जय माता दी
जय गणेश जी
श्री शिवाय नमस्तुभ्यम

The Main aim of Deep learning is to mimic human Brain.

Why Deep Learning is Becoming Popular?

① 2005 → FB, Insta, Whatsapp, Linkedin

Data → Exponentially ↑↑↑

2008 → {Big Data} → Efficiently

2013 → Company had huge amount of Data

⇓ {AI → popular}

Seamless Experience for Products

⇓

Company will generate Revenue, Better Decision

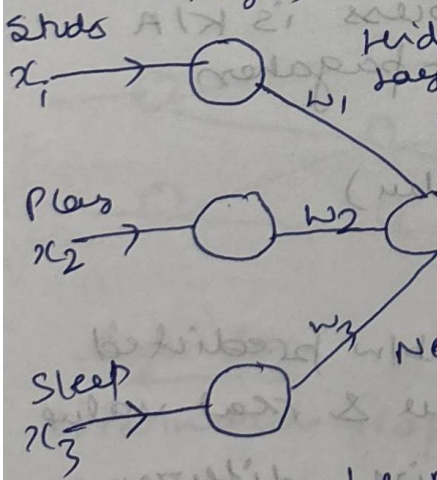
② Hardware advancement (Nvidia)

↳ GPUs → Training the Model

GPUs cost is decreasing

* Perceptron:

{Single Layered Neural Network}

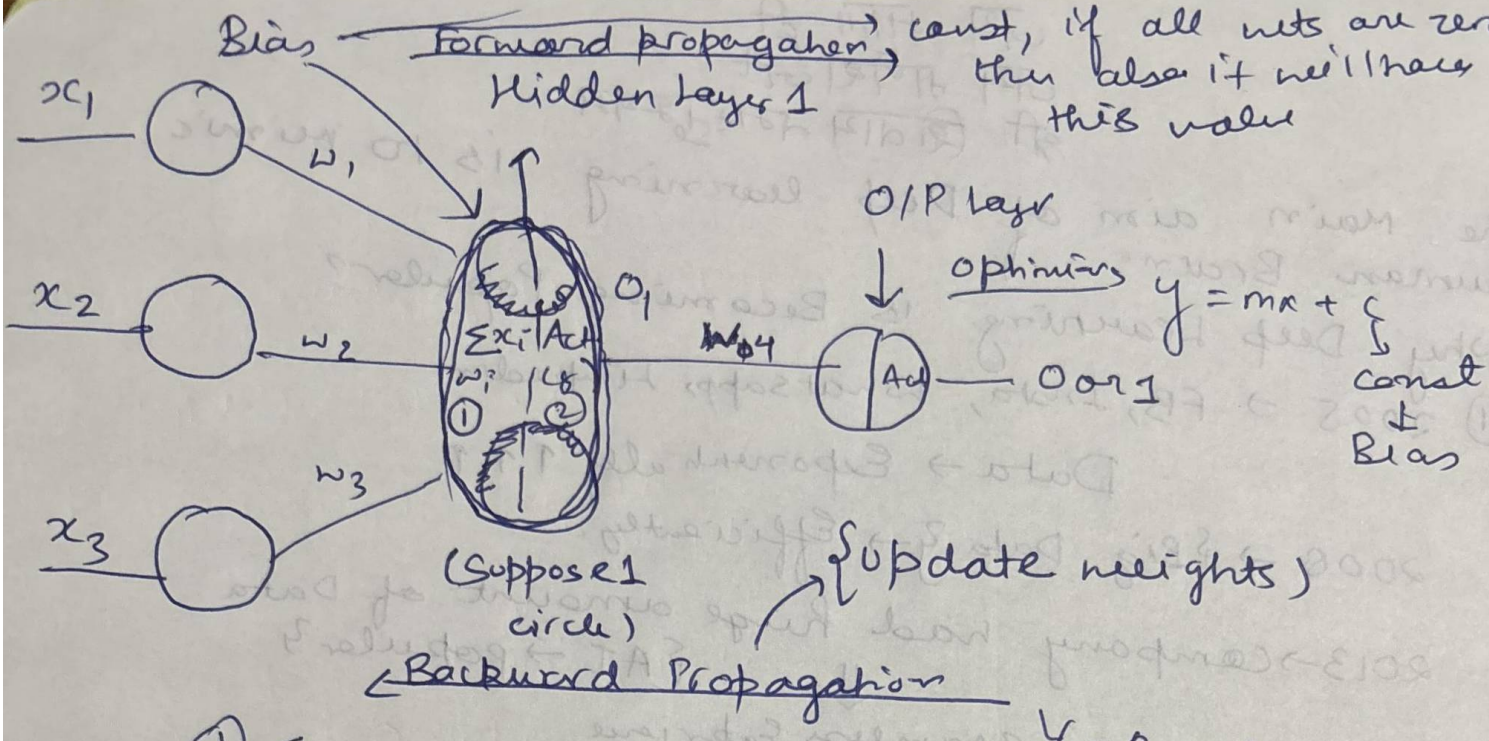


Dataset

Study	Play	Sleep	Pass/Fail
7	3	7	1
2	5	0	0
4	3	7	1

* There can be more than 1 hidden layers

w_1, w_2, w_3 → weights basically tell how much a neuron should get activated



① $\Sigma x_i w_i = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$

$\Downarrow$

$w^T x$

$y = \beta_0 + \beta_1 x$

$y = \beta^T x$

$b \rightarrow$ bias

② Activation func^n

eg - Sigmoid (0 to 1) $= \frac{1}{1+e^{-y}} = \frac{1}{1+e^{-(\Sigma w_i x_i + b)}}$

$\geq 0.5 \rightarrow 1$

$< 0.5 \rightarrow 0$

* We multiplied the inputs with the weights, added a Bias & activated → This entire process is K/A Forward Propagation

Suppose O/P $= \hat{y} = 0$ & $y = 1$ (Truth value)

$y - \hat{y} = 1$

Loss func^n = diff b/w predicted value & real value

The main aim should be to minimize difference and make it near to zero

→ The main aim of Backward propagation is to update weights & hence reducing loss fun^n

→ Optimizers - main aim is to make sure that Each & Every weight will be updated while we are Back - Propagating

Inputs → weights → $\Sigma x_i w_i$ → $\Sigma x_i w_i + B$ → Activation fencn → O/P → Checking Loss fencn → Backward Propagation ← Optimizers ← O/P

