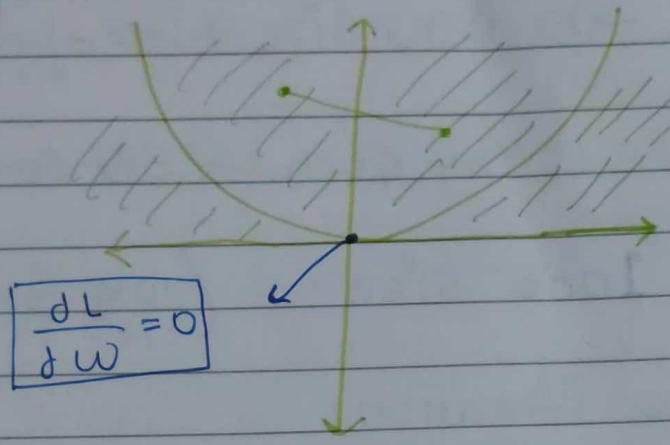
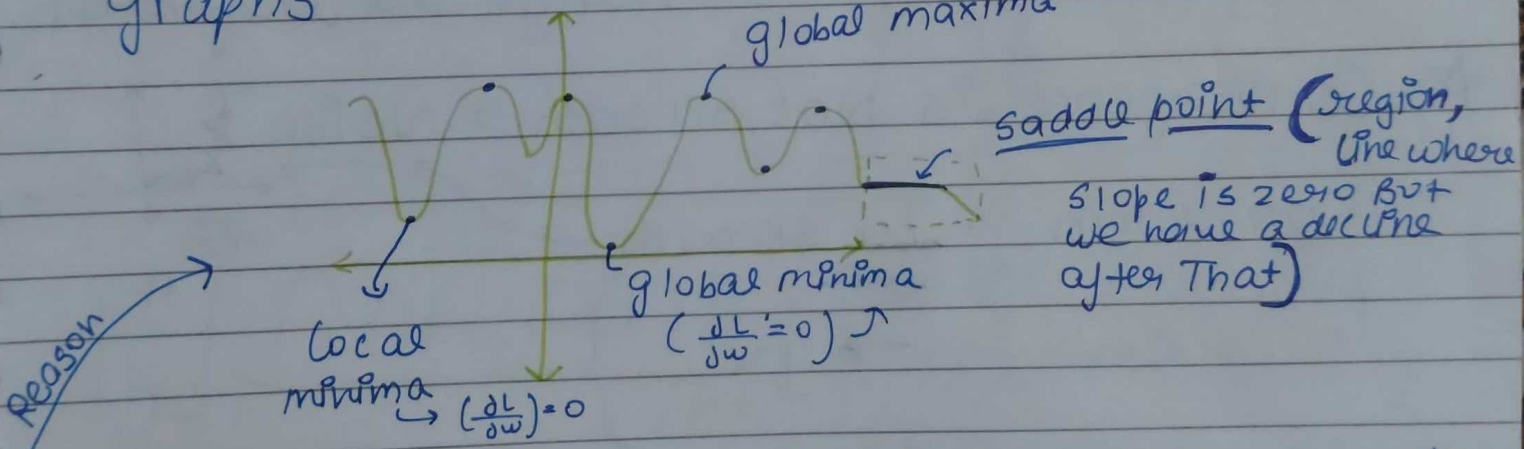
In ML → most/all the times we get Convex graph



[2 points conn, lies in same Region]

In DL → most of the time we get Non-convex graphs



Because of Diff loss func and so many neurons, when weight has to be optimised for every point.

* Derivative → of loss wrt w_{old} at minima point is zero
  ↳ it means $w_{new} = w_{old}$ and we now know we have reached an optimal soln.

Problem ↓

In DL-curve → the derivative at local minima is zero but we haven't reach, the global minima

↳ Solution → Advance optimisers (apart from GD, SGD, MBSGD) and Adv Loss func.

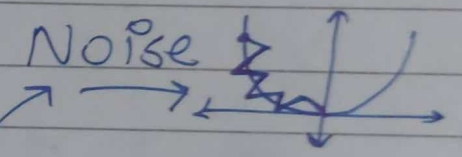
→ SGD with Momentum

- In MB SGD → we take Batch Size → loss $= \frac{1}{2}\sum_{i=1}^{K} (y-\hat{y})^2$ $(K<n)$
  - ↳ 10K records so every epoch we set some size
    - Epoch = 1000 size $\left(\frac{10K}{1K} \Rightarrow 10\right)$

1 Epoch = 10 iteration.
20 Epoch = 200 iterations

- iterations are in Mid-Range and lil faster.

