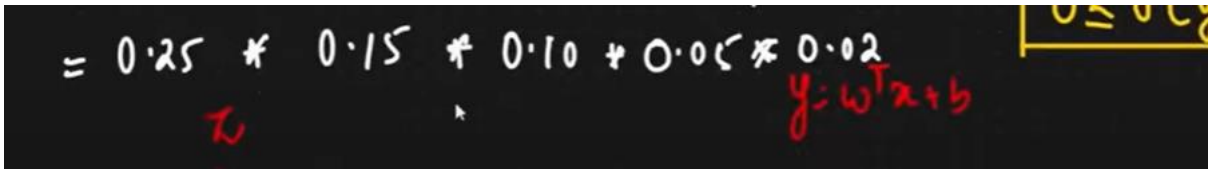


Derivative of sigmoid function ranging from 0 to 0.25

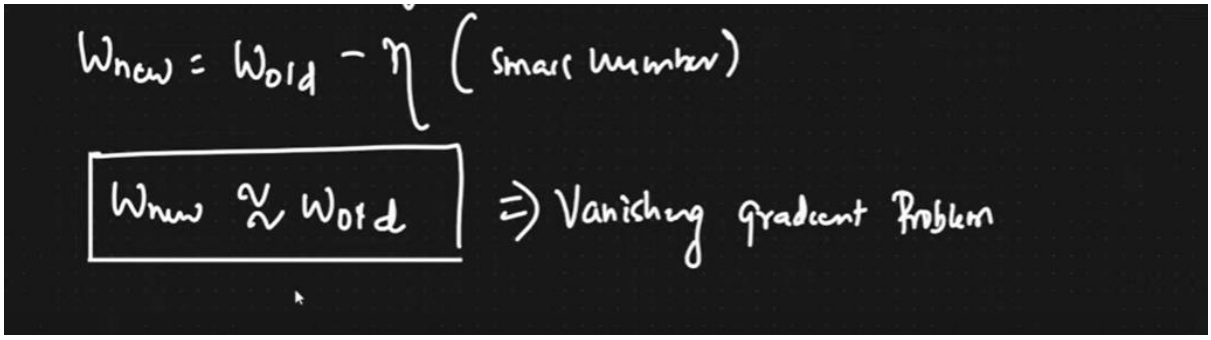




The value will range from 0 to 0.25 because we multiply with sigmoid activation function

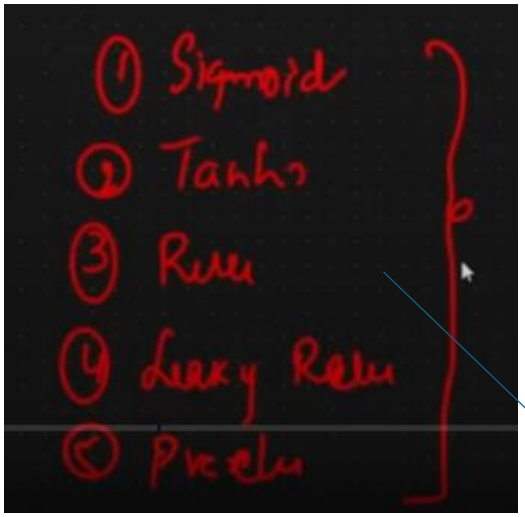


If we keep compute like this and multiply with sigmoid activation function then values will get smaller what we do next then we will update our weights



If there is no change in weights it is called vanishing gradient problem

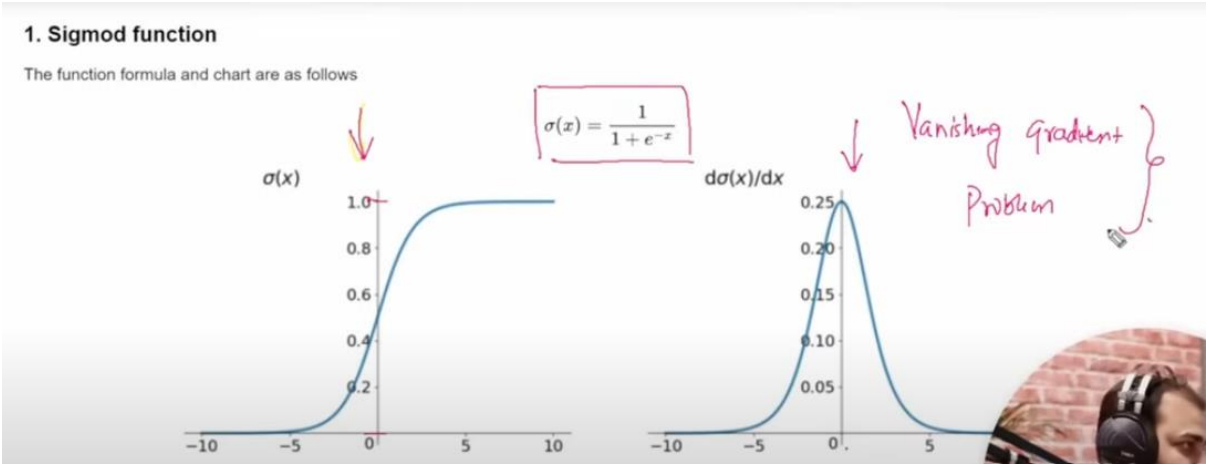
To solve this problem use another activation function



different types of activation function

1. Sigmod function

The function formula and chart are as follows



1) When the input is slightly away from the coordinate origin, the gradient of the function becomes very small, almost zero. In the process of neural network backpropagation, we all use the chain rule of differential to calculate the differential of each weight w. When the backpropagation passes through the sigmod function, the differential on this chain is very small. Moreover, it may pass through many sigmod functions, which will eventually cause the weight w to have little effect on the loss function, which is not conducive to the optimization of the weight. This The problem is called gradient saturation or gradient dispersion.

2) The function output is not centered on 0, which will reduce the efficiency of weight update.

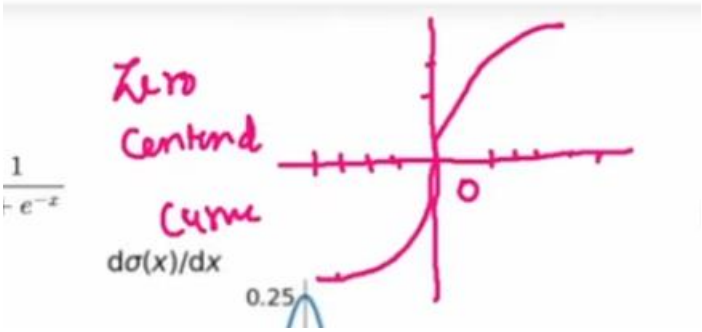
3) The sigmod function performs exponential operations, which is slower for computers.

Advantages of Sigmoid Function : -

1. Smooth gradient, preventing "jumps" in output values.
2. Output values bound between 0 and 1, normalizing the output of each neuron.
3. Clear predictions, i.e very close to 1 or 0.

Sigmoid has three major disadvantages:

- Prone to gradient vanishing
- Function output is not zero-centered
- Power operations are relatively time consuming



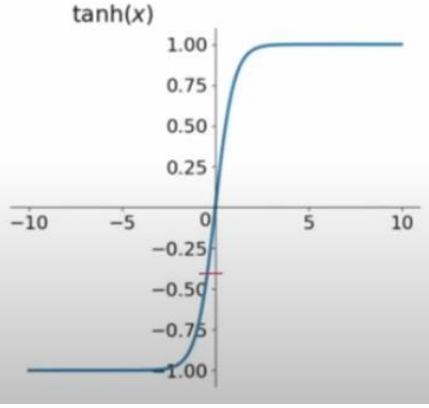
If we have zero centred curve then weight updation becomes very faster

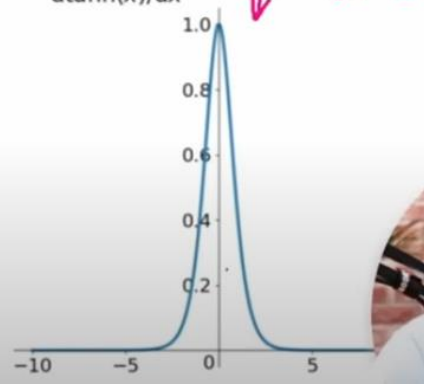
2. tanh function

The tanh function formula and curve are as follows

-1 to 1

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$





Tanh is a hyperbolic tangent function. The curves of tanh function and sigmod function are relatively similar. Let 's compare them. First of all, when th large or small, the output is almost smooth and the gradient is small, which is not conducive to weight update. The difference is the output interval.

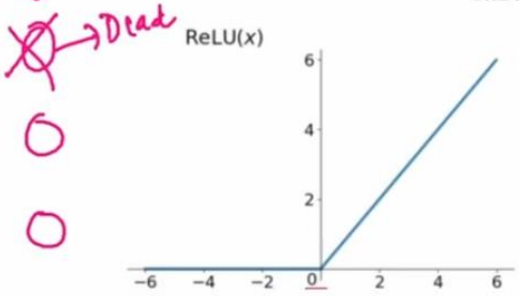
The output interval of tanh is 1), and the whole function is 0-centric, which is better than sigmod.

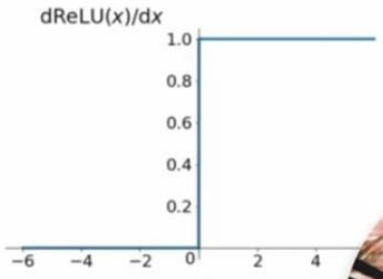
In general binary classification problems, the tanh function is used for the hidden layer and the sigmod function is used for the output laye not static, and the specific activation function to be used must be analyzed according to the specific problem, or it depends on debuggi

3. ReLU function

ReLU function formula and curve are as follows

$$\text{ReLU} = \max(0, x)$$





The ReLU function is actually a function that takes the maximum value. Note that this is not fully interval-derivable, but we can tak
the figure above. Although ReLU is simple, it is an important achievement in recent years.

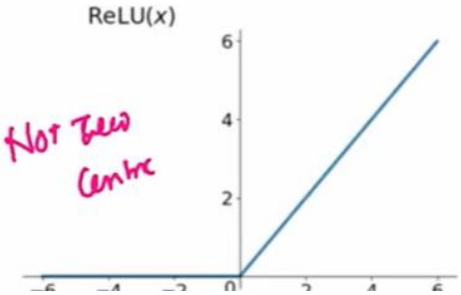
The ReLU (Rectified Linear Unit) function is an activation function that is currently more popular. Compared with the sigmod functio
has the following advantages:

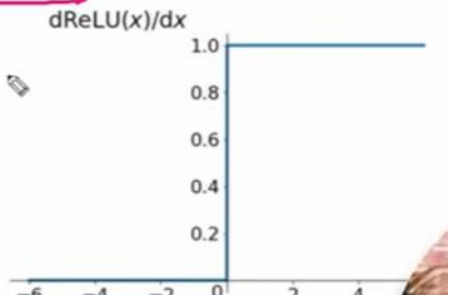
1) When the input is positive, there is no gradient saturation problem.

3. ReLU function

ReLU function formula and curve are as follows

$$\text{ReLU} = \max(0, x)$$





The ReLU (Rectified Linear Unit) function is an activation function that is currently more popular. Compared with the sigmod function and the tanh function, it has the following advantages:

1) When the input is positive, there is no gradient saturation problem.

2) The calculation speed is much faster. The ReLU function has only a linear relationship. Whether it is forward or backward, it is much faster than sigmod and tanh. (Sigmod and tanh need to calculate the exponent, which will be slower.)

Ofcourse, there are disadvantages:

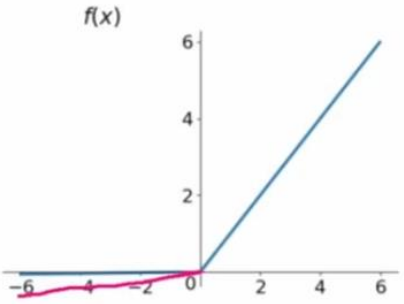
1) When the input is negative, ReLU is completely inactive, which means that once a negative number is entered, ReLU will die. In this way, in the forward propagation process, it is not a problem. Some areas are sensitive and some are insensitive. But in the backpropagation process, if you enter a negative number, the gradient will be completely zero, which has the same problem as the sigmod function and tanh function.

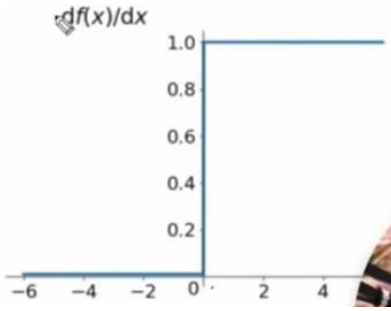
2) We find that the output of the ReLU function is either 0 or a positive number, which means that the ReLU function is not a 0-centric functi

4. Leaky ReLU function

Solve dead Neuron (Relu)

$$f(x) = \max(0.01x, x)$$



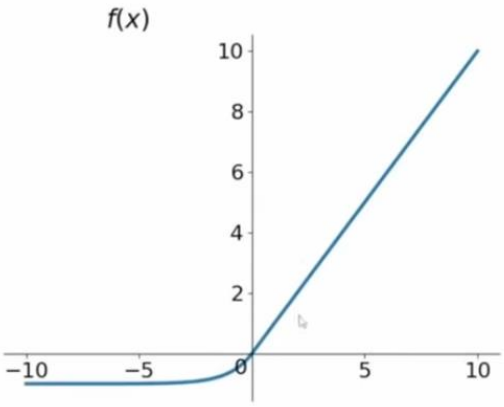


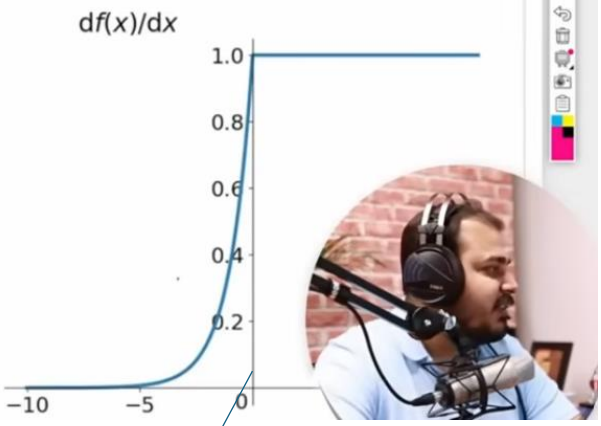
In order to solve the Dead ReLU Problem, people proposed to set the first half of ReLU 0.01x instead of 0. Another intuitive idea is a parameter-based method, Parametric ReLU : f(x)= max(alpha x,x), which alpha can be learned from back propagation. In theory, Leaky ReLU has all the advantages of ReLU, plus there will be no problems with Dead ReLU, but in actual operation, it has not been fully proved that Leaky ReLU is always better than ReLU.

5. ELU (Exponential Linear Units) function ¶

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$$

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$$





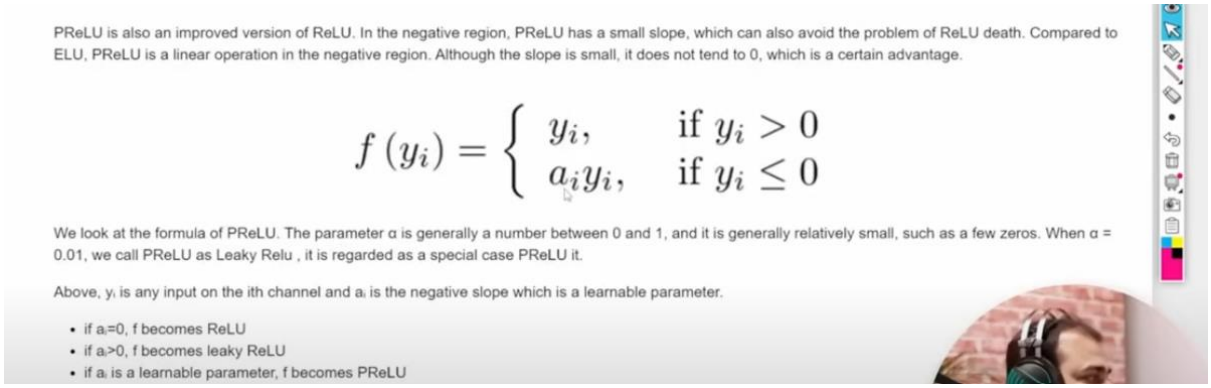
Values ranging from 0 to 1

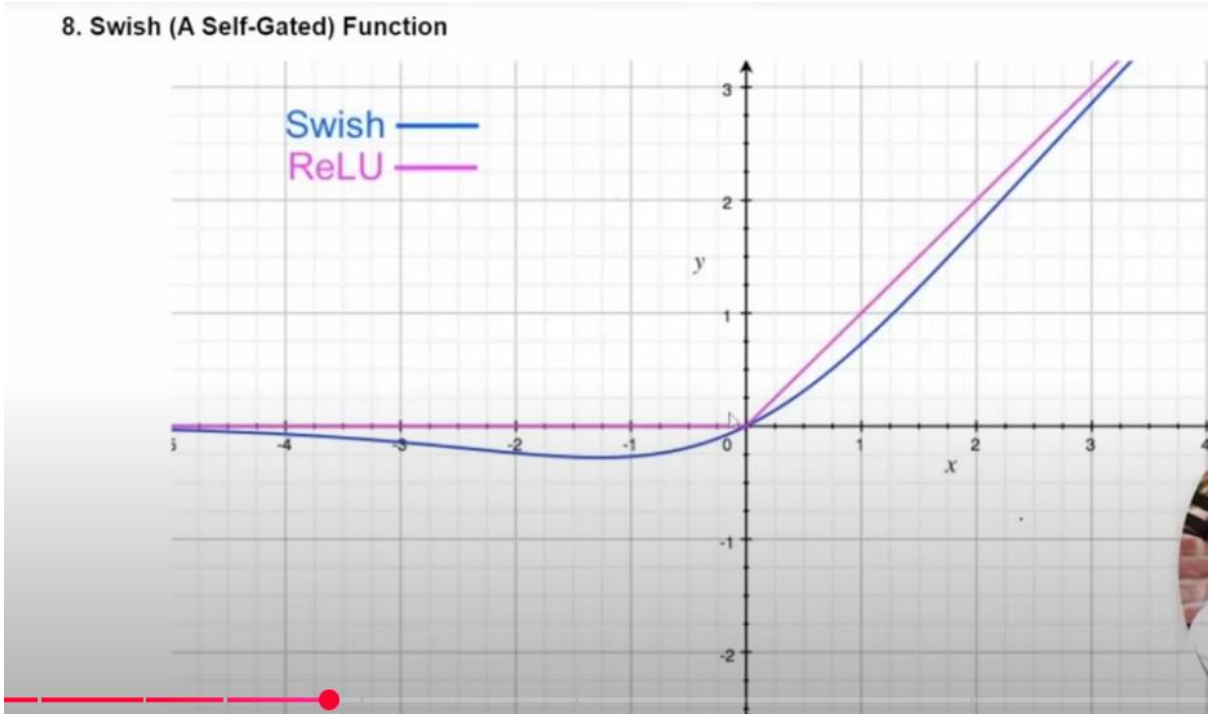
ELU is also proposed to solve the problems of ReLU. Obviously, ELU has all the advantages of ReLU, and:

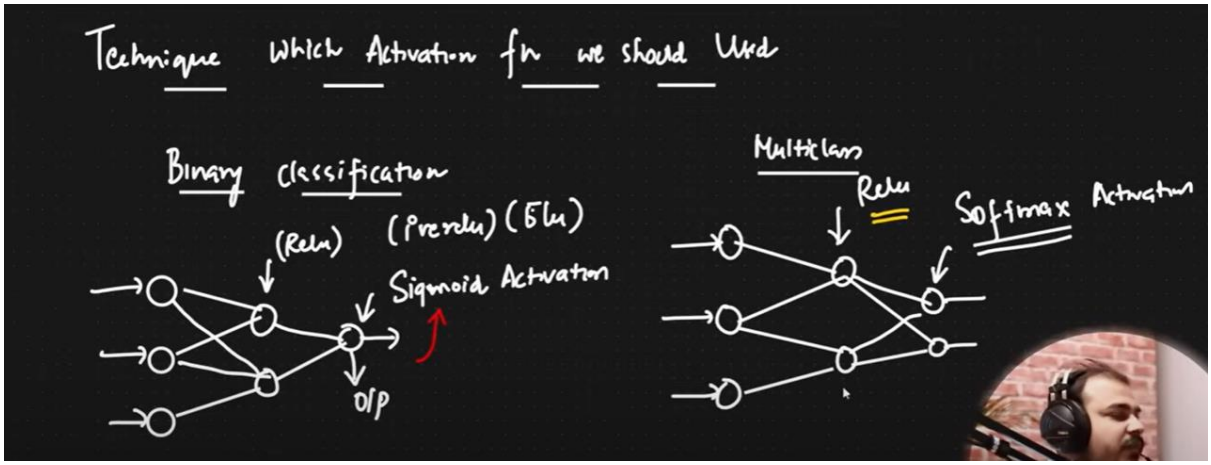
- No Dead ReLU issues
- The mean of the output is close to 0, zero-centered