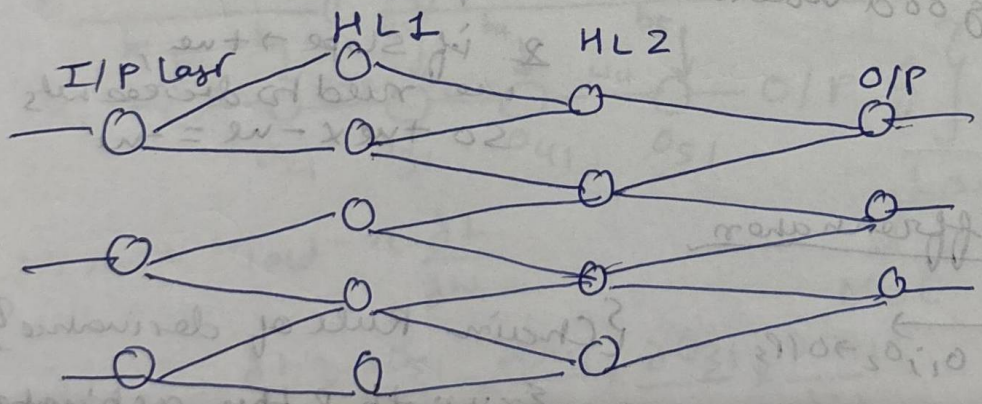
Bias

→ Gradient descent is an Eg. of optimizer

① I/P Layers
② Weights
③ Bias
④ Activation fencn
} Forward propagation

⑤ Loss fencn $\{\hat{y} - y\}$
⑥ Optimizers
⑦ Update the weight
} Backward propagation

Multilayer Neural Network



Agendas -

1) Forward Propagation
2) Chain rule of Derivative
3) Vanishing Gradient Problem
4) Loss fencn

* bias gets added at each hidden layer

→ Backpropagation

1) Weight updation formula
2) Chain rule of Differentiation

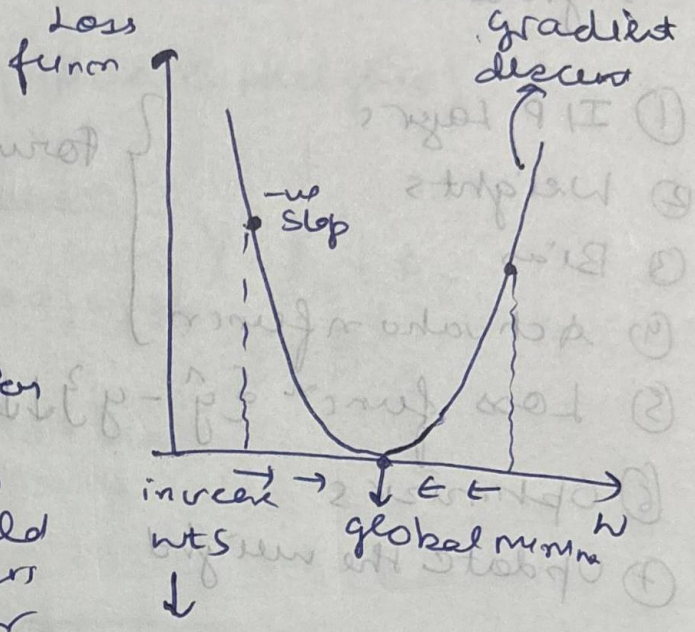
• Weight updation formula

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

$\frac{\partial L}{\partial W_{old}}$ → derivative of Loss

η → learning rate

$\frac{\partial L}{\partial W_{old}}$ → Slope



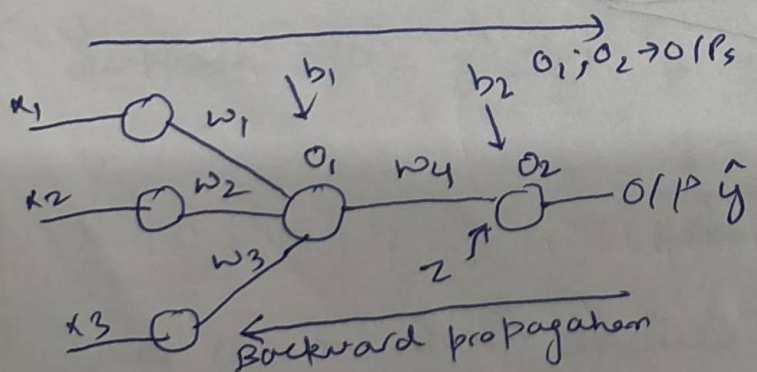
* Learning rate would be better if it is a small number as it will slowly converge to global minima, if it would be larger it would take larger steps & hence might never reach global minima.

$\eta = 0.001$ is a good value

So -ve x -ve will be +ve

& if slope → +ve, we need to decrease wts so +ve x -ve = -ve

• Chain rule of Differentiation



{Chain Rule of derivative}

$O_1 \rightarrow \Sigma x_i w_i + b$ & then activation fen^n applied then O/P = O_1

$$\frac{\partial L}{\partial W_{4old}} = \frac{\partial L}{\partial O_2} \times \frac{\partial O_2}{\partial W_4}$$

Z = activation fen^n

$Z = \sigma(O_1 W_4 + b)$

For W_4,

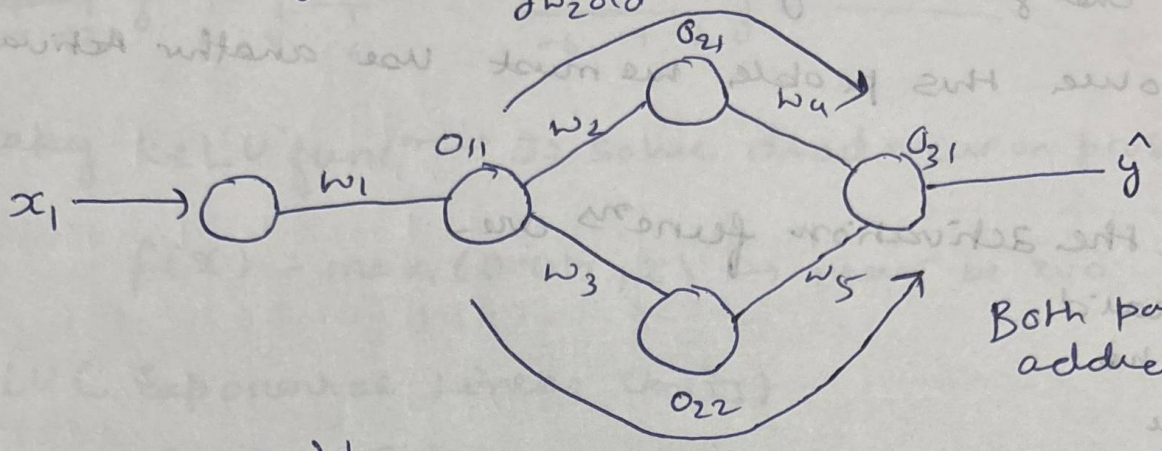
$$W_{4new} = W_{4old} - \eta \frac{\partial L}{\partial W_{4old}}$$