☆ Convergence is Not Smooth → Noise



↳ Momentum → To smooth

↳ $w_t = w_{t-1} - \eta \frac{dL}{dw_{t-1}}$

we will calculate → Exponential weighted Average

- EWA → Calculating the 'new value' on the Basis of importance of Previous Value.

→ $t_1\ t_2\ t_3\ t_4 \ldots t_n$ ↗ iteration
→ $a_1\ a_2\ a_3\ a_4 \ldots a_n$ ↳ value

→ $V_{t_1} = a_1$
→ $V_{t_2} = \beta . V_{t_1} + (1-\beta) a_2$
→ $V_{t_3} = \beta . V_{t_2} + (1-\beta) a_3$.

[β = importance param
if $\beta = 0.9$ ($a_1 \uparrow$ and $a_2 \downarrow$)
if $\beta = 0.1$ ($a_1 \downarrow$ and $a_2 \uparrow$)]

EWA

$W_t = W_{t-1} - \eta V_{dw}$

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

$V_{dw_{t-1}} = \beta V_{dw_{t-2}} + (1-\beta)\frac{\partial L}{\partial w_{t-1}}$

deg → $\beta = 0.95$

init → $V_{dw} = 0$.

$\frac{\partial L}{\partial w_{old}}$

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

$B_{new} = B_{old} - \alpha \frac{dL}{dB_{old}}$

Bias

Similar steps for weights and Bios

4th] ADAGRAD [Adaptive Gradient Boosting

- Because of this, we stoped using EWA

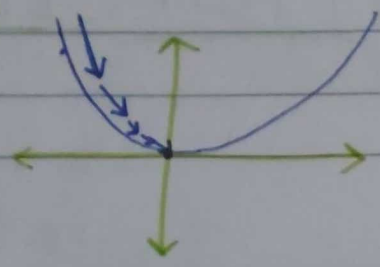
1) here our Learning Rate is NOT FIXED

2) Our α keeps changing itself as the time goes in training

3) At first → Big Jump and then small ones

α → Change dynamically

$\alpha = 0.5 \rightarrow 0.4 \rightarrow 0.3 \rightarrow 0.1$
$0.01 \leftarrow 0.05 \leftarrow 0.09$



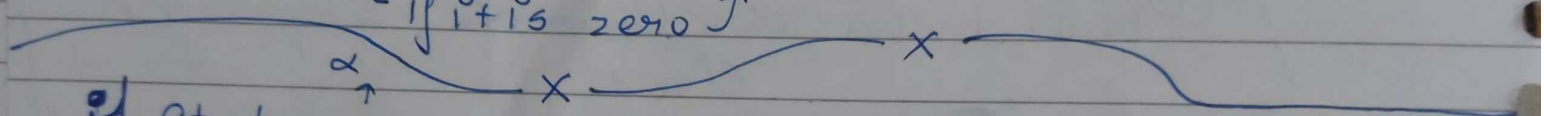
- t = new → curr iter
- t-1 = old → prev iter

GD → $W_t = W_{t-1} - \alpha \frac{\partial L}{\partial W_{t-1}}$

AGD → $W_t = W_{t-1} - \alpha' \frac{\partial L}{\partial W_{t-1}}$

$\alpha' = \frac{\alpha}{\sqrt{\eta_t + \epsilon}}$ (α: initial value) → $\eta_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial W_t}\right)^2$

(ϵ: small +ve number, if it is zero)



If at $t_1 = 0.001 \rightarrow t_2 = 0.009 \rightarrow t_3 = 0.007, \rightarrow t_4 = 0.005$

To ↓ α', η_t ↑ (Σ)

X X X

PROBLEM → In deep multi-Neural, after some time the value of η_t will be Huge and α' will be very very small → $W_{new} \sim W_{old}$ (Vanishing Prob)

due to learning Rate ↓ NO convergence

X X X

Solution →

5th] ADA DELTA and RMS PROP

Recap

restricting large values

$$\alpha' = \frac{\alpha}{\sqrt{S_{dw} + \varepsilon}}$$

initially $S_{dw} = 0$
Just like EWA / SGD momentum

Moving average = $S_{dw_t} = \beta S_{dw_{t-1}} + (1-\beta)\left(\frac{\partial L}{\partial w_t}\right)^2$
(EWA)

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

★ Smaller value to restrict larger value of squared values.

$$W_t = W_{t-1} - \alpha' \frac{\partial L}{\partial w_{t-1}} \qquad \beta_t = \beta_{t-1} - \alpha' \frac{\partial L}{\partial \beta_{t-1}}$$

→ On iteration t in mini Batch

↳ Compute $\frac{\partial L}{\partial w}, \frac{\partial L}{\partial \beta}$ on current mini Batch