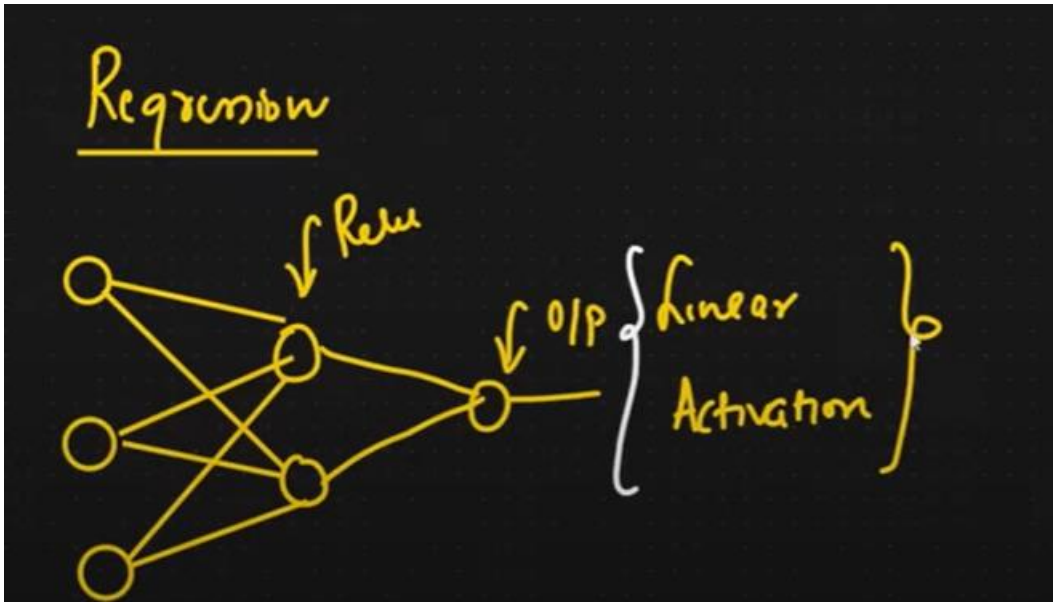
One small problem is that it is slightly more computationally intensive. Similar to Leaky ReLU, although theoretically better than ReLU, there is currently no good evidence in practice that ELU is always better than ReLU.

prelu

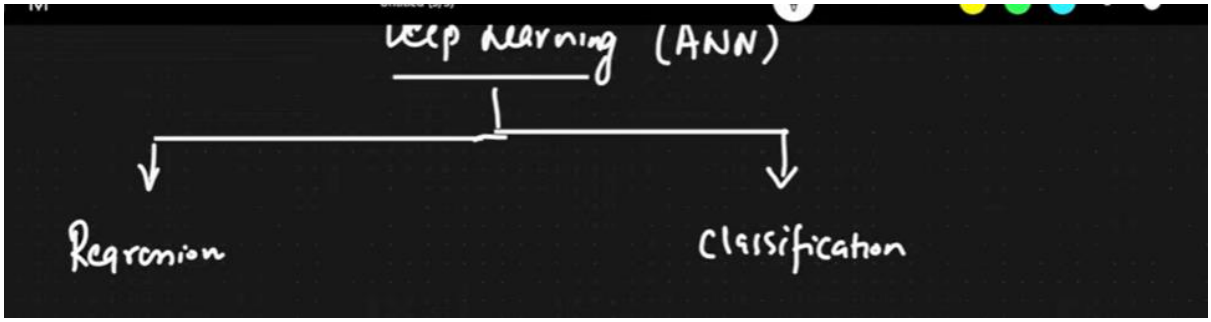

PReLU is also an improved version of ReLU. In the negative region, PReLU has a small slope, which can also avoid the problem of ReLU death. Compared to ELU, PReLU is a linear operation in the negative region. Although the slope is small, it does not tend to 0, which is a certain advantage.
$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}$$
We look at the formula of PReLU. The parameter α is generally a number between 0 and 1, and it is generally relatively small, such as a few zeros. When α = 0.01, we call PReLU as Leaky Relu , it is regarded as a special case PReLU it.
Above, y_i is any input on the ith channel and a_i is the negative slope which is a learnable parameter.
• if a_i=0, f becomes ReLU
• if a_i>0, f becomes leaky ReLU
• if a_i is a learnable parameter, f becomes PReLU


8. Swish (A Self-Gated) Function
Swish
ReLU
y
x

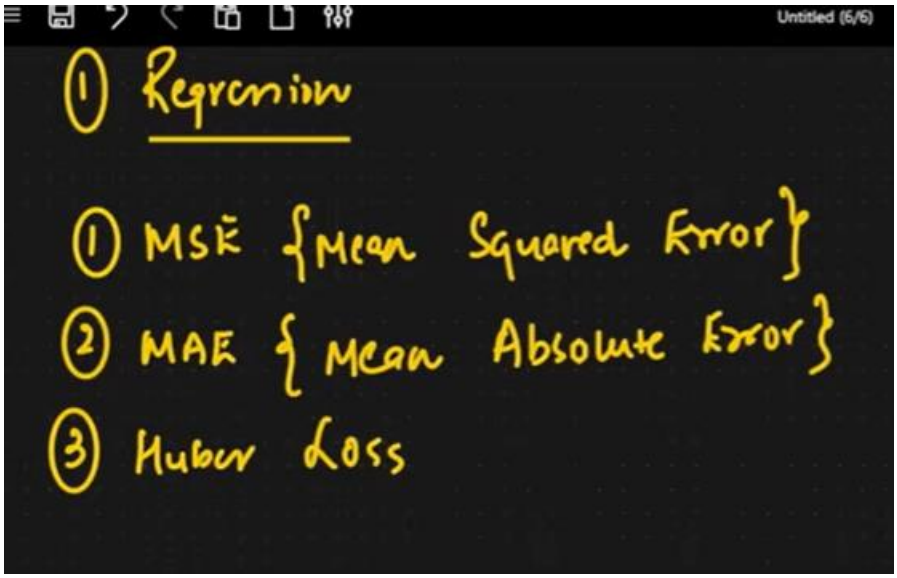

Technique Which Activation fn we should Used
Binary classification
(Relu) (Prelu) (Elu)
Sigmoid Activation
O/P
Multiclass
Relu
Softmax Activation

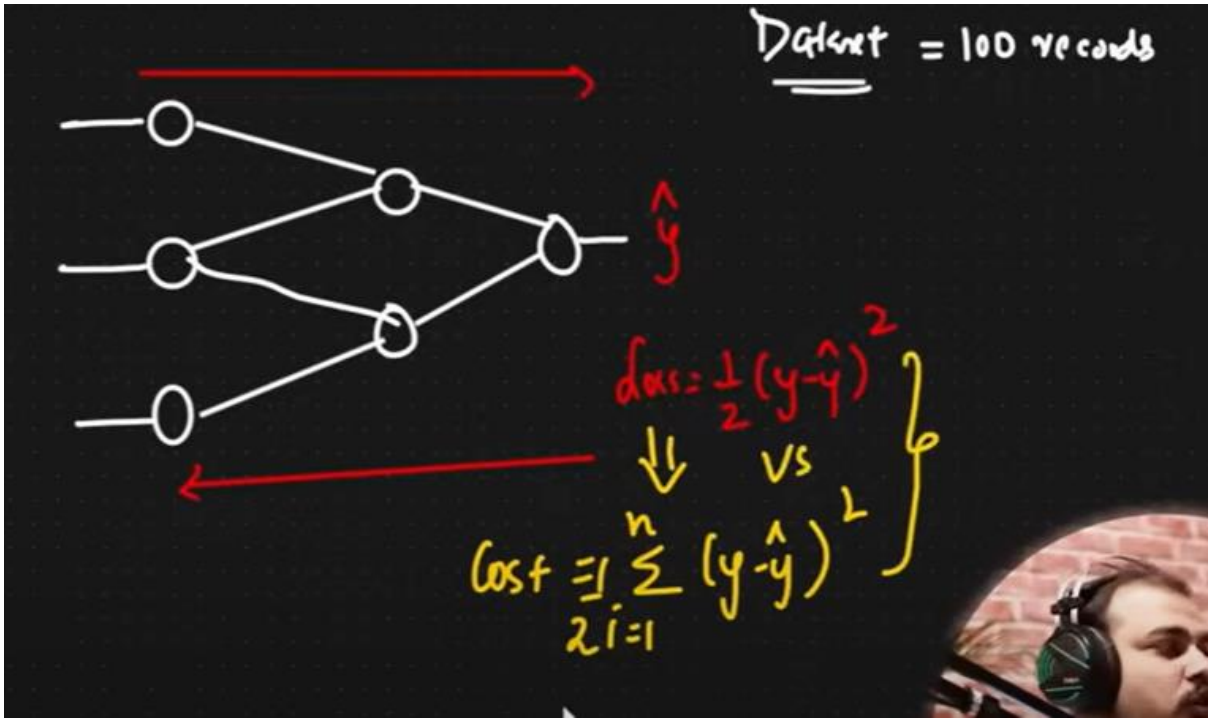


Loss function

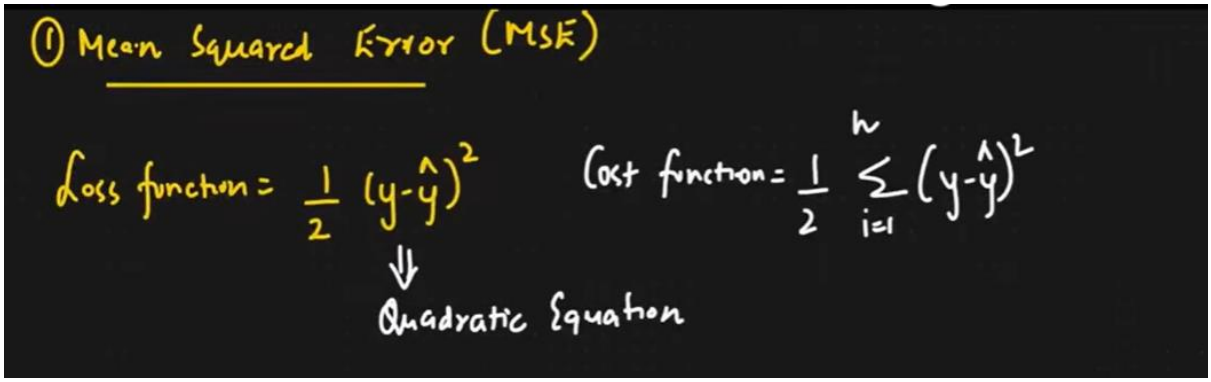


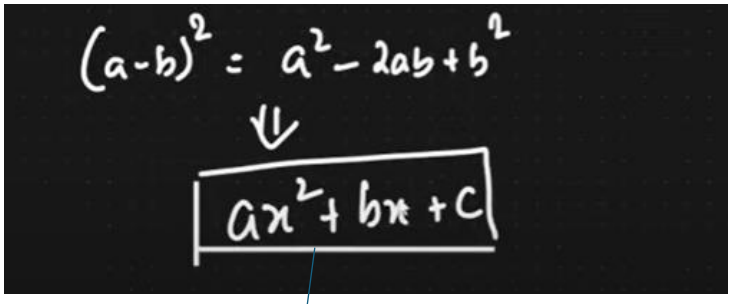

Regression
O/P
Exp
Degree
Salary
10
phd
Regression Problem
Classification
Dataset
Play
Study
Pass/Fail
10
2
Fail
4
3
Fail
5
5
Maybe
2
7
Pass
loss functions


Untitled (6/6)
① Regression
① MSE {Mean Squared Error}
② MAE {Mean Absolute Error}
③ Huber Loss