$$b_{2new} = b_{2old} - \eta \cdot \frac{\partial L}{\partial b_{2old}}$$

$w_{1new} = w_{1old} - \eta \frac{\partial L}{\partial w_{1old}} \Rightarrow \frac{\partial L}{\partial w_{1old}} = \frac{\partial L}{\partial o_2} \times \frac{\partial o_2}{\partial o_1} \times \frac{\partial o_1}{\partial w_{1old}}$ } This is chain rule of diff

$w_{2new} = w_{2old} - \eta \frac{\partial L}{\partial w_{1old}} \Rightarrow \frac{\partial L}{\partial w_{2old}}$ → similarly

eg -

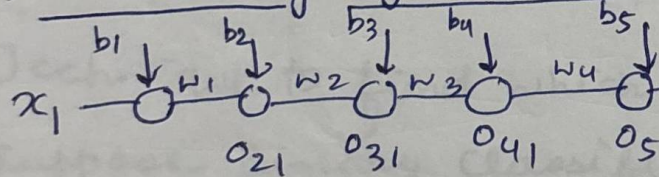


Both paths added

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

$$\frac{\partial L}{\partial w_{1old}} = \left[\frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial o_{21}} \times \frac{\partial o_{21}}{\partial o_{11}} \times \frac{\partial o_{11}}{\partial w_{1old}}\right] + \left[\frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial o_{22}} \times \frac{\partial o_{22}}{\partial o_{11}} \times \frac{\partial o_{11}}{\partial w_{1old}}\right]$$

→ Vanishing Gradient Problem



eg of deep gradient neural n/w

Sigmoid Activation [0, 1]

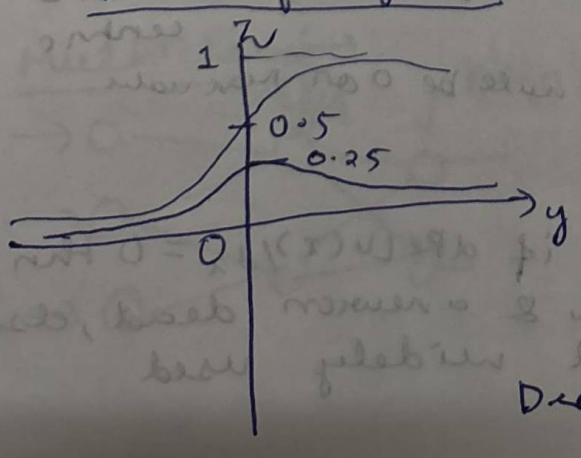
$Loss = \frac{1}{2}(y - \hat{y})^2$ → MSE Loss func

$y = \frac{1}{1+e^{-x}}$ → 0 or 1; ≥0.5 → 1, <0.5 → 0

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

$\frac{\partial L}{\partial w_{1new}} = \frac{\partial L}{\partial o_{51}} \times \frac{\partial o_{51}}{\partial o_{41}} \times \frac{\partial o_{41}}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial o_{21}} \times \frac{\partial o_{21}}{\partial w_1}$

graph of Sigmoid



eg → $o_{51} = \sigma[(o_{41} \times w_4]) + b$

So, the values multiplied in chain will give very small value

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

The derivative of Sigmoid will be b/w 0 - 0.25

Derivatives $0 \leq \sigma(y) \leq 0.25$

$W_{new} = W_{old} - \eta(\text{Small number})$

$W_{new} \simeq W_{old}$ ⇒ Vanishing gradient Problem

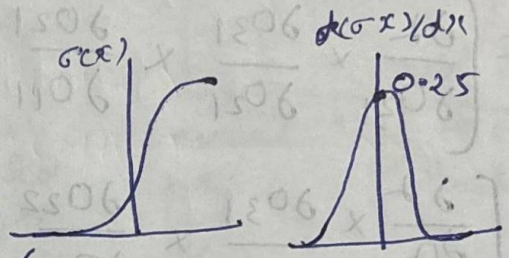
$W_{new} \simeq W_{old}$ ⇒ No change in weights

So, to solve this problem we must use another Activation func^{n}.

Some of the activation func^{n}s are-

1. Sigmoid
2. Tanh
3. Relu
4. Leaky Relu
5. Prelu

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



Not a zero centred curve

→ Smooth gradient, prevents jumbing
→ Fun^{n} having zero centred curve update weights easily
→ Time complexity is high
→ Prone to vanishing gradient

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

[-1 to 1)

drivative (0 to 1)

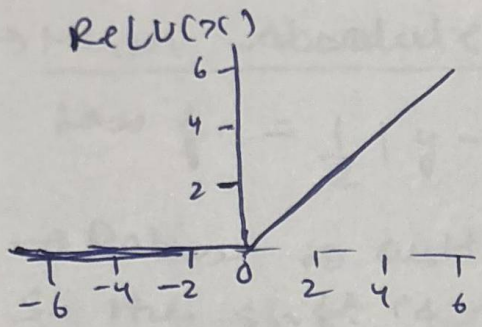
- Still prone to vanishing gradient in case of Deep Neural network
- zero centred
- Hyperbolic tan^{n} fun^{n}

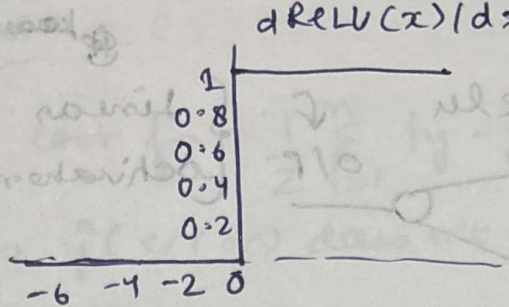
- **ReLU** (Rectified Linear Unit) → quickest, non-zero centric

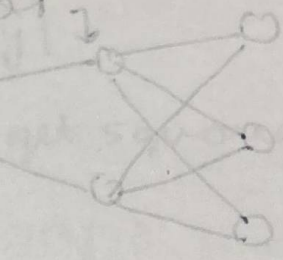
ReLU = max(0, x) → either it will be 0 or max value

dReLU(x)/dx = either 0 or 1

Only one disadvantage is that if dReLU(x)/dx = 0 then it will make again $W_{old} = W_{new}$ & a neuron dead, else it is the best & the most widely used activation func^{n}







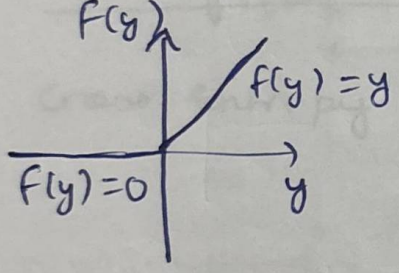
- Leaky ReLU func^n → To solve dead neuron problem

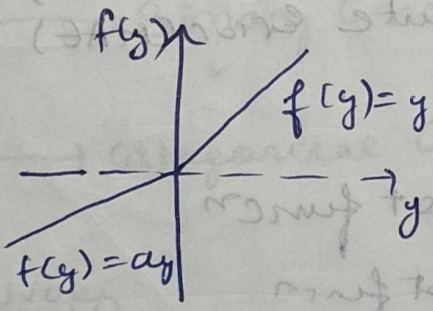
$$f(x) = \max(0.01x, x) \rightarrow \text{never be zero}$$

- ELU (Exponential Linear Units)

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

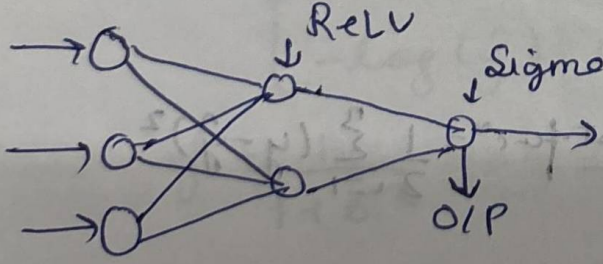
→ ReLU V/S PReLU



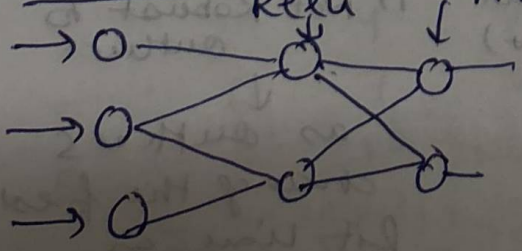


Technique to find which activation func^n to used?

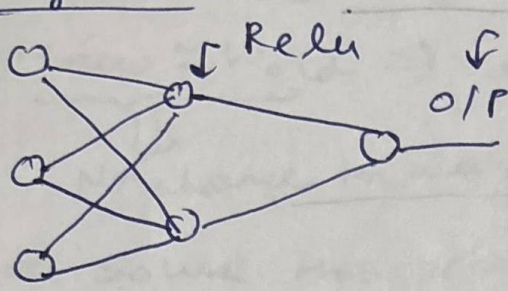
Suppose Binary classification → In these at O/P, Sigmoid Activation must be used & ReLU at Hidden layers or PReLU or ELU



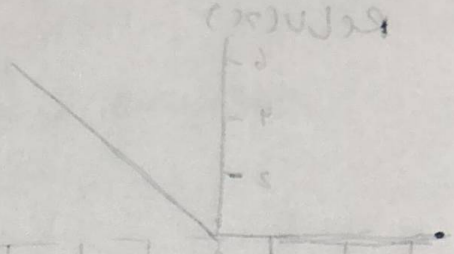
Multi-class → At O/P use Softmax Activation



Regression



{ Linear Activation }



(*) Loss functions

Deep Learning (Artificial Neural Networks)

- Regression
- Classification

For Regression

- Mean Squared Error (MSE)
- Mean absolute Error (MAE)
- Huber Loss

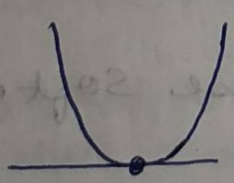
→ Diff b/w Loss & cost func^n

Loss fun^n	Cost fun^n
$\frac{1}{2}(y-\hat{y})^2$	$\frac{1}{2}\sum_{i=1}^{n}(y-\hat{y})^2$
→ it takes a single data point	→ It takes a set of data points

- **MSE** → Loss fun^n $= \frac{1}{2}(y-\hat{y})^2$ Cost fun^n $= \frac{1}{2}\sum_{i=1}^{n}(y-\hat{y})^2$

$(a-b)^2 = a^2 - 2ab + b^2$

Quadratic Eqn ↓ graph



(just like gradient descent)

Advantages of Quadratic Eqn:

1. Differentiable
2. It has only 1 local or global minima
3. converges faster

Disadvantage

1) Not Robust to outliers

↓

as outliers change the best fit line as error $\hat{y}$ gets squared

→ Mean Absolute Error -

Loss fn = $\frac{1}{2}|y-\hat{y}|$ cost fn = $\frac{1}{2}\sum_{i=1}^{n}|y-\hat{y}|$

→ Robust to outliers as $\hat{y}$ (error) does n't get squared So the shift is minor.