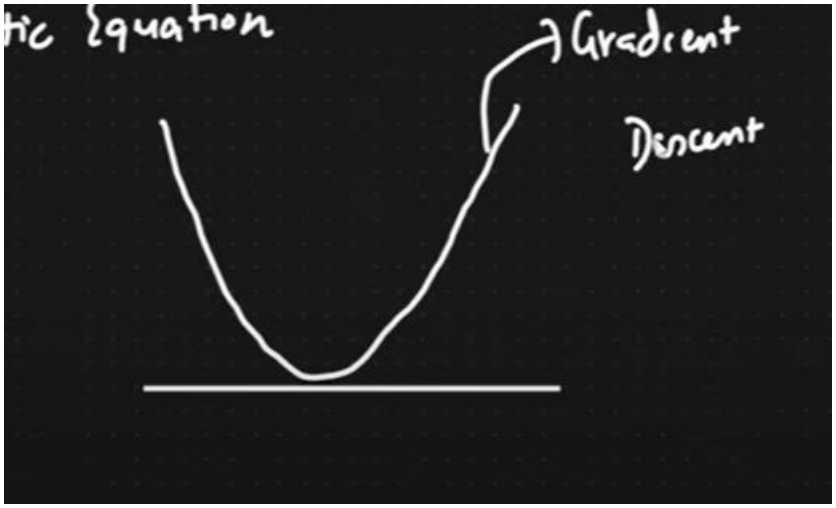


Epochs – we provide batch of datasets for each propagation

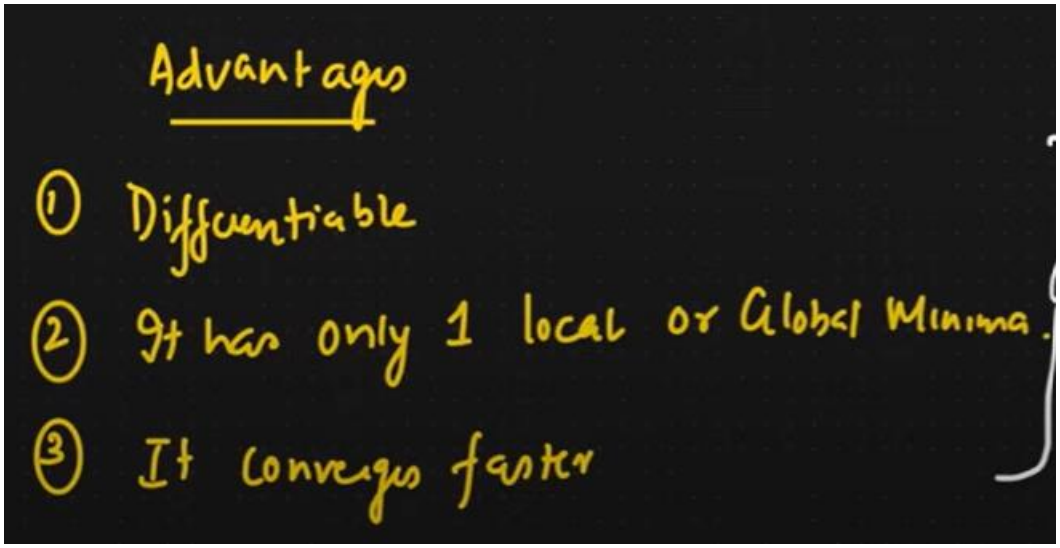


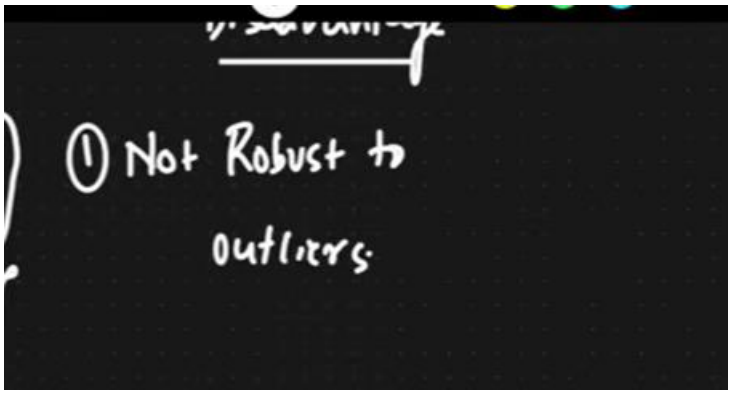


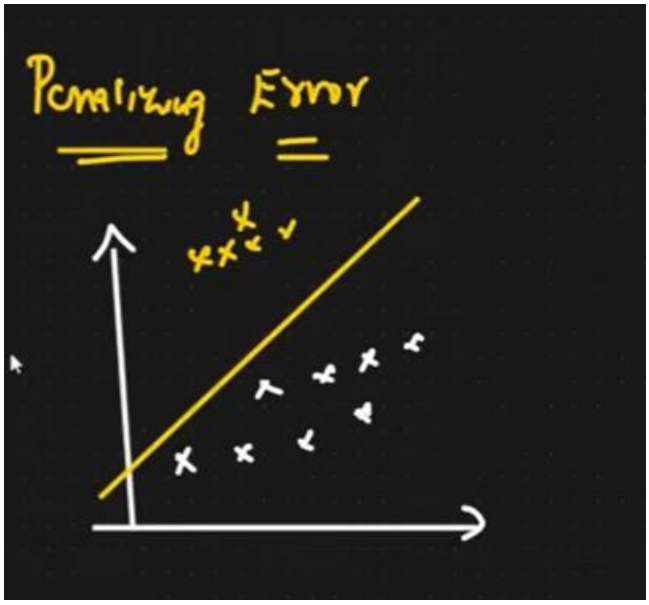
General equation of quadratic function

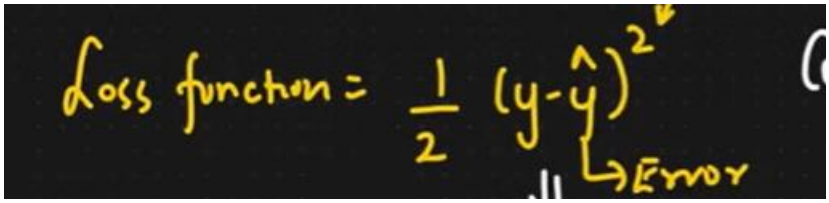


Quadratic curve it is giving some kind of gradient descent



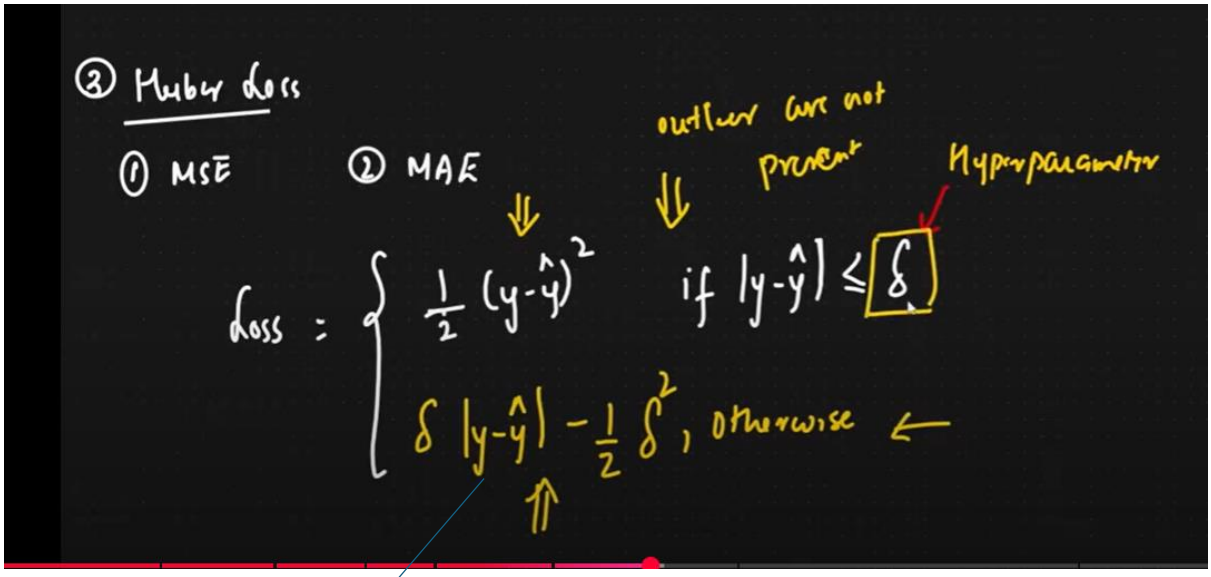




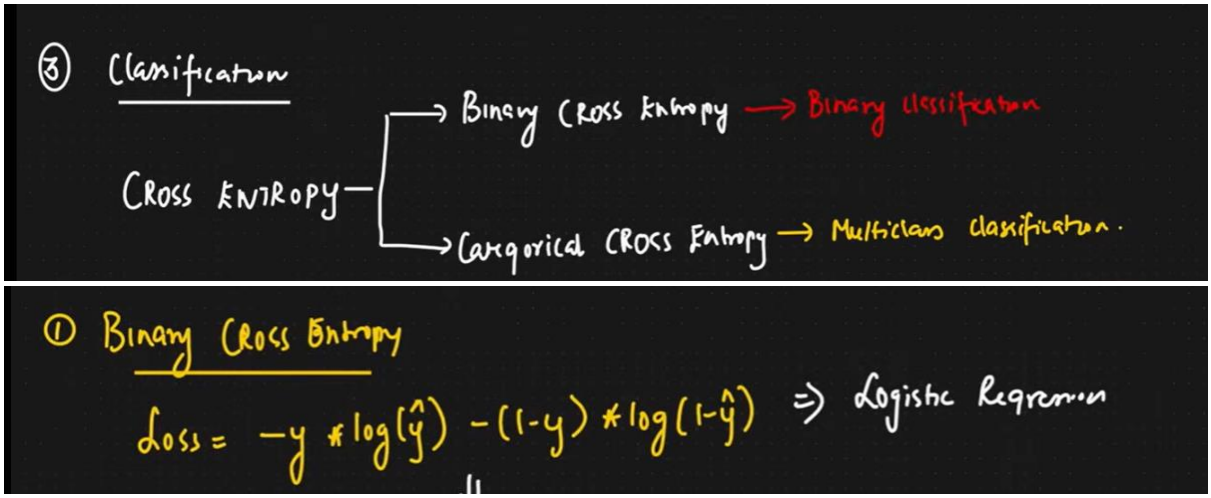


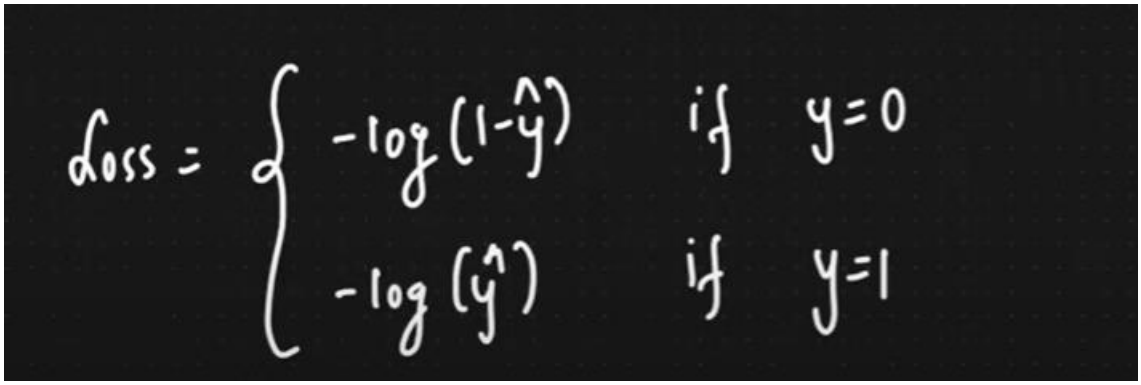
Penalizing the error with respect to error by squaring them

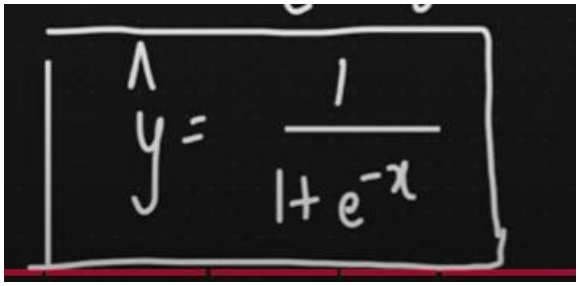
Sub gradient – time consuming divide the line part by part and compute the slope

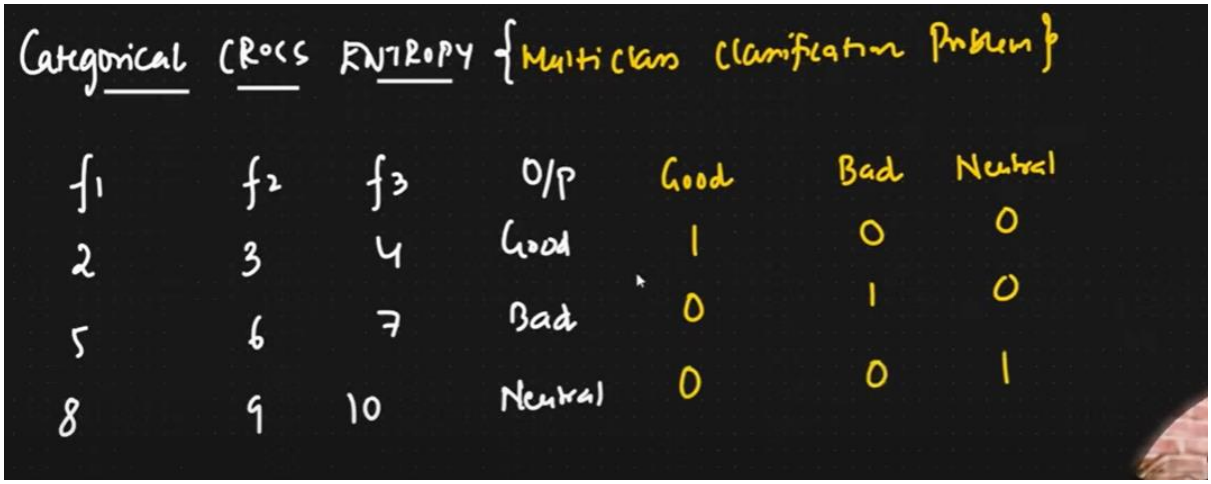


When outliers are present









In his for dependent variable we convert dep variable into one hot encoder

$$L(x_i, y_i) = -\sum_{j=1}^{c} y_{ij} * \ln(\hat{y}_{ij})$$

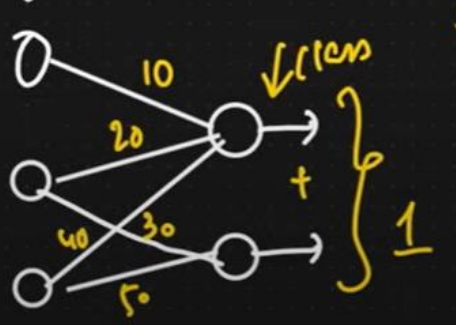
$y_i = [y_{i1}, y_{i2}, y_{i3} \cdots y_{ic}]$

$$y_{ij} = \begin{cases} 1 & \text{if the element is in class.} \\ 0 & \text{otherwise} \end{cases}$$

$\hat{y}_{ij}$ = Softmax Activation → O/p Layer

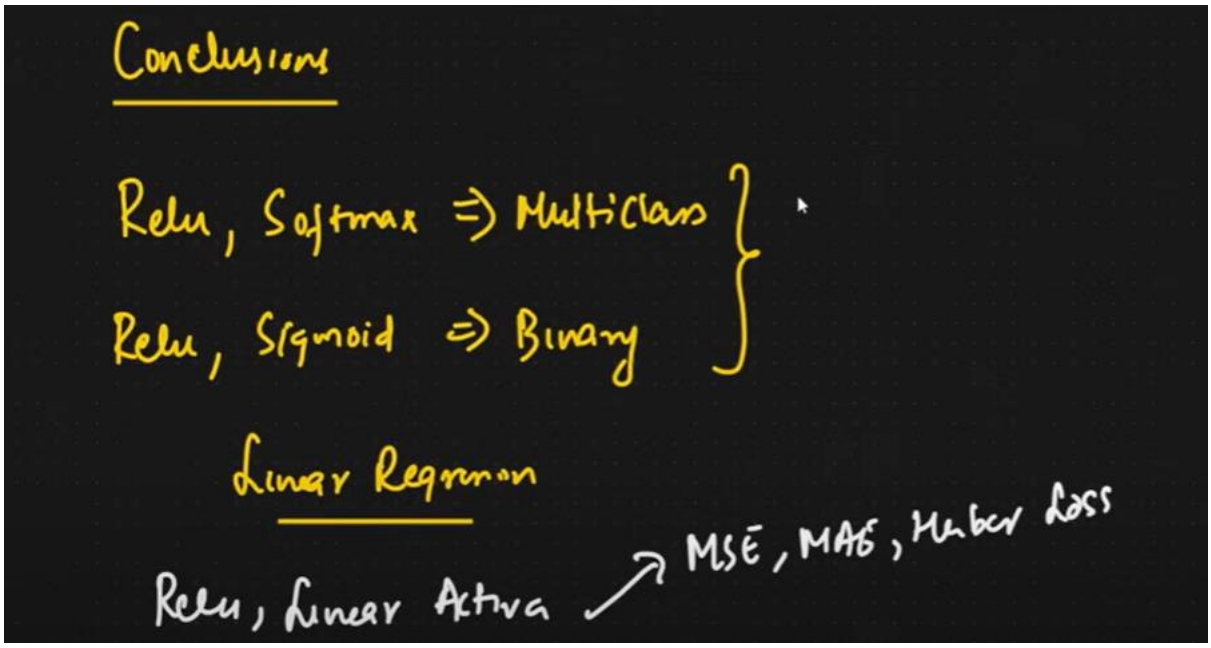
$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$ ⇒ Softmax Activation

0.4, 0.5, 0.6



$$= \frac{e^{10}}{e^{10+20+30+40+50}}$$ ⇒

When we sum the prob value from softmax activation function we get the value probability is equal to 1



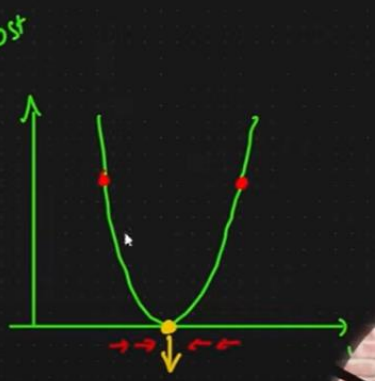
Batch, Epochs, Iterations

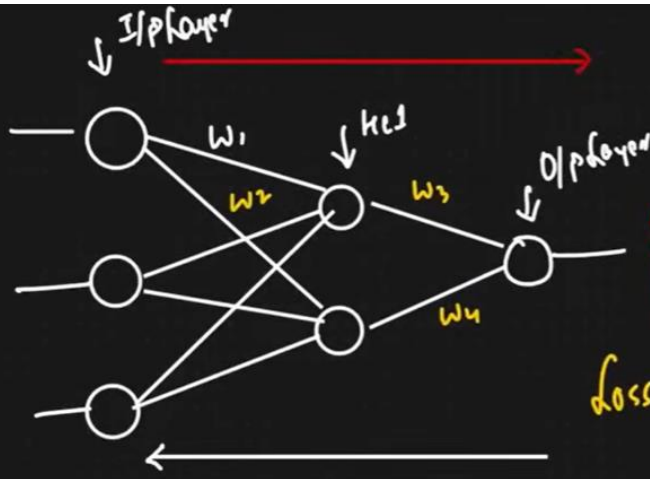
① GRADIENT DESCENT

Weight Updation Formula

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

Learning Rate





MSE ⇓ Optimizers

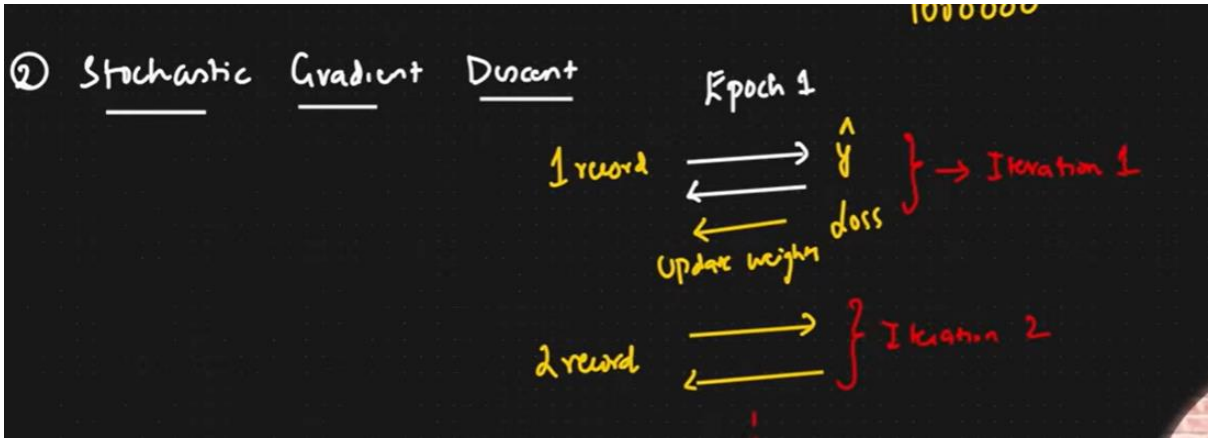
$$Loss = \frac{1}{2n} \sum_{i=1}^{n} (y - \hat{y})^2$$

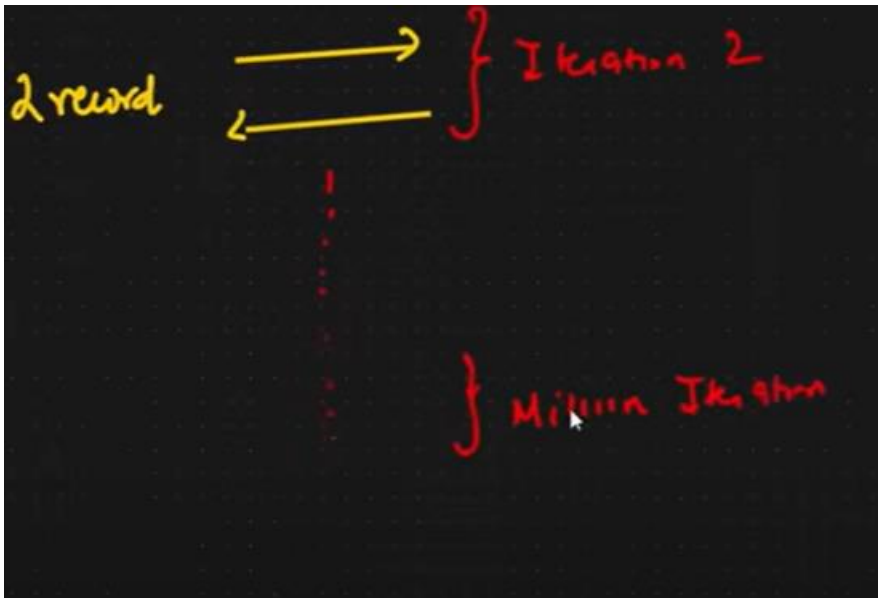
Cost function $\Rightarrow \frac{1}{2n} \sum_{i=1}^{n} (y - \hat{y})^2$ }

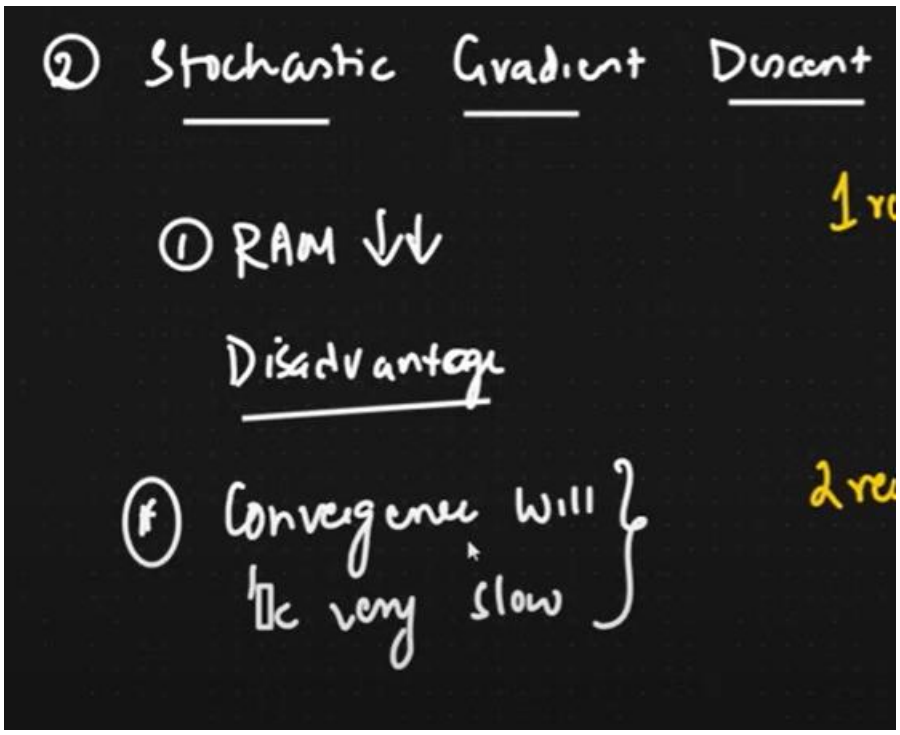
1000000

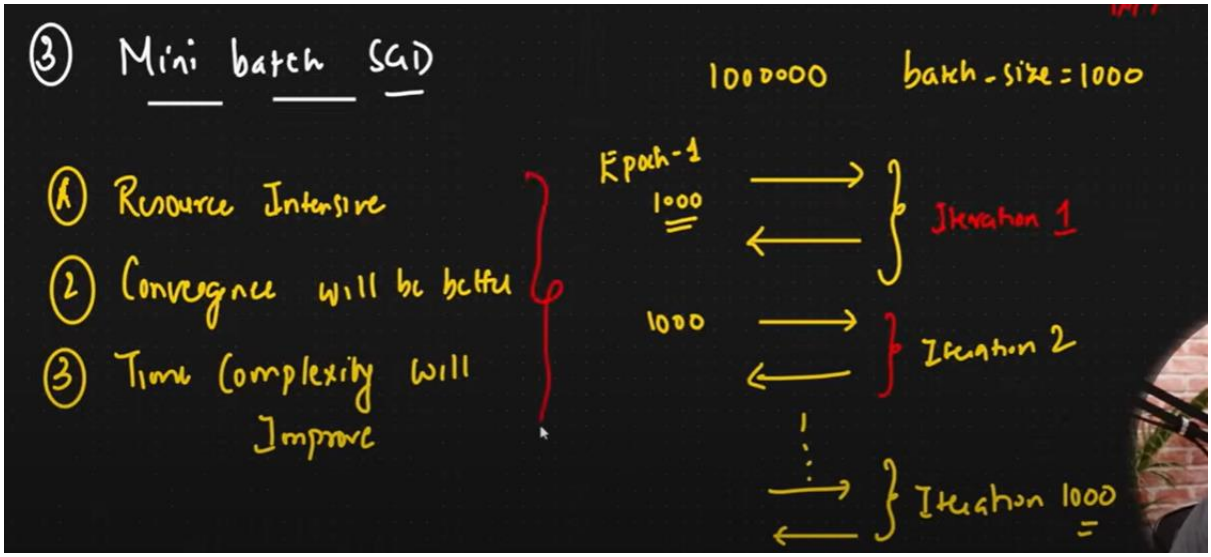
Epoch } 1 Epoch { 1000000 }

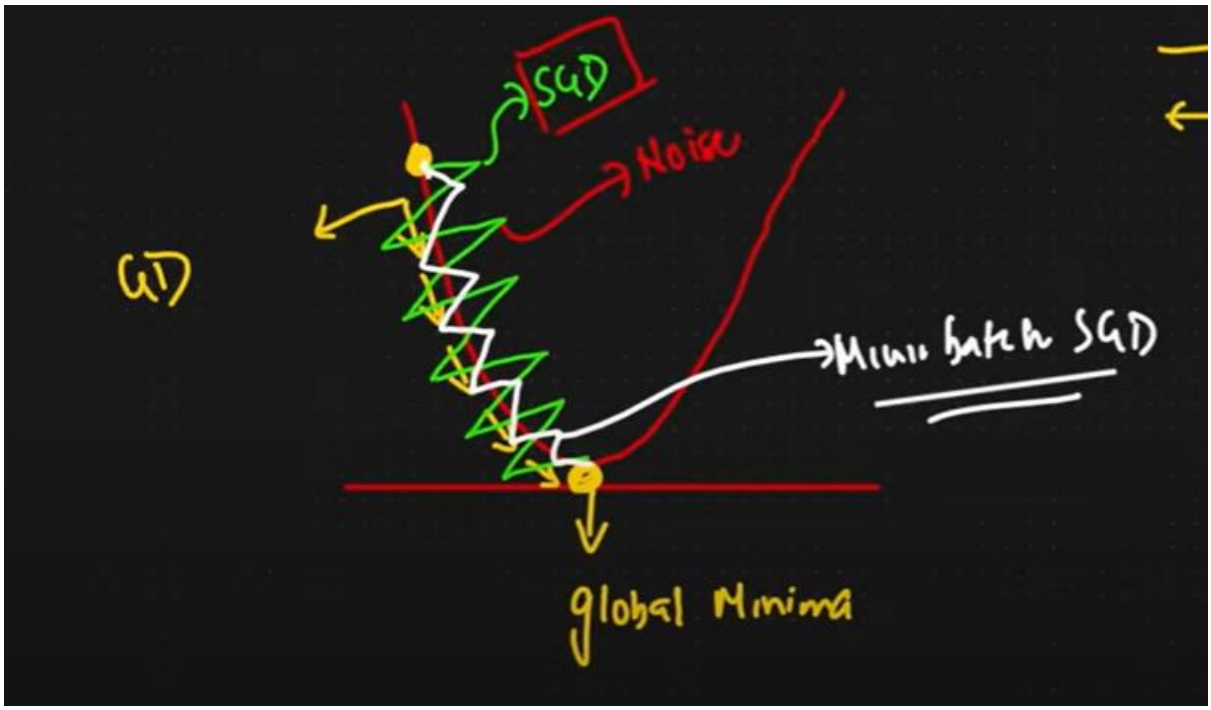
disadvantage


② Stochastic Gradient Descent
Epoch 1
1 record
ŷ
loss
Update weights
→ Iteration 1
2 record
Iteration 2


2 record
Iteration 2
Million Iteration


② Stochastic Gradient Descent
1 re
① RAM ↓↓
Disadvantage
2 re
② Convergence will be very slow





Sgd has highest noise

Gd has least noise

But mini batch has less noise

How do we remove the noise ?

We use concept called momentum

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

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

$$w_t = w_{t-1} - \eta \frac{\partial L}{\partial w_{t-1}}$$

Exponential Weighted Average

{Forecasting}

t_1 t_2 t_3 t_4 ... t_n

a_1 a_2 a_3 a_4 ... a_n

$\beta \Rightarrow$ Hyperparameter

β: 0 to 1

$\Downarrow$

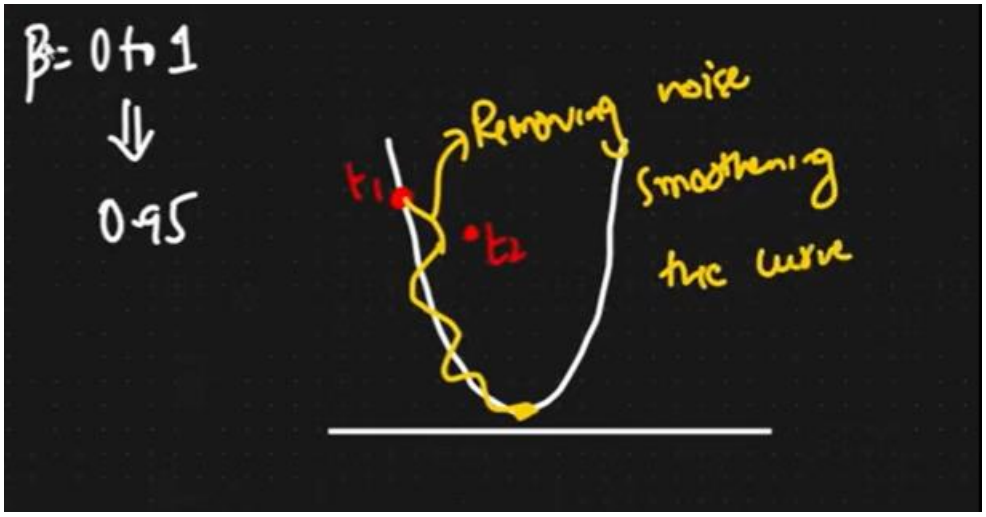
0.95

$V_{t_1} = a_1$

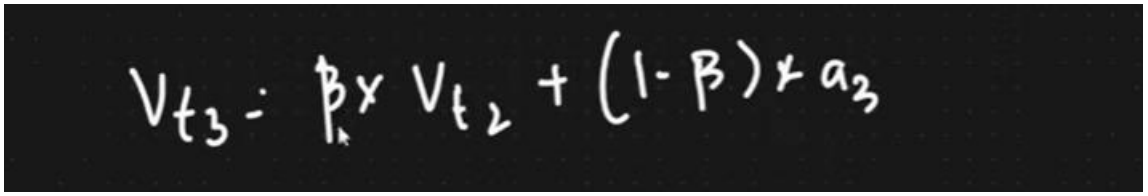
$V_{t_2} = \beta * V_{t_1} + (1-\beta) * a_2$

$= (0.95) * V_{t_1} + (0.05) * a_2$

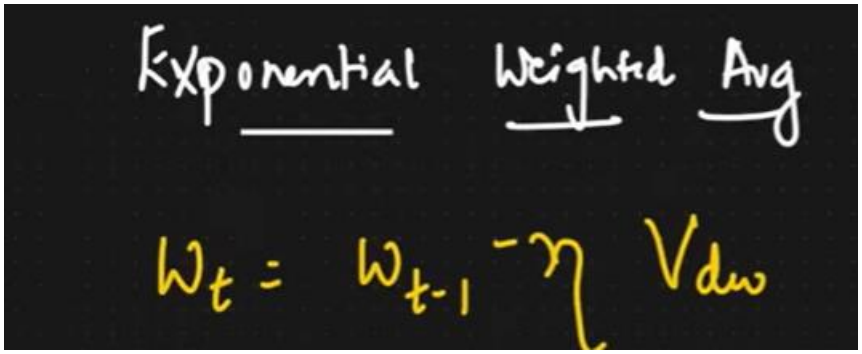
Beta is a hyper parameter which value of time stamp should focus on

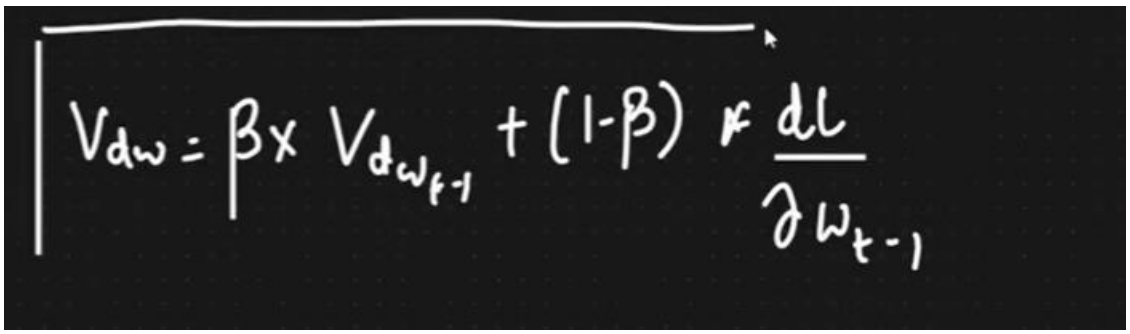


Bete hyperparameters helps you to smoothen (meaning removing the noise)

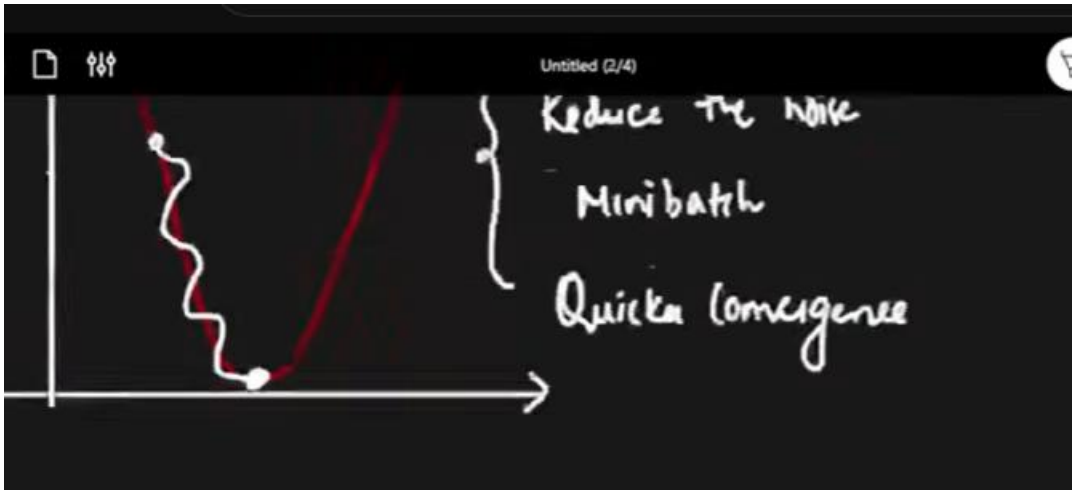


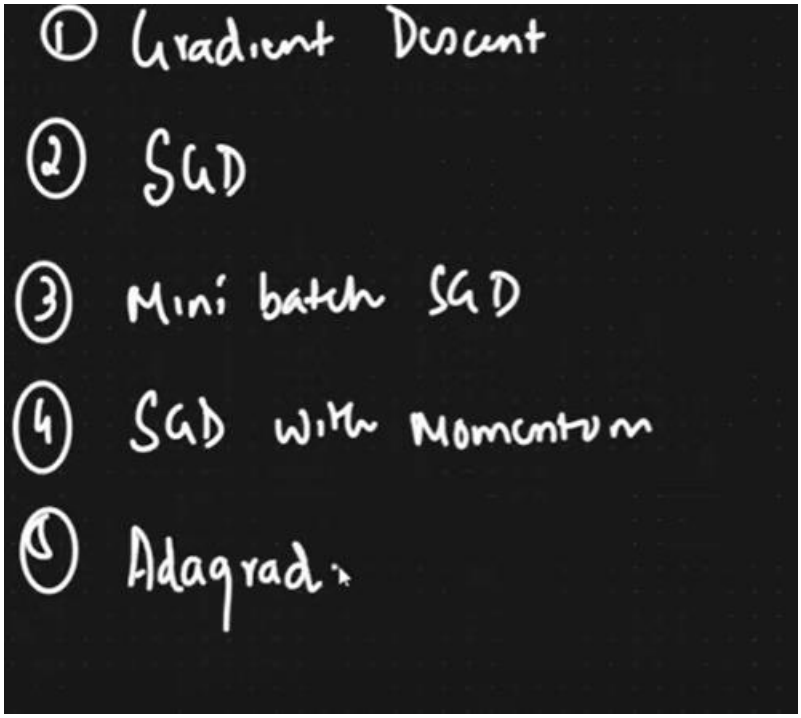
Where do I apply exponentially weighted average – we apply in derivative of loss function



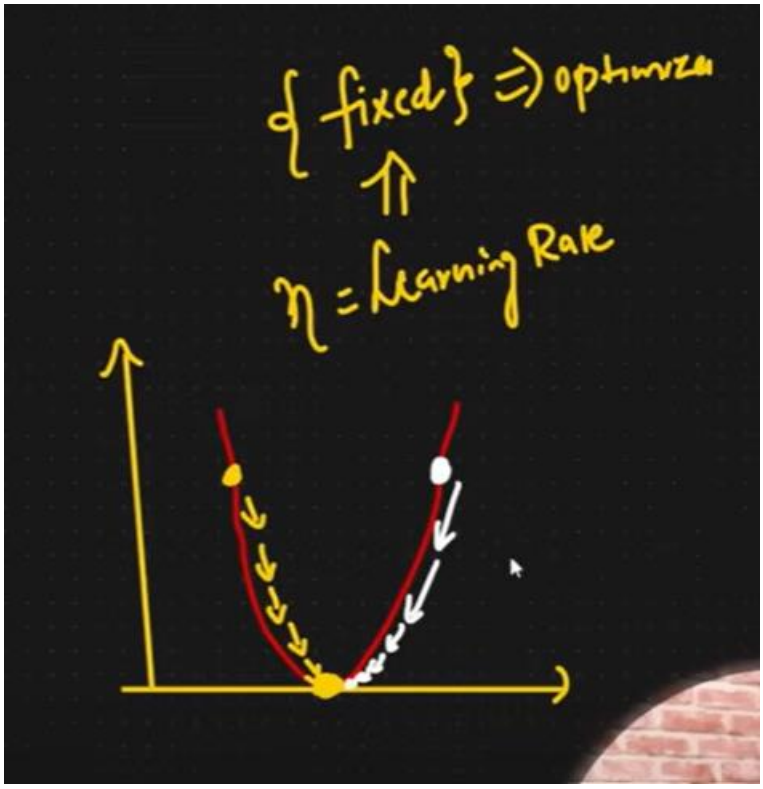


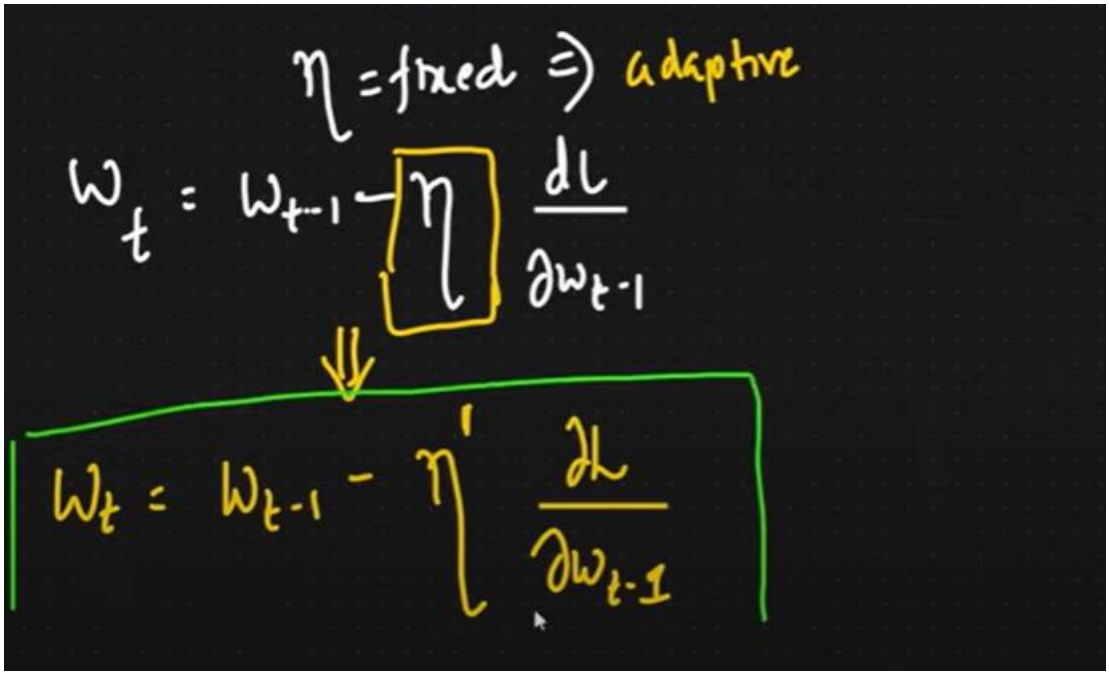
We have solved



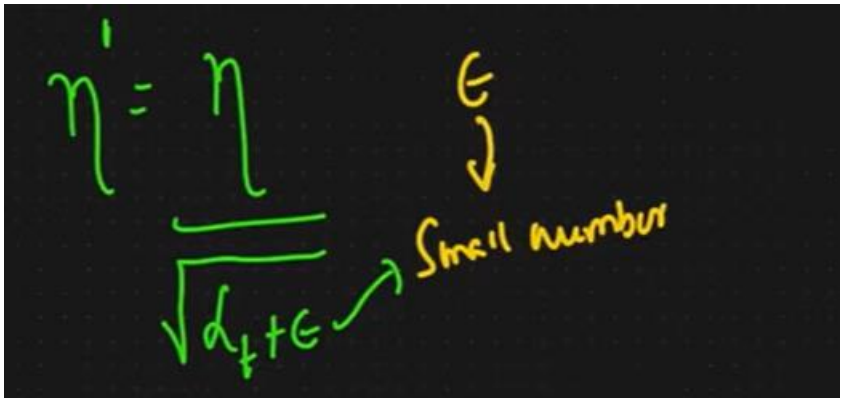


⑤ Adagrad → Adaptive Gradient Descent



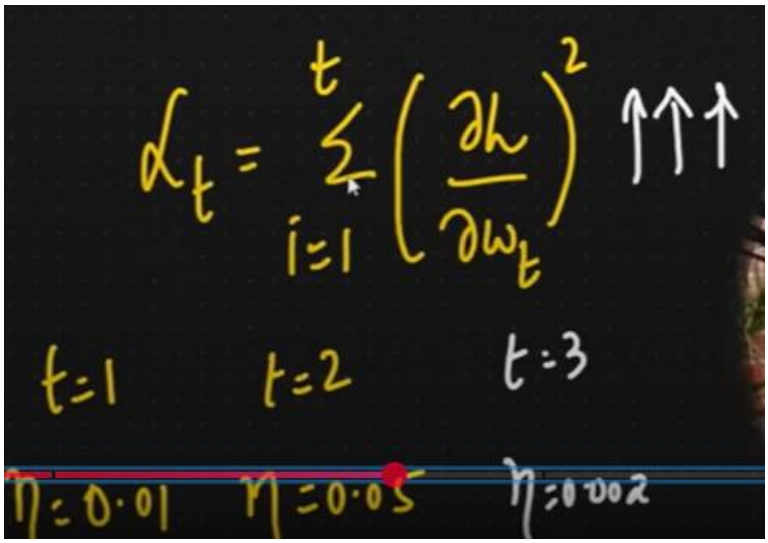


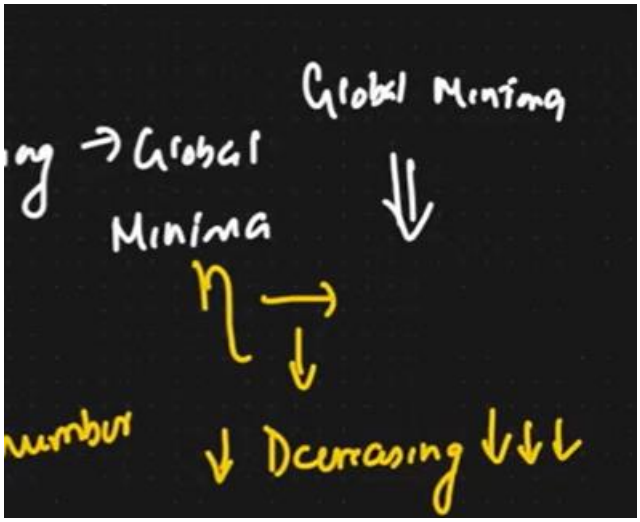
Earlier learning rate is fixed where as in adagrad learning rate is adaptive meaning it may range