→ Time consuming

→ Huber Loss → combination of MSE & MAE & is specifically used when outliers are not there

$$\text{Loss} = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{if } (y-\hat{y}) \le \delta \leftarrow \text{Hyperparameter} \\ \delta(y-\hat{y}) - \frac{1}{2}\delta^2 \end{cases}$$

Classification

cross entropy → Binary Cross Entropy → Binary classification
cross entropy → Categorical Cross Entropy → multiclass classification

i) Binary Cross Entropy

Loss = $-y \times \log(\hat{y}) - (1-y) \times \log(1-\hat{y})$ ⇒ Log likelihood

$$\text{Loss} = \begin{cases} -\log(1-\hat{y}) & \text{if } y=0 \\ -\log(\hat{y}) & \text{if } y=1 \end{cases} \} \text{ Binary classification}$$

$$\hat{y} = \frac{1}{1+e^{-x}}$$

ii) Categorical Cross Entropy -

f_1	f_2	f_3	O/P
2	3	4	good
5	6	7	Bad
0	9	10	Neutral

Good	Bad	Neural
1	0	0
0	1	0
0	0	1

$L(x_i, y_i) = -\sum_{j=1}^{c} y_{ij} \ln(\hat{y}_{ij})$ $\quad y_i = [y_{i1}, y_{i2} \cdots y_{ic}]$

$y_{ij} = \begin{cases} 1 & \text{if Element in class} \\ 0 & \text{else} \end{cases}$

$\hat{y}_{ij}$ = Soft Max Activation

Softmax func^n $\Rightarrow \sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$

Conclusion

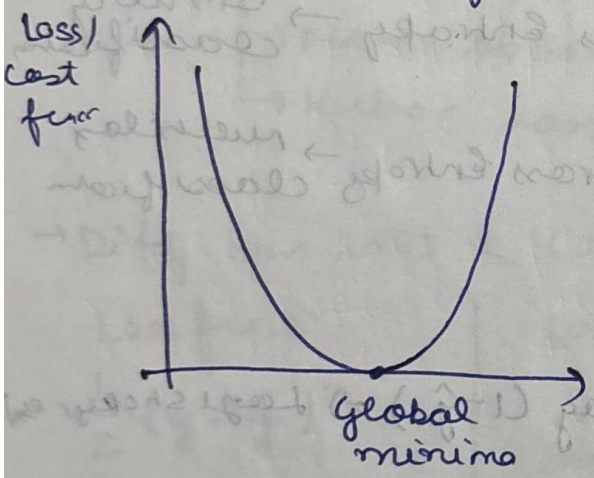
Relu, softmax $\Rightarrow$ Multi-class $\rightarrow$ categorical cross entropy
Relu, Sigmoid $\Rightarrow$ Binary $\rightarrow$ Binary cross Entropy

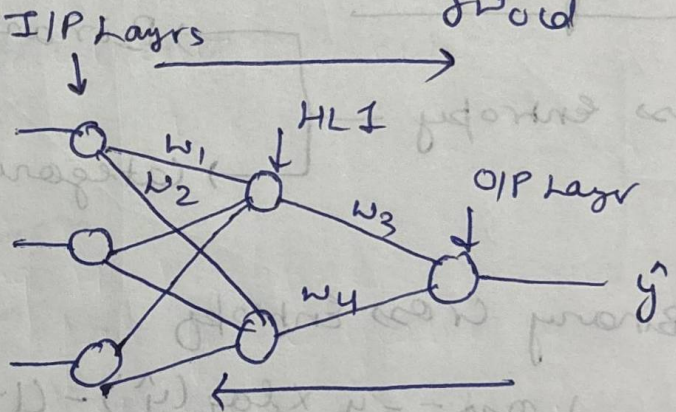
Linear Regression –
Relu, Linear activation $\rightarrow$ MSE, MAE, Huber Loss

Optimizers –

① Gradient Descent :

wt updation formula $\rightarrow W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$





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

$\Uparrow$ MSE

Epoch $\rightarrow$ 1 forward propagation & 1 backward propagation forms 1 Epoch

Generally we pass Million Records i.e. n = 1 Million $\Rightarrow$ 1 Million Record in Each Epoch

Disadvantage of Gradient Descent –
i) Resource Extensive {Huge RAM}

② Stochastic gradient Descent $\rightarrow$ 1 Record in 1 Epoch

* RAM $\downarrow\downarrow$ but convergence will be very Slow.

1 record $\rightarrow \hat{y}$ $\rightarrow$ iteration 1
update wt.

* High Time Complexity

but this will have million Epochs

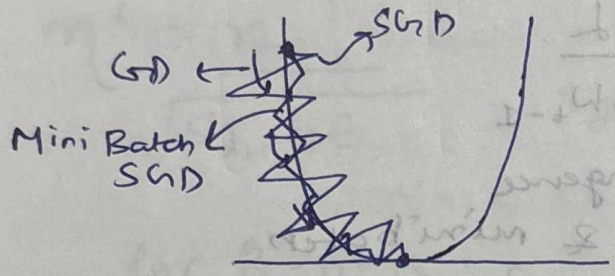
③ Mini Batch SGD (Stochastic Gradient Descent)

→ we will set batch size Suppose → batch-size = 1000

Epoch-1 ⟷ } iteration 1
1000

So for Million records it is 1000 iterations only

* Resource Intensive
* Convergence will be better
* Time complexity will Improve



→ These zig zag is called Noise.
SGD has highest Noise
But still Mini Batch SGD also has some Noise

So, Now how do we remove the Noise ?

⇓

Momentum

Next optimiser -

④

*Mini Batch SGD with Momentum -

→ Exponential Weighted Average
" or Moving " } Also used in Time Series

As we know, $w_{new} = w_{old} - \eta \frac{\partial L}{\partial w_{old}}$

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

t → current time
t-1 → previous time

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

time t_1 t_2 t_3 t_4 -- t_n
value a_1 a_2 a_3 a_4 -- a_n

β → hyperparameter

⇓

β = 0 to 1

Suppose 0.95

$v_{t_1} = a_1$

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

$= 0.95 v_{t_1} + 0.05 a_2$

$V_{t_2} = 0.95 \times V_{t_1} + 0.05\, a_2$

taking $\beta = 0.95$ we are giving more imp. to V_{t_1}

↳ we are removing Noise & smoothening curve

$$V_{t_3} = \beta V_{t_2} + (1-\beta) a_3$$

So, Exponential weighted Avg:

$$\left(w_t = w_{t-1} - \eta V_{dw}\right)$$

here $V_{dw} = \beta V_{dw-1} + (1-\beta) \frac{\partial L}{\partial w_{t-1}}$

This will have → quicker convergence
→ Reduce the noise & minibatch

(5) Adagrad → Adaptive Gradient Descent.

η = fixed ..