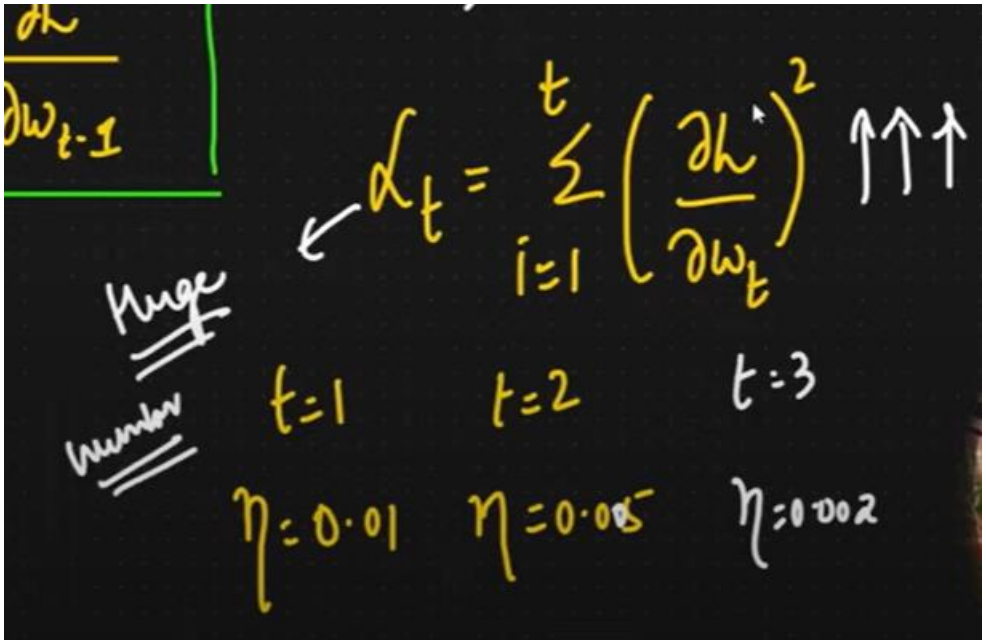


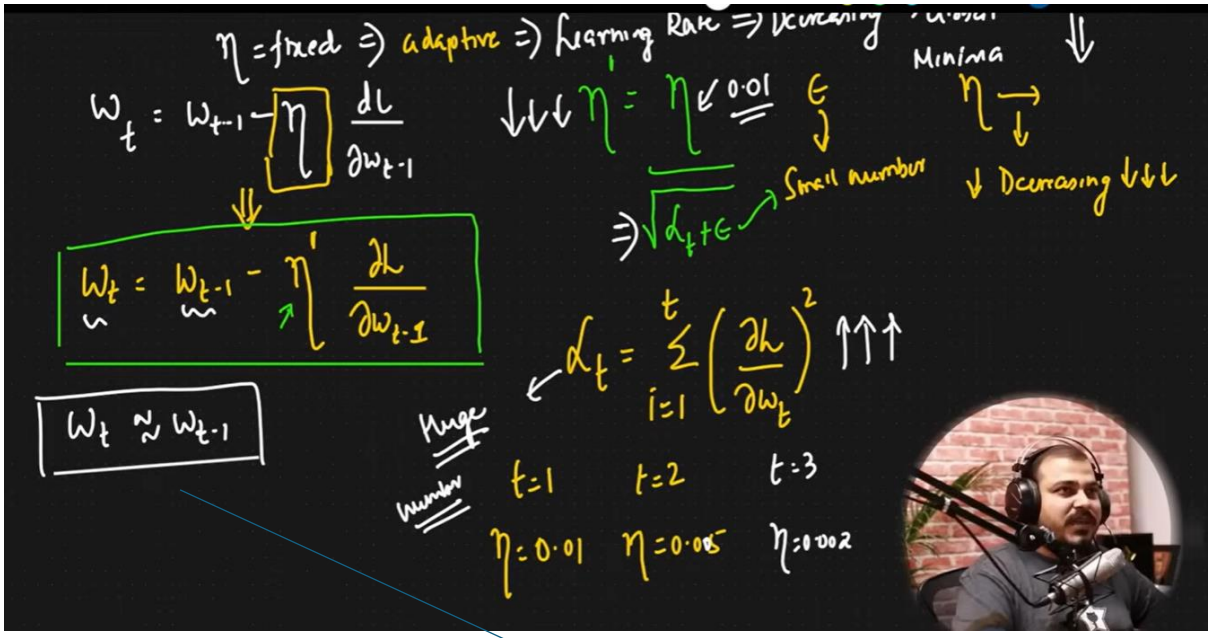
Just to avoid 0 in denominator we are adding small number epsilon

We need to decrease it as we reach global minima



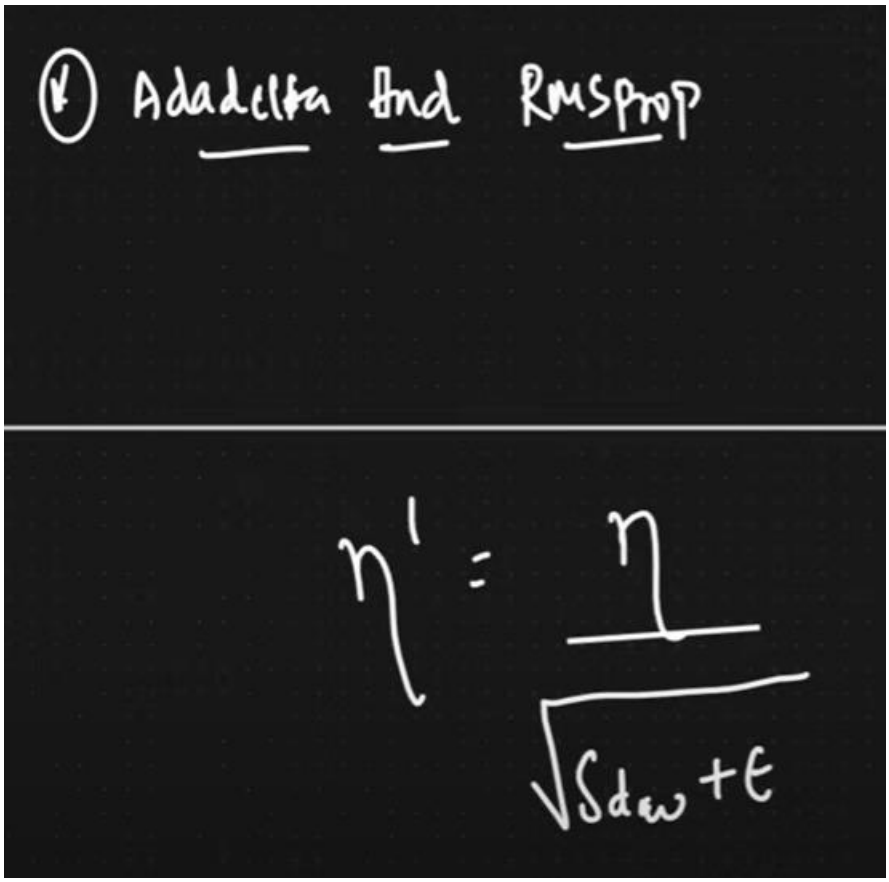

Global Minima
→ Global
Minima
η
number
↓ Decreasing


$\frac{\partial h}{\partial w_{t-1}}$
$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial h}{\partial w_t}\right)^2$
Huge
number
t=1
t=2
t=3
η=0.01
η=0.005
η=0.002



What if we add huge number since we add derivatives learning rate will be changed then in order to overcome we will go for adadelta and rms prop

The learning rate will be smaller and weight will not get updated it will be same as old weight



Exponential weighted avaergage

$$Sdw = 0$$

$$Sdw_t = \beta\, Sdw_{t-1} + (1-\beta)\left(\frac{\partial L}{\partial w_{t-1}}\right)^2$$

$\beta = 0.95$

$$Sdw_t = (0.95)\, Sdw_{t-1} + (0.05)\left(\frac{\partial L}{\partial w_{t-1}}\right)^2$$

With help of this beta parameter we are controlling the increase there will be slow decreasing of learning rate

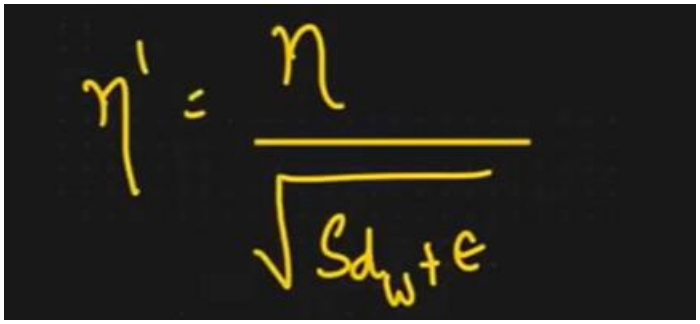
⑧ Adam Optimizer (Best Optimizer)

Momentum + RMSPROP (Adaptive Learning Rate)

$$Vdw = 0 \quad Vdb = 0 \quad Sdw = 0 \quad Sdb = 0$$

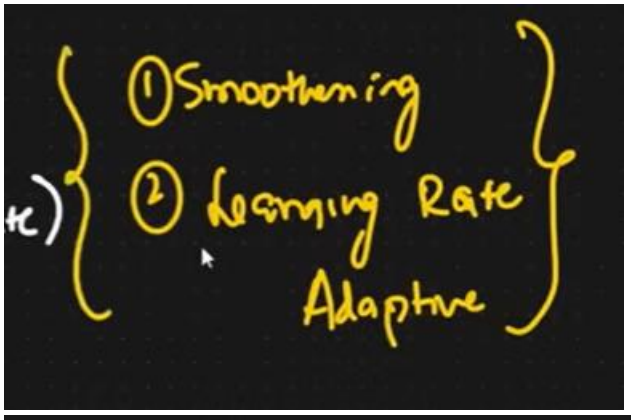
$$W_t = W_{t-1} - \eta' Vdw$$

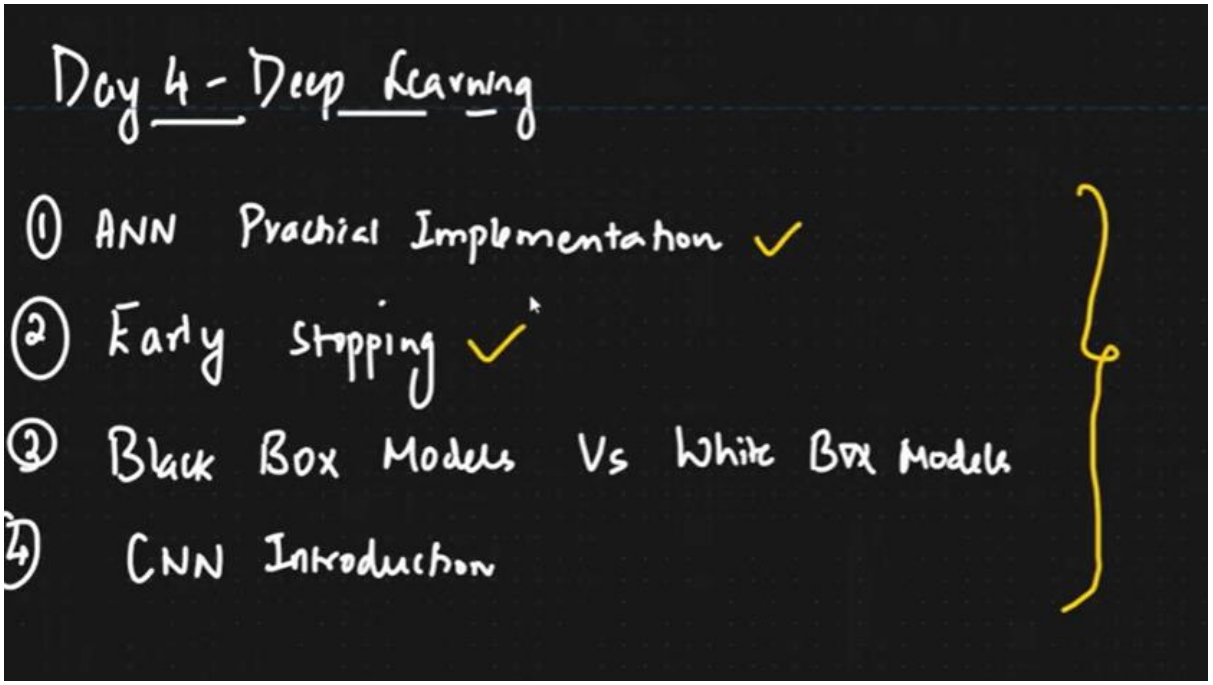
$$b_t = W_{b\,t-1} - \eta' Vdb$$

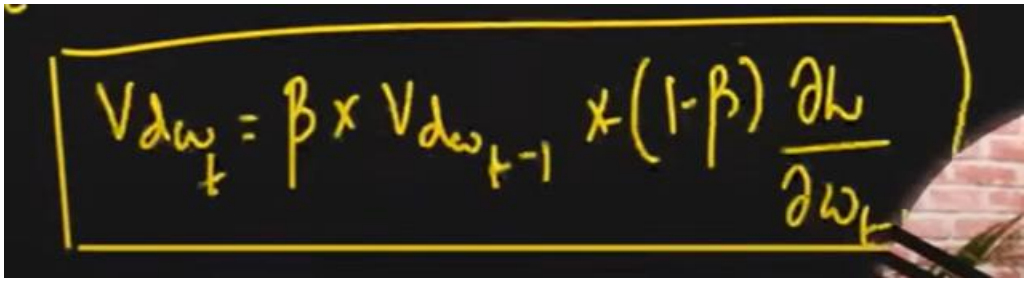

$\eta' = \frac{\eta}{\sqrt{S d_w + \epsilon}}$


$V_{dw_t} = \beta \times V_{dw_{t-1}} + (1-\beta)\frac{\partial L}{\partial w_{t-1}}$

It achieves


etc)
1 Smoothening
2 Learning Rate Adaptive


Day 4 - Deep Learning
1 ANN Practical Implementation ✓
2 Early Stopping ✓
3 Black Box Models Vs White Box Models
4 CNN Introduction




```
!pip install tensorflow-gpu
```

```
import tensorflow as tf
print(tf.__version__)
```

```
2.8.0
```

```
[3] ## import some basics libraries
    import numpy as np
    import matplotlib.pyplot as plt
    import pandas as pd
```

```
dataset=pd.read_csv('Churn_Modelling.csv')
dataset.head()
```

```
## Divide the dataset into independent and dependent features

X=dataset.iloc[:,3:13]
y=dataset.iloc[:,13]
```

```
## Ferature Engineering
pd.get_dummies(X['Geography'])
```

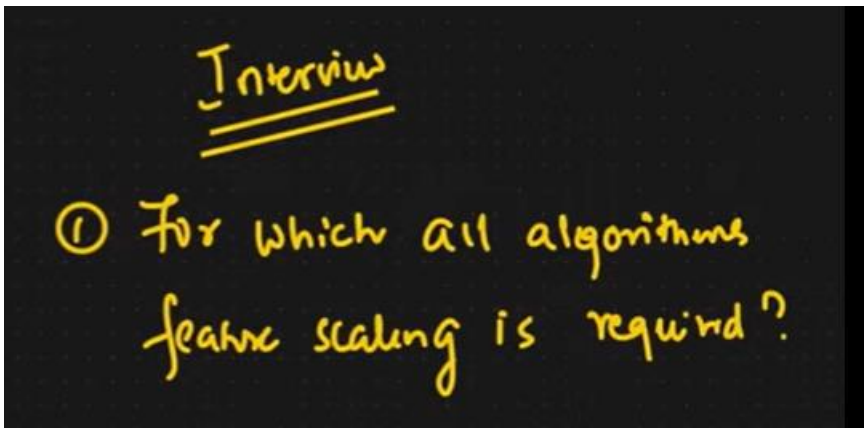
	France	Germany	Spain
0	1	0	0
1	0	0	1
2	1	0	0
3	1	0	0
4	0	0	1

```
## Ferature Engineering
pd.get_dummies(X['Geography'],drop_first=True)
```

```
[10] ## Ferature Engineering
     geography=pd.get_dummies(X['Geography'],drop_first=True)
     gender=pd.get_dummies(X['Gender'],drop_first=True)
```

```
[12] ## concatenate these variables with dataframe
     X=X.drop(['Geography','Gender'],axis=1)
```

```
[15] X=pd.concat([X,geography,gender],axis=1)
```



ANN {Feature Scaling}

{Feature Scaling}

ANN? ✓

LR? ✓

Log R? ✓

Decision? ✗

RF ? ✗

XGB

KNN ✓

KMeans ✓

ANN {Feature Scaling}

① Distance based Scaling

② Gradient Descent ✓

XGB

KNN ✓

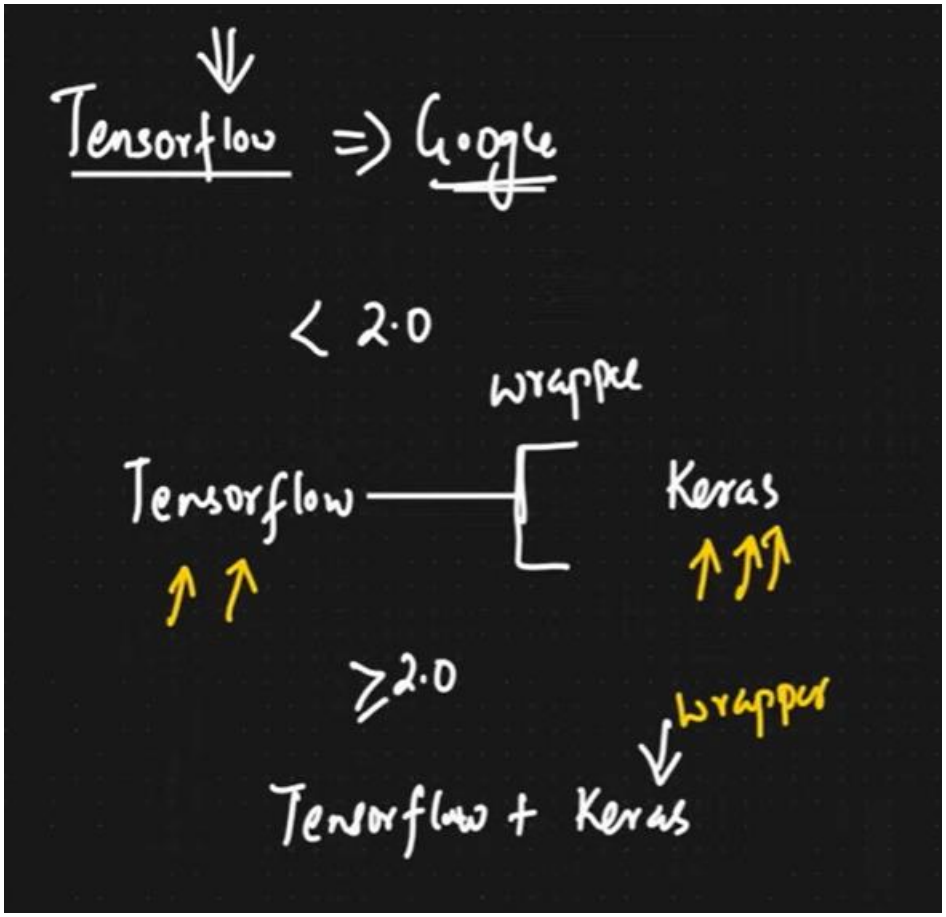
KMeans ✓

```
[16] #Splitting the dataset into Training set and Test Set
     from sklearn.model_selection import train_test_split

     X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=0)
```

```
#feature Scaling

from sklearn.preprocessing import StandardScaler
sc =StandardScaler()
X_train=sc.fit_transform(X_train)
X_test=sc.transform(X_test)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:444: Use
  f"X has feature names, but {self.__class__.__name__} was fitt
```