* It is slow & fixed
→ can't we do something that initial learning rate becomes fast when far from global minima, & when comes closer then becomes slow?
→ for this is Adagrad

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

$$\boxed{w_t = w_{t-1} - \eta' \frac{\partial L}{\partial w_{t-1}}} \quad \text{where } \eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

↓ small number to avoid divide by zero condn (for cases $\alpha_t = 0$)

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_t}\right)^2$$

So α_t will always increase
So η' will Decrease when we reach global minima

Sometimes α_t can be huge number which may give inputs
η' a negligible change. So, we will use another optimiser to solve this problem.

$$\eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

η → suppose 0.01

huge no ← $\sqrt{\alpha_t + \epsilon}$

$\Rightarrow W_t \simeq W_{t-1}$

6) Adadelta & RMSprop -

$$\eta' = \frac{\eta}{\sqrt{S_{dw} + \epsilon}}$$

Suppose initially $S_{dw_t} = 0$

$$S_{dw_t} = \beta S_{dw_{t-1}} + (1-\beta)\left(\frac{\partial L}{\partial w}\right)^2_{t-1}$$

for $\beta = 0.95$; $S_{dwt} = 0.95\, S_{dwt-1} + 0.05 \left(\frac{\partial L}{\partial w_{t-1}}\right)^2$

→ concept of Exponential weighted average is used

7) Adam Optimiser

→ Solves Smoothening problem
→ Assures that Learning Rate becomes Adaptive

Momentum + RMSprop → Best optimizer

$V_{dw} = 0$ $V_{db} = 0$ $S_{dw} = 0$ $S_{db} = 0$

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

$$b_t = W_{bt-1} - \eta' V_{db}$$

$$\eta' = \frac{\eta}{\sqrt{S_{dw} + \epsilon}}$$

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

$$V_{db_t} = \beta \times V_{db_{t-1}} + (1-\beta)\frac{\partial L}{\partial b_{t-1}}$$

ANN Practical Implementation

Churn_Modeling.csv → Dataset

```
!pip install tensorflow-gpu

import tensorflow as tf
print(tf.__version__)

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

dataset = pd.read_csv('Churn_Modeling.csv')
dataset.head()

X = dataset.iloc[:, 3:13] → 3 to 12 columns

y = dataset.iloc[:, 13]

X.head

geography = pd.get_dummies(X['Geography'], drop_first = True)

gender = pd.get_dummies(X['Gender'], drop_first = True)

X.drop(['Geography', 'Gender'], axis=1)

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

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)
```

ANN {Feature Scaling}

→ For which all algorithms feature scaling is required.
→ ANN, LR, Log.R, KNN, K means } for distance based or gradient Descent

feature Scaling is required

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

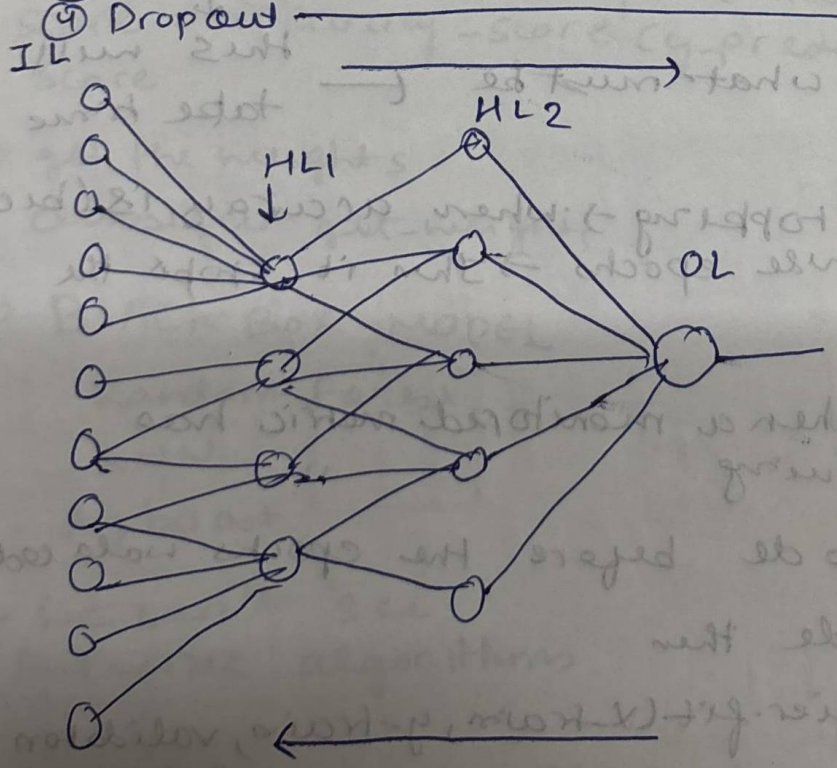
Part 2 Now lets create ANN

Tensorflow => open sourced by Google
PyTorch → by facebook

In version < 2.0 Tensorflow — wrapper, keras
≥ 2.0 Tensorflow + keras

```
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
```

① Sequential – Neural block which can do FP & BP
② Dense – we are able to create Neurons & its layers
③ Activation – To use activation func
④ Dropout



Sometimes, Neural network may lead to overfitting i.e. My training accuracy is high but test accuracy goes down, So we use Dropout layer – which acts as Regularization parameter, which deactivates neurons, & drops out layers

```
# Lets initialise ANN
classifier = Sequential()
# Adding the input layer
classifier.add(Dense(units=11, activation='relu'))
# adding 1st HL
classifier.add(Dense(units=7, activation='relu'))
# 3rd HL
classifier.add(Dense(units=6, activation='relu'))
# Adding O/P Layer
classifier.add(Dense(1, activation='sigmoid'))

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

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

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

(taking only 67% of data) ←

↓ this will take time

So how to know that what must be the no. of epochs? ←

we must use Early Stopping → when accuracy is (becomes) approx constant while epochs → then it stops the epochs at that time.

Stop training when a monitored metric has stopped improving

we must paste code before the epochs wala code.

```
# Early Stopping - code then
model_history = classifier.fit(X_train, y_train, validation_
split=0.33,
batch_size=10, epochs=1000, callbacks=Early_Stopping)
```

model_history.history.keys()
↳ tells what all factors we have focussed on.

```
#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
```

Making predictions & Evaluate Model

```
#predicting the test set results
y_pred = classifier.predict(X_test)
y_pred = (y_pred >= 0.5)
```

make confusion matrix

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

Calculate accuracy

```
from sklearn.metrics import accuracy_score
score = accuracy_score(y_pred, y_test)
score
```

get the weights

```
classifier.get_weights
```

→ BLACK BOX MODEL

- Random Forest
- ANN
- Xgboost

• we can't see internal algorithms

WHITE BOX Model

- Decision tree
- Linear Regression

• here we can see internal algorith

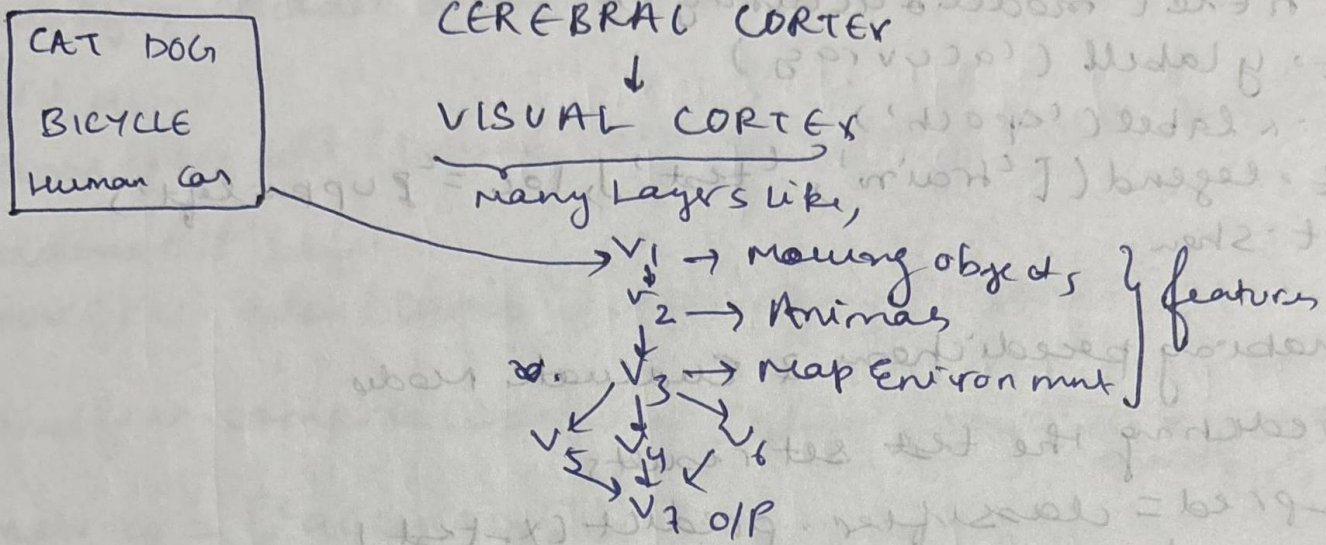
*To look into the Black box Models nowadays Explainable AI is being discovered.

Convolution Neural Network (CNN)

ANN → Theoretical, practical

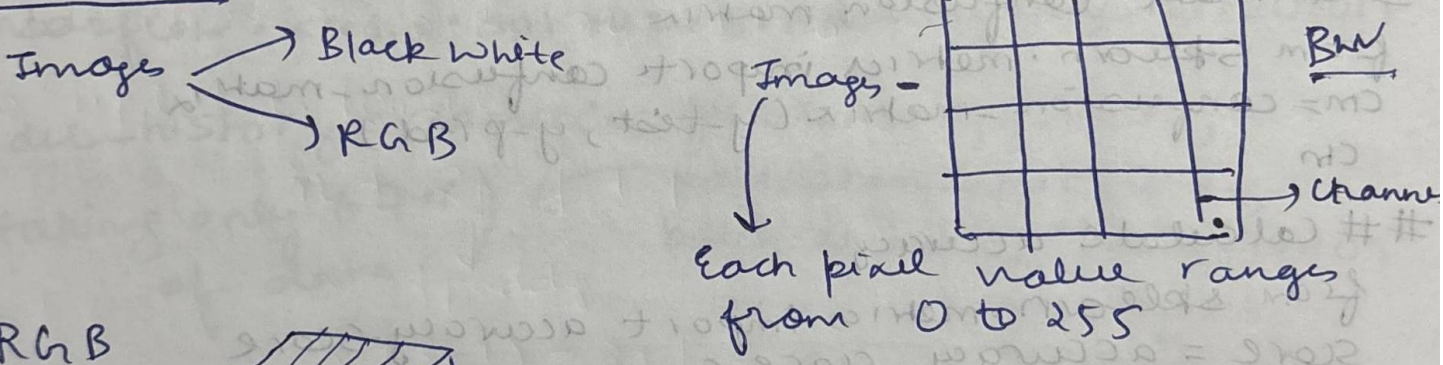
CNN → Image, video frame

CNN V/S Human Brain

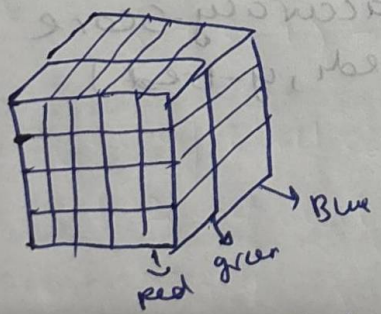


CNN is divided into following steps-

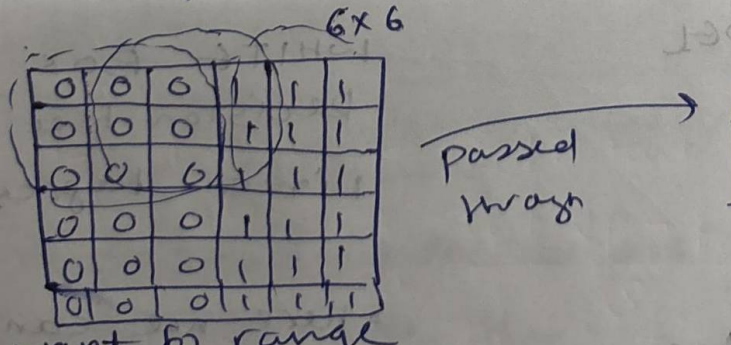
i) Convolution



RGB



Convolution



If we want to range each pixel value b/w 0 to 1 be must divide Each pixel value by 255 → This is called as Min / Max scaling

Suppose that filter is placed at 1st quarter of image.

→ we will multiply each cell value to other & then add it (0+0+0+0+0+0+0+1) = 0

& then 1 step to right (suppose stride = 1) if, Stride = 2 then 2 steps)

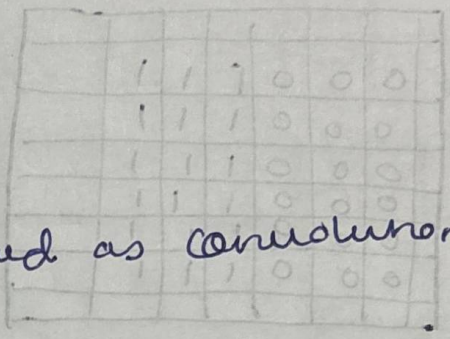
② 0+0+1+0+0+0+0+0−1 = 0

③ 0+2+1+0+0+0−2−1 = 0

④ → 0

O/P ⇒

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0



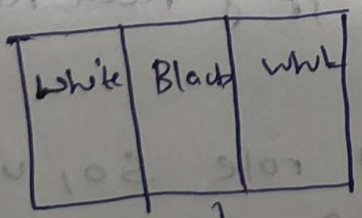
This operation is termed as convolution