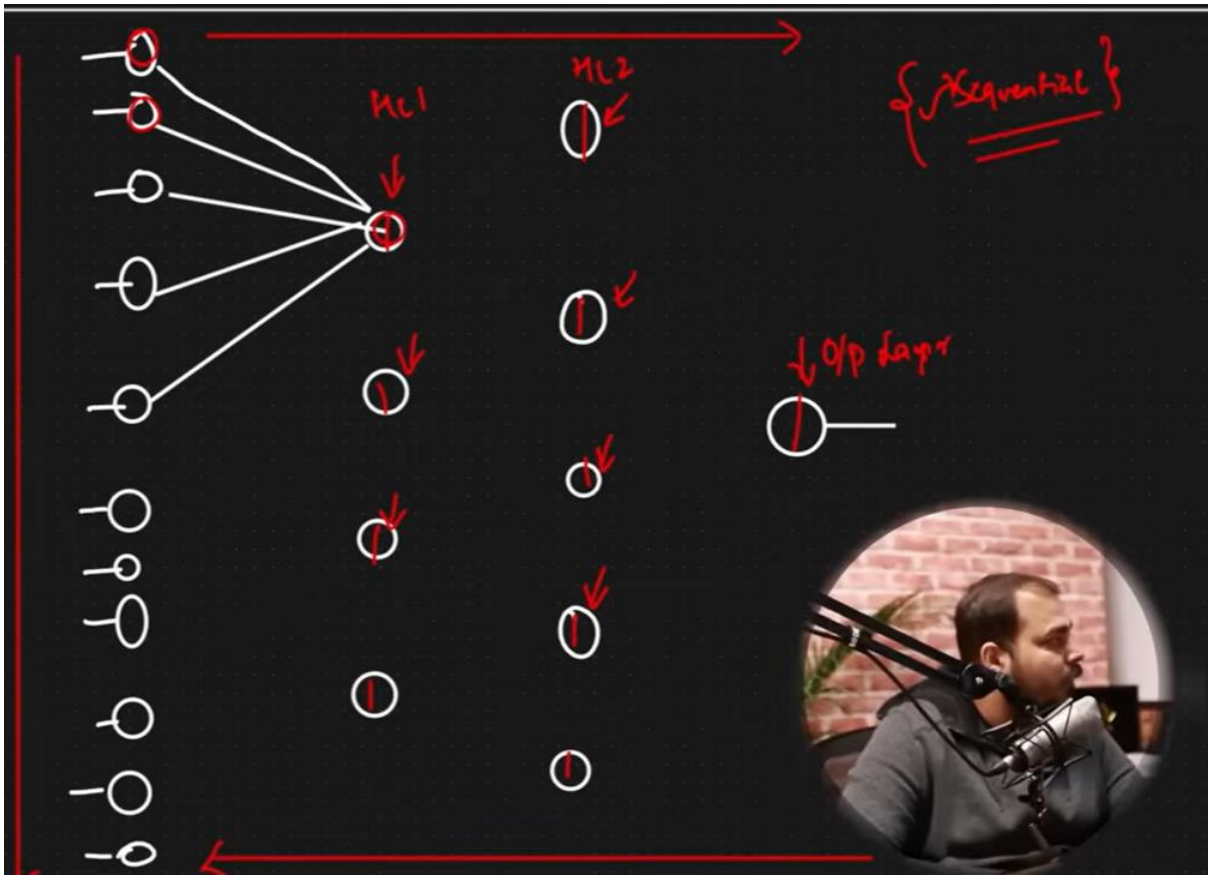
Why we apply fit_transform on training dataset and not an test dataset - just to avoid data leakage



Keras has been integerated with tensorflow 2.0

```
## Part 2 Now lets create the ANN
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LeakyReLU,PReLU,ELU,ReLU
from tensorflow.keras.layers import Dropout
```

With help of dense layer we can create output layer, hidden layer



To avoid overfitting we can introduce drop out layer in ann drop out ratio =0.30 means 30 % of neurons will be deactivated

Remaining will be randomly selected

```
[23] ## Part 2 Now lets create the ANN
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense
     from tensorflow.keras.layers import LeakyReLU,PReLU,ELU,ReLU
     from tensorflow.keras.layers import Dropout
```

```
[24] ### Lets initialize the ANN
     classifier=Sequential()
```

```
## Adding the input Layer
classifier.add(Dense(units=11,activation='relu'))
```

```
[26] # adding the first hidden layer
     classifier.add(Dense(units=7,activation='relu'))
```

```
##adding the second hidden layer
classifier.add(Dense(units=6,activation='relu'))
```

```
[28] ##  Adding the output layer
     classifier.add(Dense(1,activation='sigmoid'))
```

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

Adam uses default learning rate is 0.01

```
import tensorflow
opt=tensorflow.keras.optimizers.Adam(learning_rate=0.01)
```

```
<keras.optimizer_v2.adam.Adam at 0x7ff2d0187e50>
```

```
32] classifier.compile(optimizer=opt,loss='binary_crossentropy',metrics=['accuracy'])
```

```
model_history=classifier.fit(X_train,y_train,validation_split=0.33,batch_size=10,epochs=1000)
```

```
Epoch 1/1000
535/536 [============================>.] - ETA: 0s - loss: 0.4262 - accuracy: 0.8114
```

What will be my number of epochs for that we will use early stopping

Stop training when a monitored metric has stopped improving.

Assuming the goal of a training is to minimize the loss. With this, the metric to be monitored would be `'loss'`, and mode would be `'min'`. A `model.fit()` training loop will check at end of every epoch whether the loss is no longer decreasing, considering the `min_delta` and `patience` if applicable. Once it's found no longer decreasing, `model.stop_training` is marked True and the training terminates.

The quantity to be monitored needs to be available in `logs` dict. To make it so, pass the loss or metrics at `model.compile()`.

```
## Early Stopping
import tensorflow as tf
early_stopping=tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    min_delta=0.0001,
    patience=20,
    verbose=1,
    mode="auto",
    baseline=None,
    restore_best_weights=False,
)
```

```
model_history=classifier.fit(X_train,y_train,validation_split=0.33,batch_size=10,epochs=1000,callbacks=early

Epoch 1/1000
347/536 [==================>.....] - ETA: 0s - loss: 0.4509 - accuracy: 0.7939
```

```
36] model_history.history.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
37] # summarize history for accuracy
    plt.plot(model_history.history['accuracy'])
    plt.plot(model_history.history['val_accuracy'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()
```

```
# summarize history for accuracy
plt.plot(model_history.history['accuracy'])
plt.plot(model_history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
[39] # Part 3 - Making the predictions and evaluating the model

    # Predicting the Test set results
    y_pred = classifier.predict(X_test)
    y_pred = (y_pred >= 0.5)
```

```
## make the confusion matrix
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
cm
```

```
array([[1527,   68],
       [ 215,  190]])
```

```
[42] ## Calculate the accuracy
     from sklearn.metrics import accuracy_score
     score=accuracy_score(y_pred,y_test)
```

```
score
```

```
0.8585
```

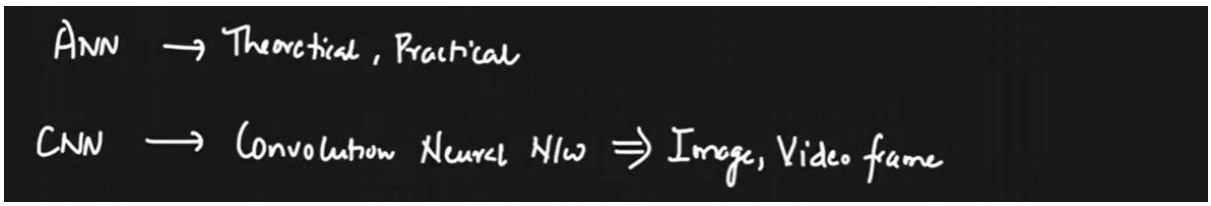
```
##get the weights
classifier.get_weights()
```

```
          -1.072681  , -0.9926772 , -1.2926713 , -0.16671637,  0.598037  ,
```

```
# adding the first hidden layer
classifier.add(Dense(units=7,activation='relu'))
classifier.add(Dropout(0.2))
```

```
##adding the second hidden layer
classifier.add(Dense(units=6,activation='relu'))
classifier.add(Dropout(0.3))
```

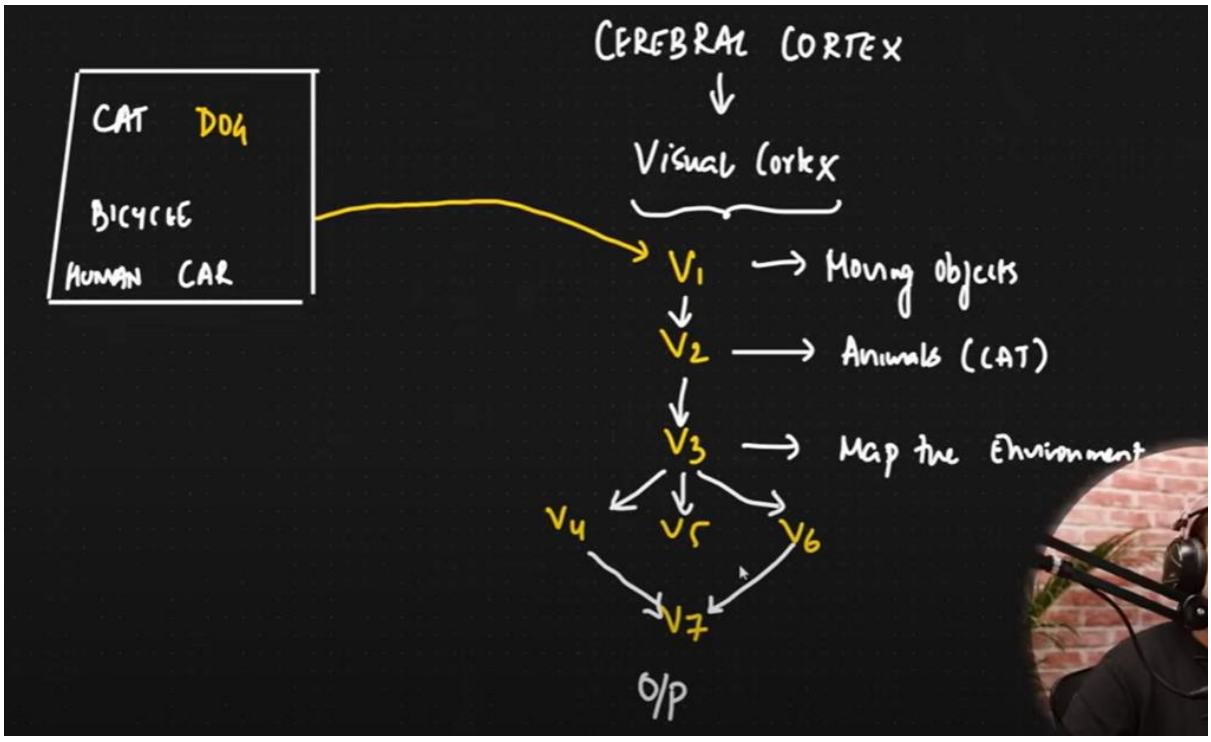
CNN

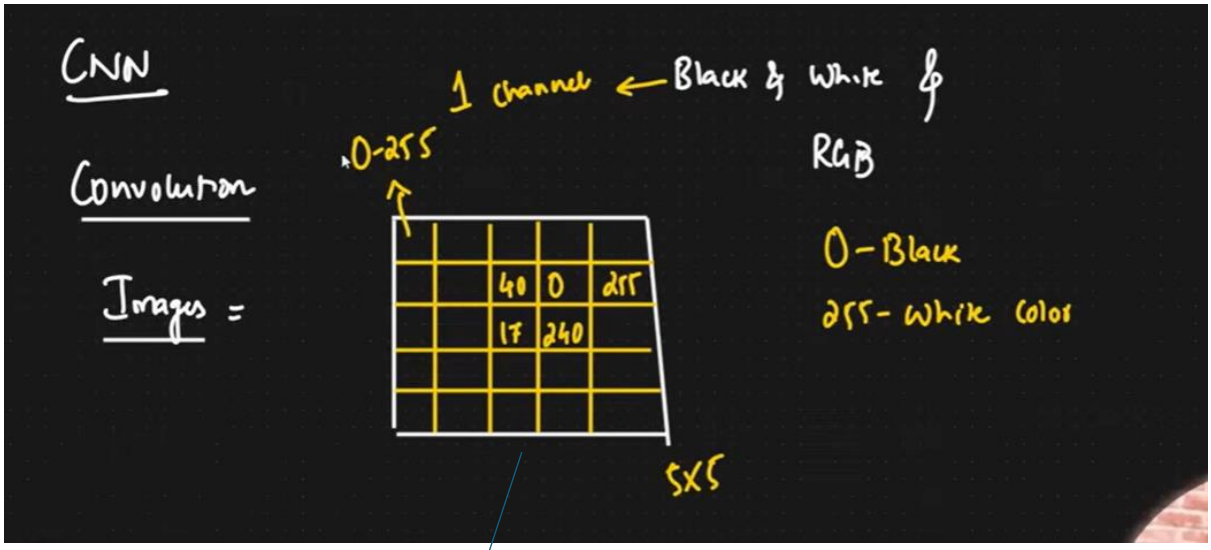


Agenda

1. CNN Vs Human Brain
2. Convolution opeation
   - Convolution
   - Padding
   - Stride
   - Filtus (Kernels)

3. Max Pooling
4. Flattening
5. Practical Implementa.

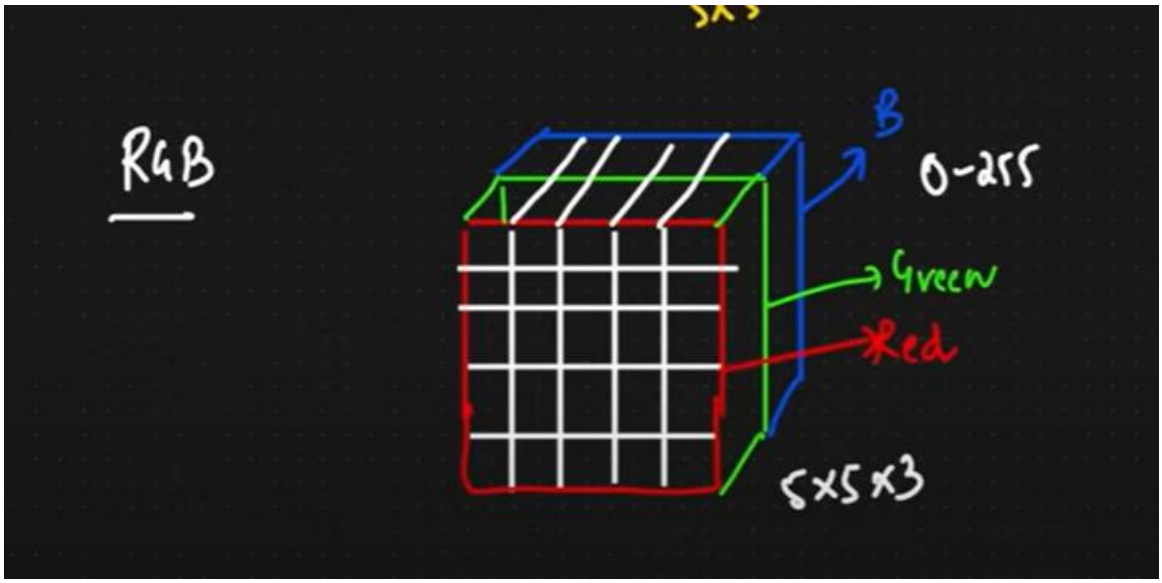


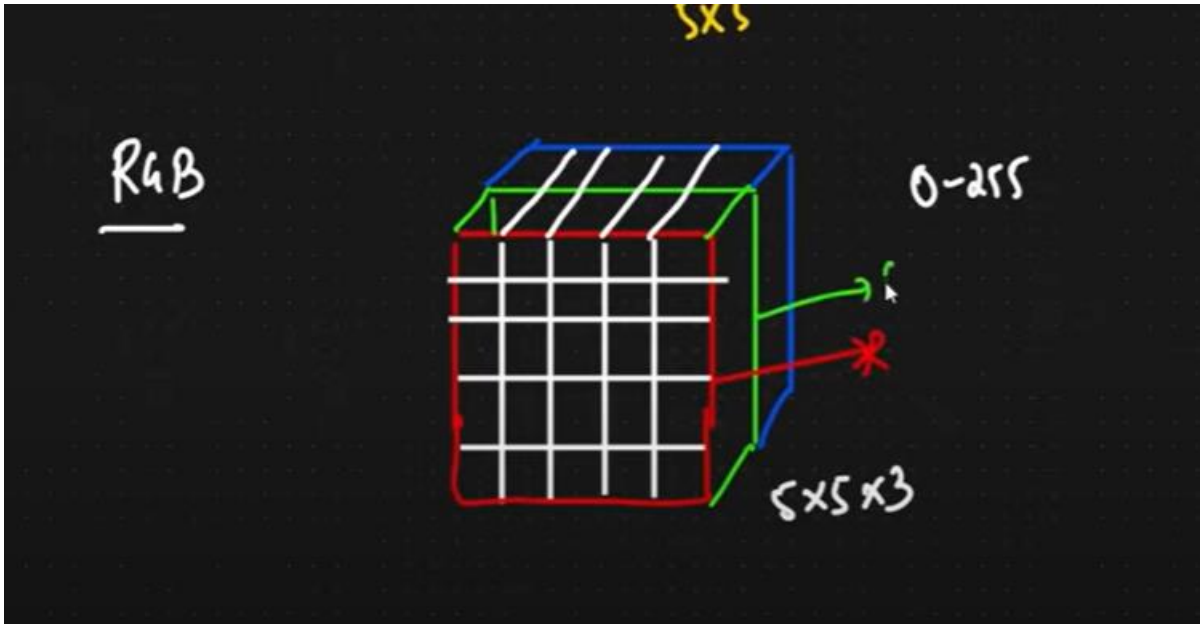


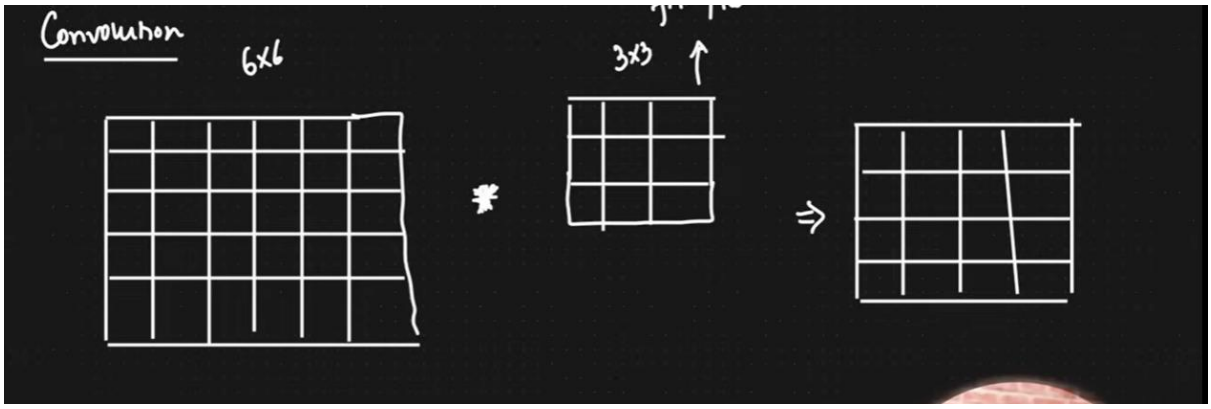
Each image it has a pixel range from 0 to 255

5 * 5 pixel

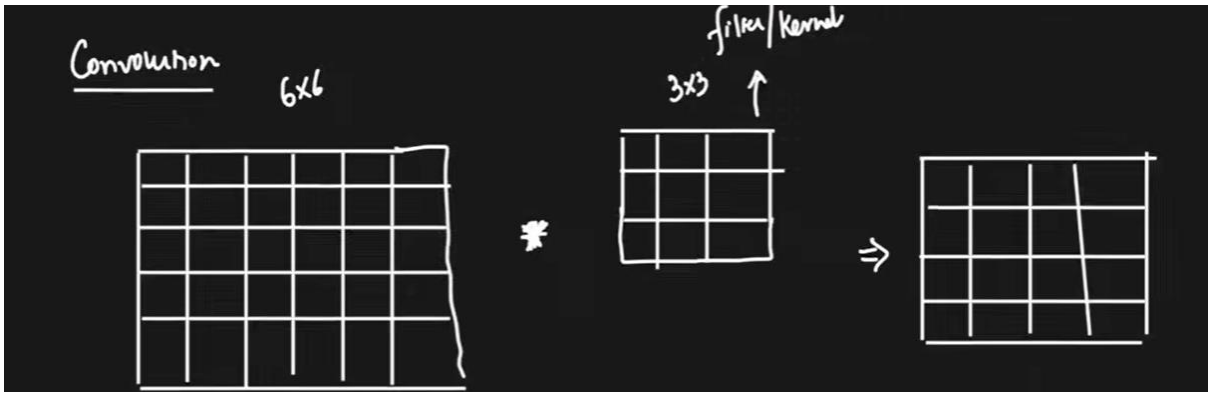
RGB every color image has been setup in RGB format (red blue and green)