Suppose O/P = $\begin{bmatrix} 0,-4,-4,0 \\ 0,-4,-4,0 \\ 0,-4,-4,0 \\ 0,-4,-4,0 \end{bmatrix}$ & the filter applied is vertical Edge filter

After min Max scaling highest value = 255
lowest value = 0

⇒ $\begin{bmatrix} 255,0,0,255 \\ 255,0,0,255 \\ 255,0,0,255 \\ 255,0,0,255 \end{bmatrix}$

255 → white color
& 0 → Black color



vertical Edge filter

→ Similarly many types of filters → which can take out specific information from a specific image.

6 × 6 image → 3 × 3 filter → O/P = 4

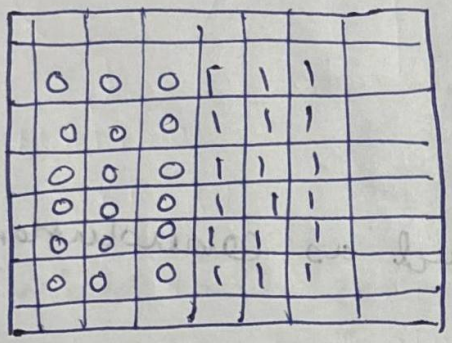
n = 6 f = 3

$$O/P = n - f + 1$$

In case of $6 \times 6 \rightarrow 3 \times 3$ filter $\rightarrow$ O/P 4×4
So as we see our image size is reducing & this should not happen that means we are losing some kind of information - In order to prevent that we can do padding i.e. building a specific compound around the image

↓

process of applying a layer or the top of image



Now we can fill that by anything may be zero or nearest value

↓

zero padding

Now $n = 8$ so $n - 3 + 1 = 6$

So updated formula

$$n + 2p - f + 1$$

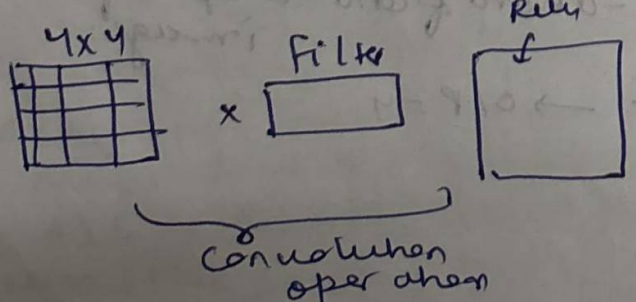
↓

paddy

$$6 + 2(1) - 3 + 1 = 6$$

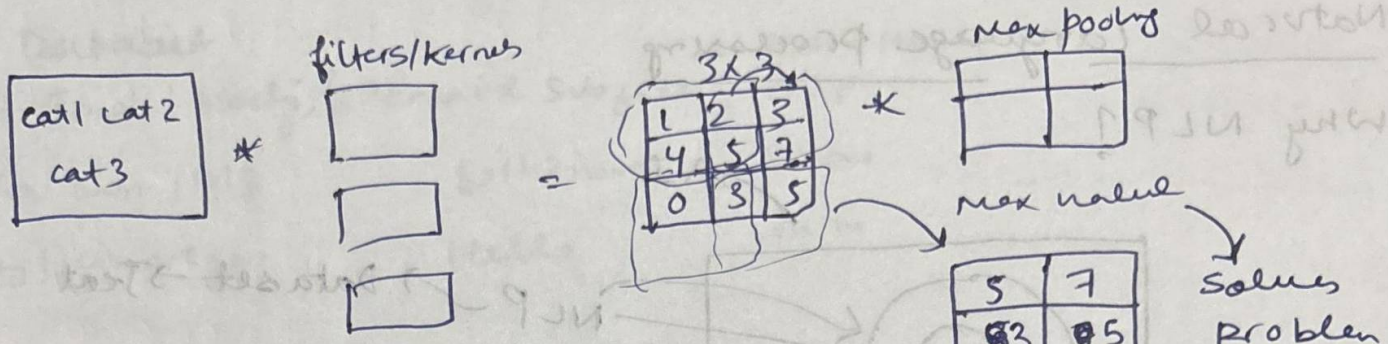
In ANN, we updated weights, here in back propagation we update filters based on I/P image
→ on O/P on each value we apply Relu Activation func^n

Stride also plays an important role so, updated formula $\Rightarrow \dfrac{n + 2p - f + 1}{S}$



② Max pooling -

pooling is of 3 types → Avg pooling, Min pooling, Max pooling



Solves problem of Locaten invariant

Avg pooling → Avg of values

Min pooling → Min of values

→ Flatening layer -

ANN Dense layer

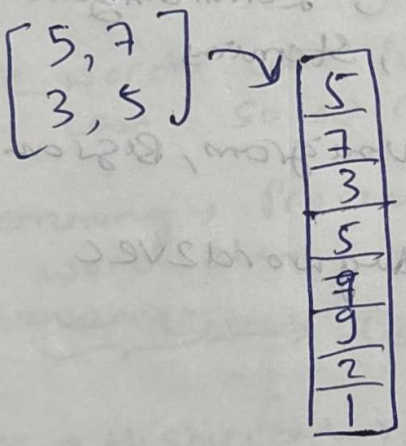


Image classification