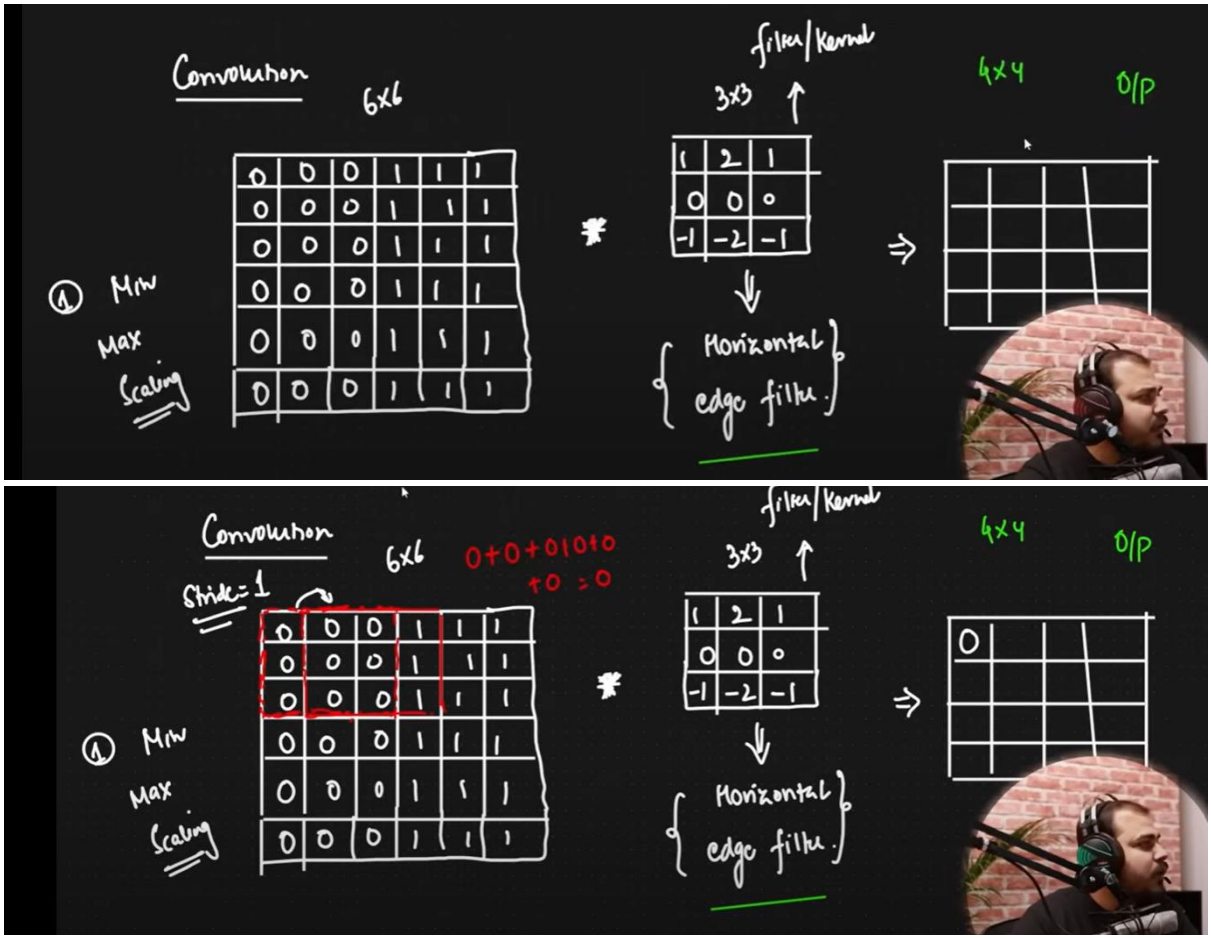




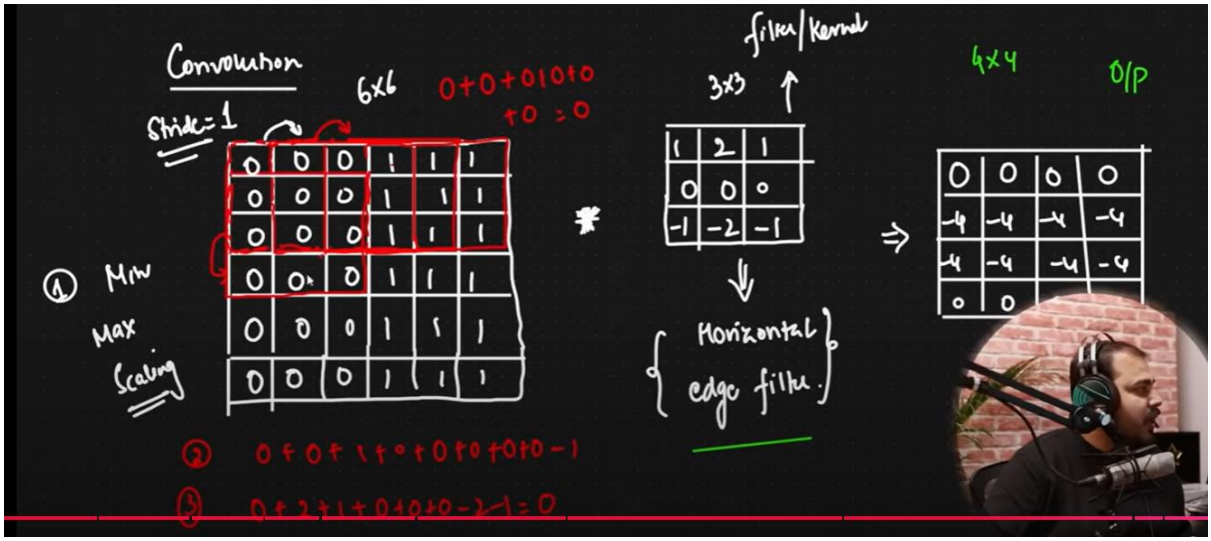


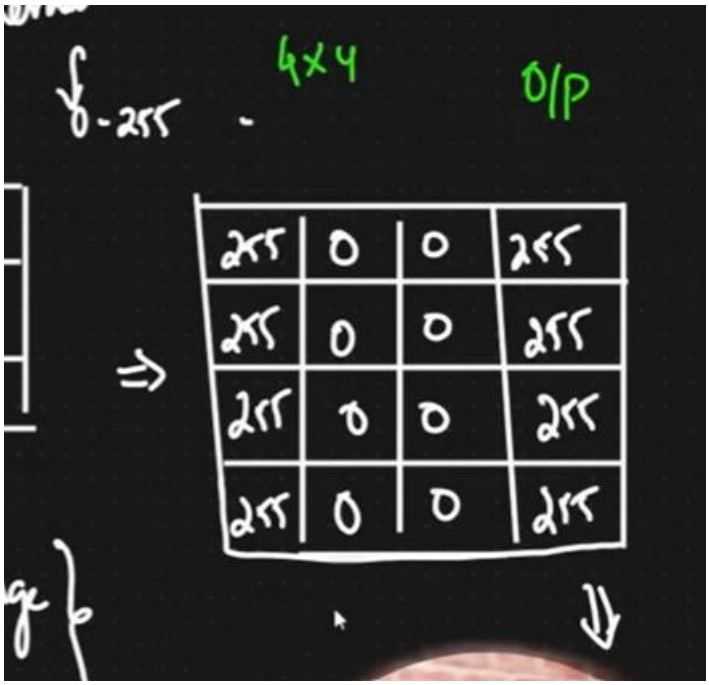
In covolution we bring the values 0 to 1 by min -max scalar





When we move the filter on top of matrix then we called it as stride say when move towards left side with one step then stride will equal to 1 , if move towards 2 steps then stride will be 2

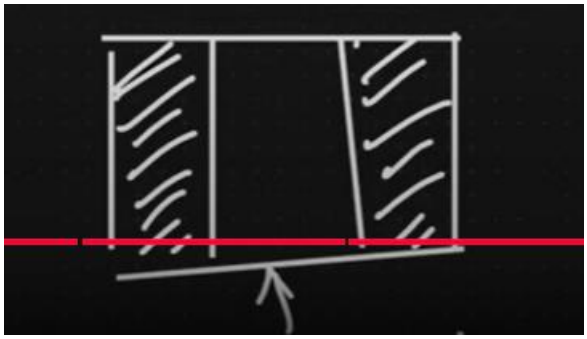




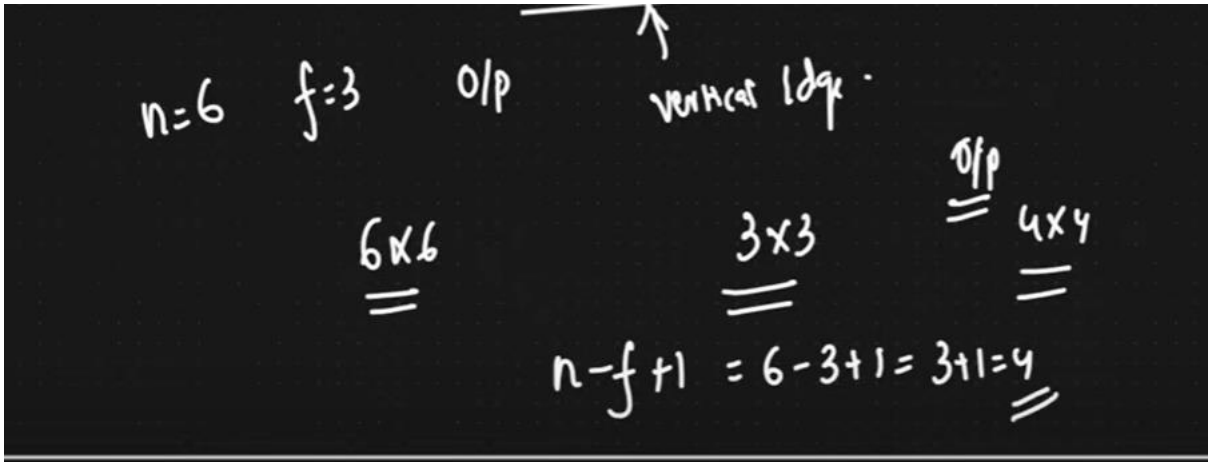
If we inverse the values means -4 becames 0 and 0 will become 255 since 0 being the highest and -4 being the lowest

255 is white color 0 is black color

If we apply vertical filter then we should vertical edges this process is called as convolution operation we are trying to extract information from image



Calculation for getting output matrix



If image size decreases we lose some kind of information

Padding

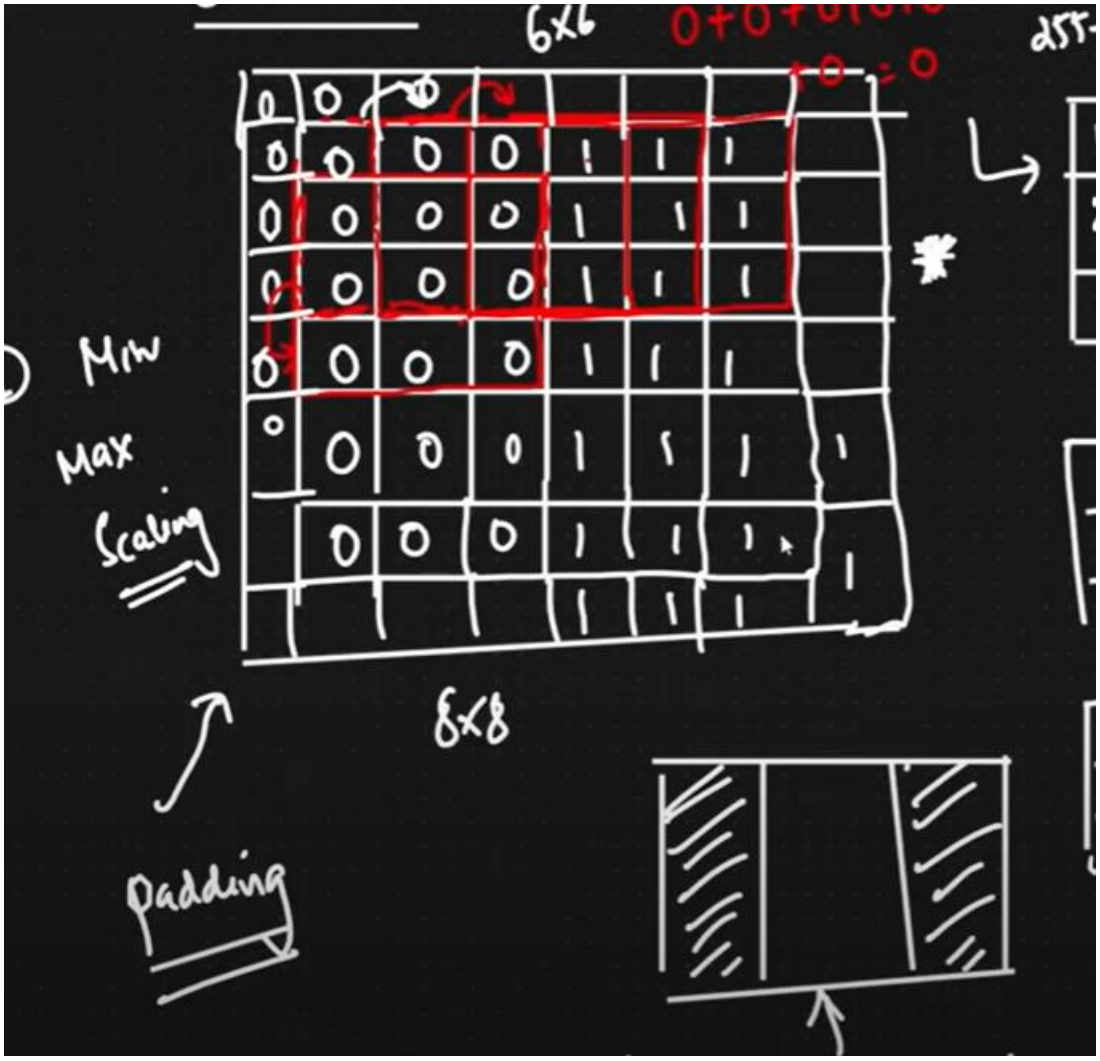
Concept of adding a layer is called as padding

Protecting image (size of a image) by adding a layer

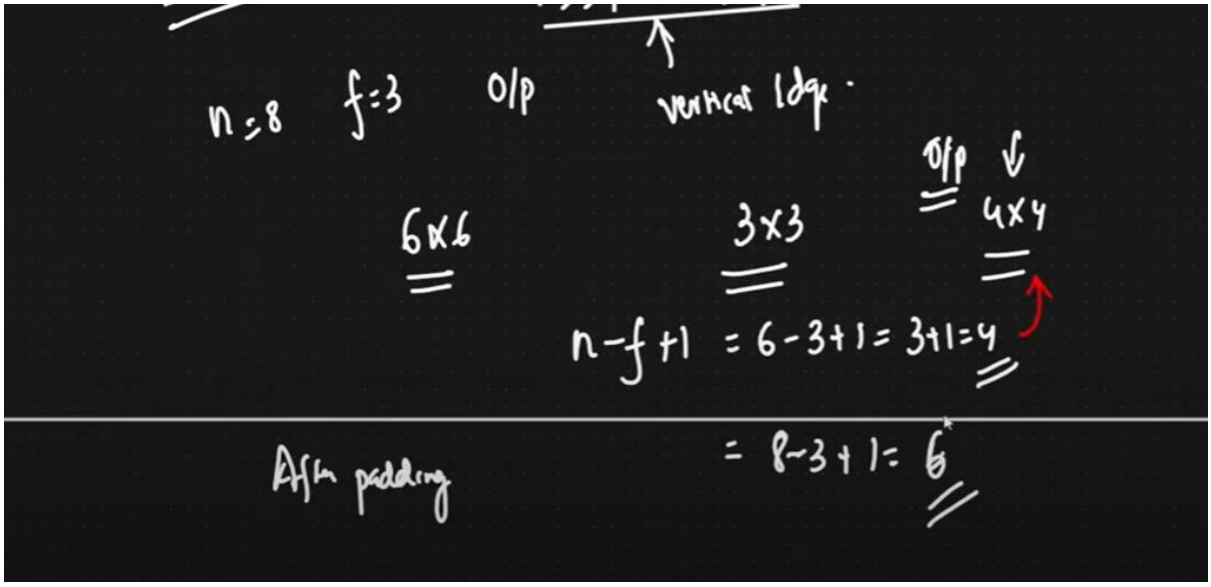
What values we can fill there is called zero-padding we add simply 0

Whatever nearest value we can that one also

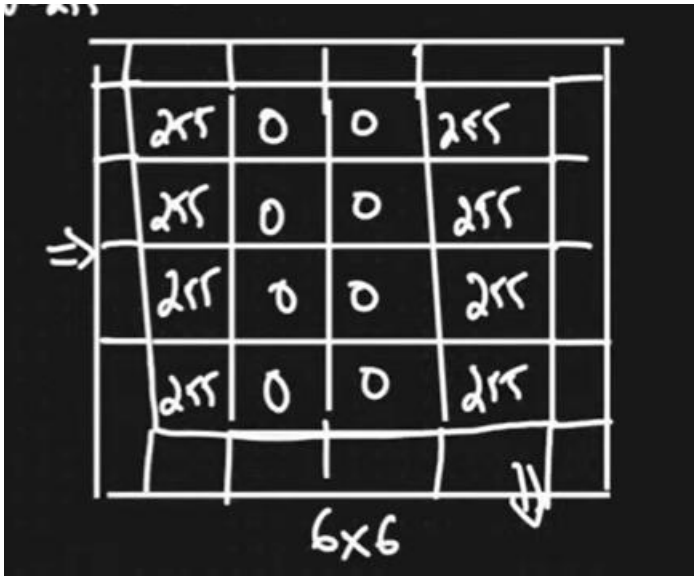
To prevent information loss of a image we are adding a layer called padding



After applying padding



Then output will become like



After padding

$= 8-3+1 = 6$

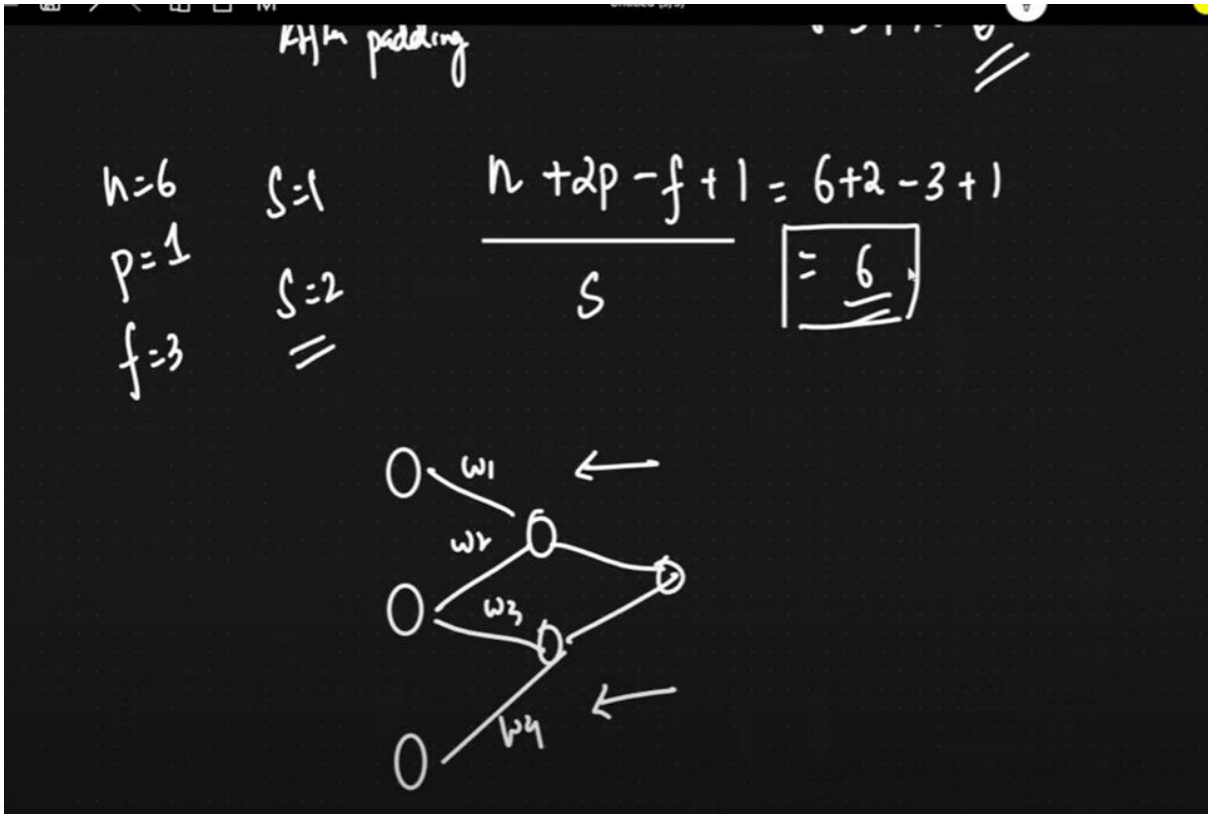
$n = 6$

$p = 1$

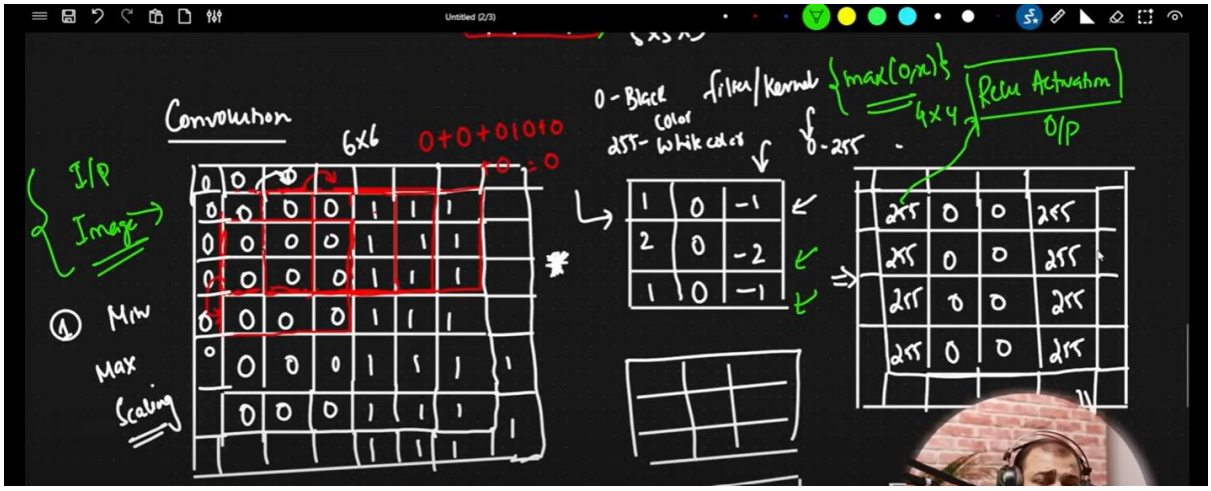
$f = 3$

$n + 2p - f + 1 = 6 + 2 - 3 + 1$

$= 6$



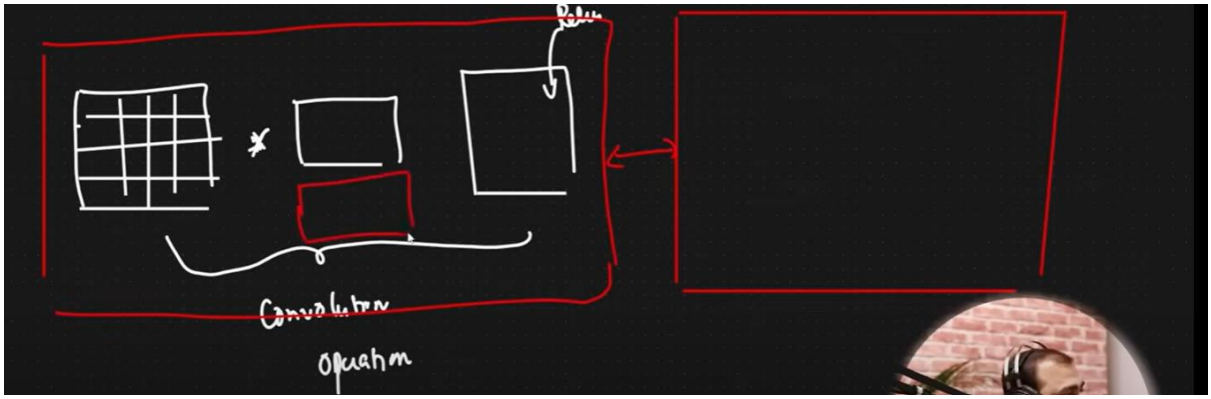
S is stride



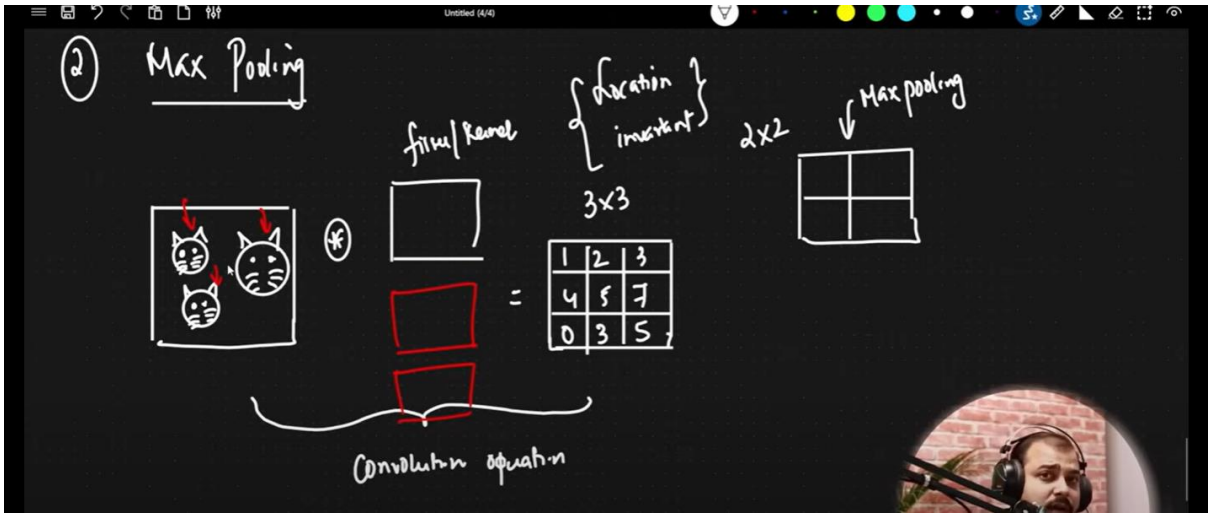
We apply activation function each and every cell to find out derivative to update the filters

By means of back propagation

Processing the image filtering it and getting output and applying output on top of relu activation function is called as convolution operation

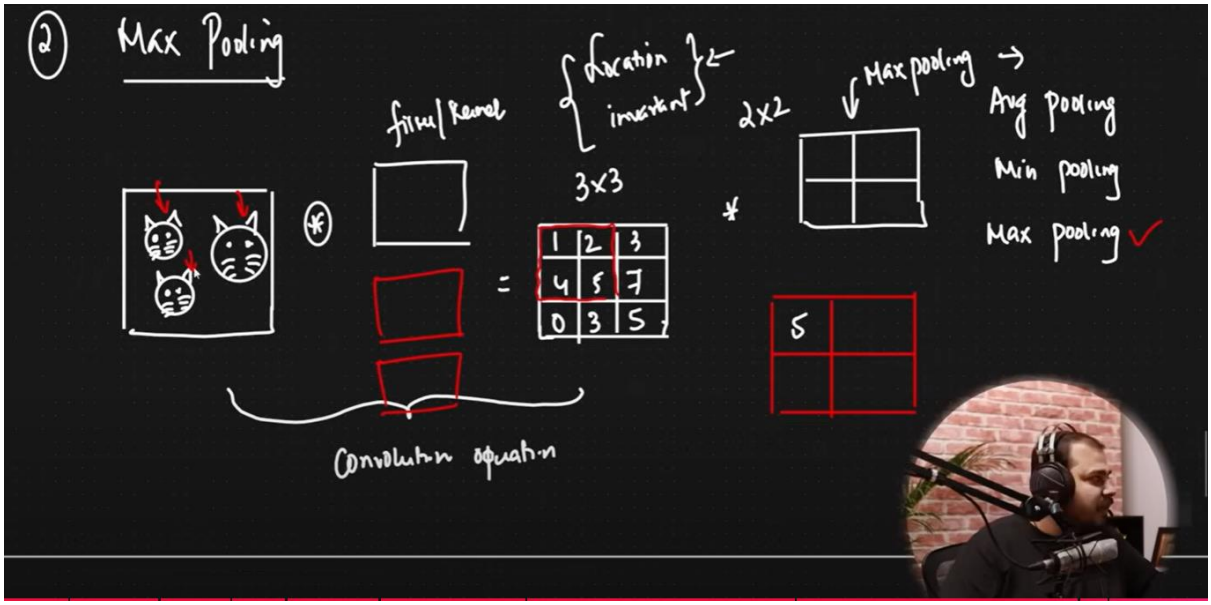


Max pooling

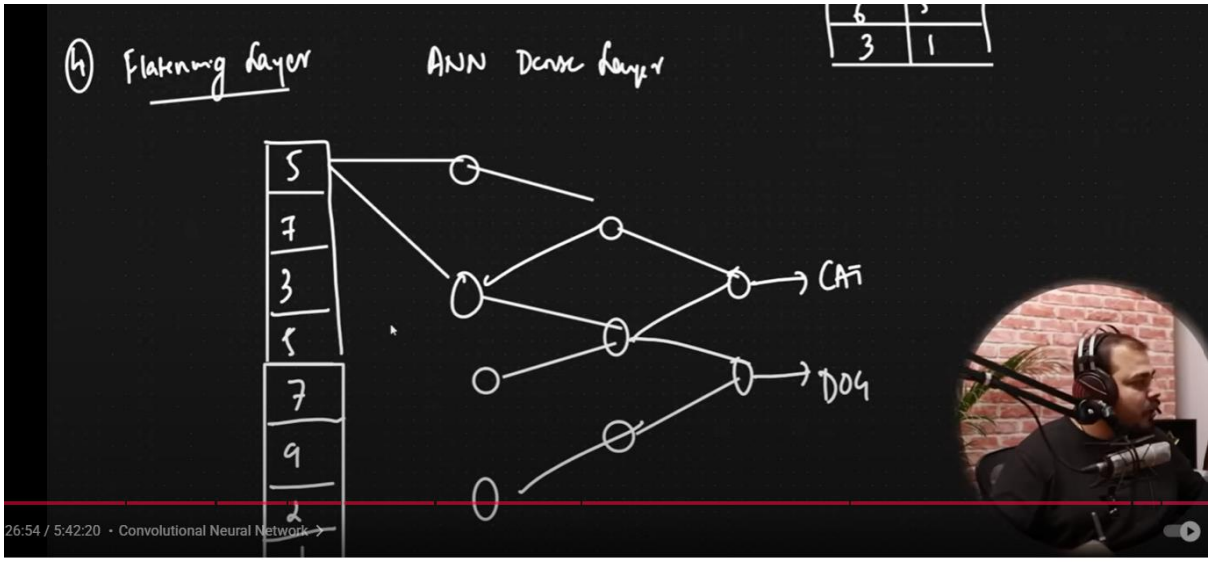


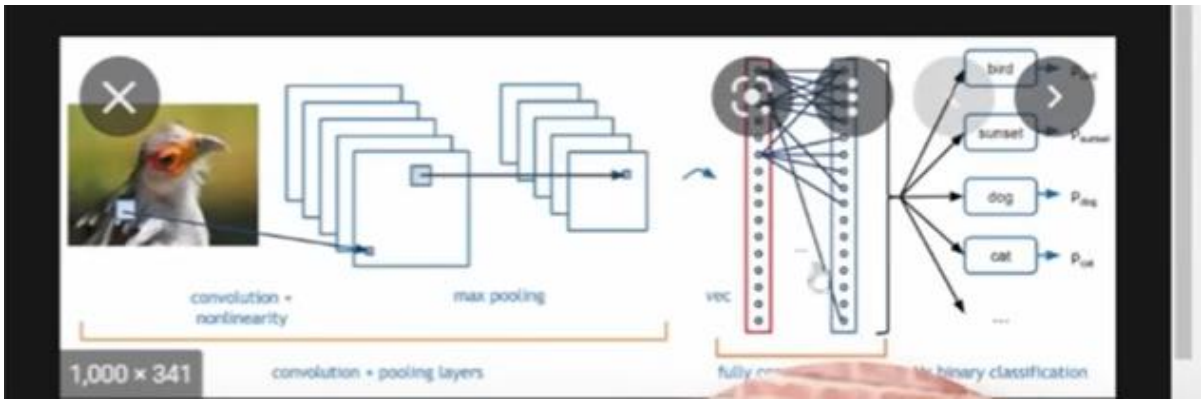
Location invariant – says as we pass on or move on with more convolution layers we extract more images capture max information for that we go for max pooling

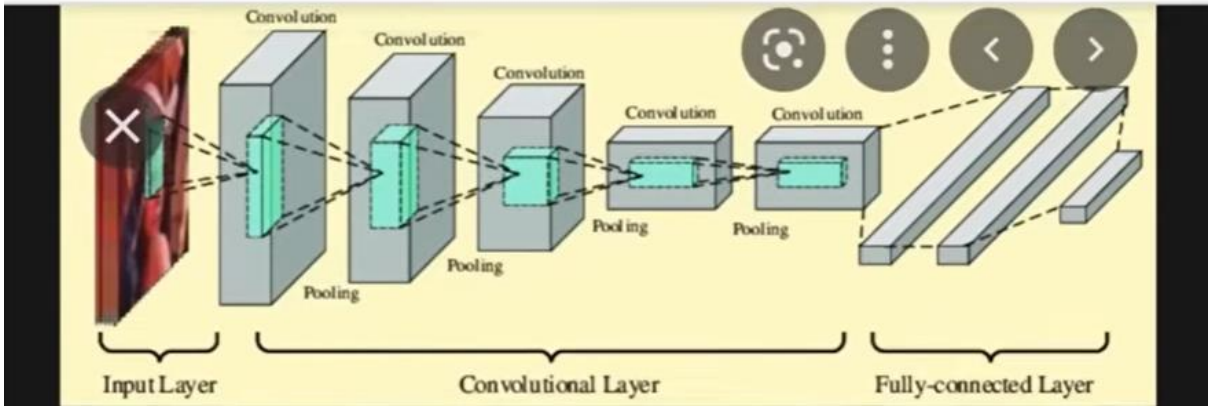
We place 2*2 max pooling matrix on top of 3*3 filters then we can get another matrix in red

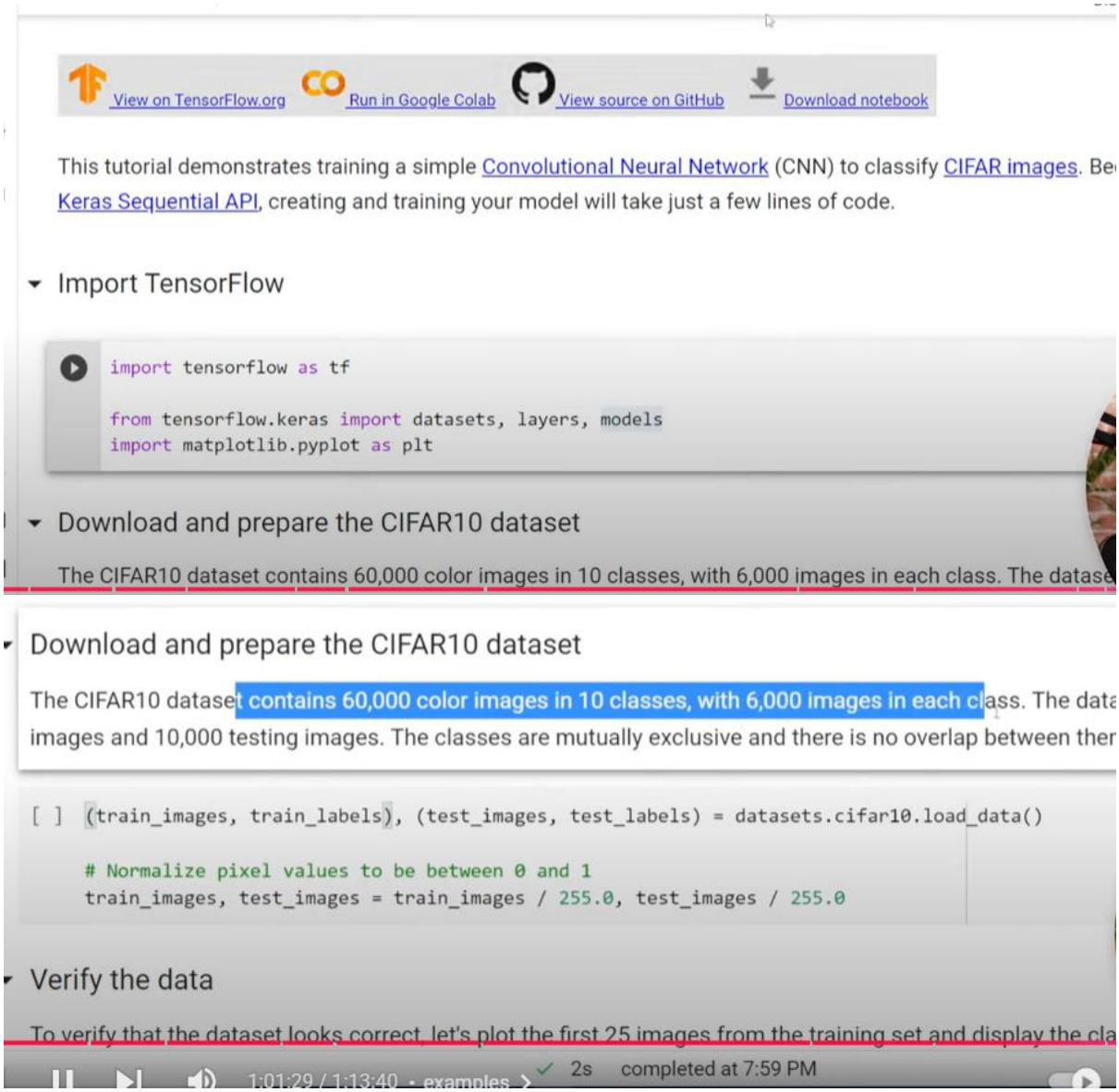


Output of max pooling will flatten basically like a input in ann (wil became a dense layer)


4 Flattening layer
ANN Dense layer
3 1
5
7
3
1
7
9
2
CAT
DOG
26:54 / 5:42:20 • Convolutional Neural Network >


convolution + nonlinearity
max pooling
vec
bird
sunset
dog
cat
1,000 × 341
convolution + pooling layers
binary classification


Convolution
Convolution
Convolution
Convolution
Convolution
Pooling
Pooling
Pooling
Pooling
Input Layer
Convolutional Layer
Fully-connected Layer



First parameter is filter 32 size is 3*3

The 6 lines of code below define the convolutional base using a common pattern: a stack of Conv2D and M

As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch s dimensions, color_channels refers to (R,G,B). In this example, you will configure your CNN to process input format of CIFAR images. You can do this by passing the argument `input_shape` to your first layer.

```
[ ]  model = models.Sequential()
     model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Conv2D(64, (3, 3), activation='relu'))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

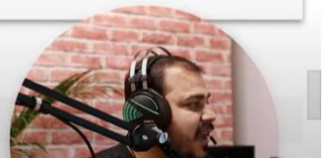
Add Dense layers on top

To complete the model, you will feed the last output tensor from the convolutional base (of shape (4, 4, 64)) into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. CIFAR has 10 output classes, so you use a final Dense layer with 10 outputs.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

Here's the complete architecture of your model:

```
[ ]  model.summary()
```



so this is my output layer and this is

Compile and train the model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```