↳ $S_{dw_t} = \beta S_{dw_{t-1}} + (1-\beta)\frac{\partial L}{\partial w_t}$

↳ $S_{db} = \beta S_{db_{t-1}} + (1-\beta)\frac{\partial L}{\partial b_t}$

def
$\beta = 0.95$
$S_{dw} \rightarrow$ init $= 0$

→ $w_t = w_{t-1} - \alpha' \frac{\partial L}{\partial w_{t-1}} \qquad b_t = b_{t-1} - \alpha' \frac{\partial L}{\partial b_{t-1}}$

X — X — X

everytime take mini-Batch of data

X — X — X — X

★ 6] ADAM OPTIMIZER

{Adaptive Moment Estimation}

restricting η_t value with help of S_{dw} → EWA

→ Rms prop + Momentum (smooth) | combine = V_{dw}, V_{db}, S_{dw}, S_{db}

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

on Iteration t ↓

2] Compute $\frac{dL}{dw}$, $\frac{dL}{db}$ using current mini Batch

3] $V_{dw_t} = \beta_1 V_{dw_{t-1}} + (1-\beta_1)\frac{dL}{dw_t}$

$V_{db_t} = \beta_1 V_{db_{t-1}} + (1-\beta)\left(\frac{dL}{d\beta_t}\right)$

$S_{dw_t} = \beta_2 S_{dw_{t-1}} + (1-\beta_2)\left(\frac{dL}{dw_t}\right)^2$

$S_{db_t} = \beta_2 S_{db_{t-1}} + (1-\beta_2)\left(\frac{dL}{d\beta_t}\right)^2$.

4] → $w_{old/t} = w_{t-1} - \alpha' V_{dw}$

or ↓

$w_t = w_{t-1} - \eta' V_{dw}$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{S_{dw} + \varepsilon}} \cdot V_{dw}$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{S_{dw_{t-1}} + \varepsilon}} \cdot V_{dw_{t-1}}$$

$$b_t = b_{t-1} - \frac{\eta}{\sqrt{S_{db_{t-1}} + \varepsilon}} \cdot V_{db_{t-1}}$$

· Bias correction ·

→ $V_{dw}^{corrected} = \frac{V_{dw}}{(1-\beta_1^t)}$ | $V_{db}^{corrected} = \frac{V_{db}}{(1-\beta_1^t)}$

$$S_{dw}^{correction} = \frac{S_{dw}}{(1-\beta_2^t)} \quad \& \quad S_{db}^{Corrected} = \frac{S_{dw}}{(1-\beta_2^t)}$$

New formula

- $w_{new} = w_{old} - \frac{\alpha}{\sqrt{S_{dw}^{corrected} + \varepsilon}} \cdot V_{dw}^{corrected}$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{S_{dw_{t-1}}^{corrected} + \varepsilon}} \cdot V_{dw_{t-1}}^{corrected}$$

- $\beta_t = \beta_{t-1} - \frac{\alpha}{\sqrt{S_{db_{t-1}}^{corrected} + \varepsilon}} \cdot V_{db_{t-1}}^{corrected}$,

---

DIFF TYPES OF Loss FUNCTION

1) Regression →
   - Squared error loss (singe point)
     - $L = (y-\hat{y})^2$
   - Mean Squared error [MSE] → $\frac{1}{m}\sum_{i=1}^{i=m}(y_i-\hat{y}_i)^2 = T$
   - Absolute Loss error [MAE] → $\frac{1}{m}\sum_{i=1}^{i=m}|y_i-\hat{y}_i|$ | $|y-\hat{y}|$ as L

★ 'MAE is more Robust to Outliers compared to MSE'.

(3) Huber loss → MSE + MAE

Hyperparameter.

$$\begin{cases} \frac{1}{2}(y-\hat{y})^2, & \text{if } |y-\hat{y}| \le \delta \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

→ Fix all prob of MSE and MAE.
→ Prob of outliers is fixed

• CLASSIFICATION

→ 1 cross Entropy (in log Reg usecase)

$$L \Rightarrow -y * \log(\hat{y}) - (1-y) * \log(1-\hat{y})$$

$y = 0 | 1$

Binary cross Entropy

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

$\hat{y}$ = sigmoid. → $\frac{1}{1+e^{-z}}$ | $\frac{1}{1+e^{-y}}$

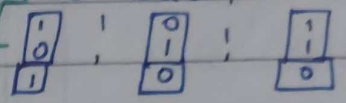
Categorical → O/P → Vector → One Hot Encoding.

- Multi°-Class Cross Entropy loss

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

→ Perform one-Hot Encoding of output feat. -①

- O/P → multiclass → O/P layer
  - Good, Bad, Neutral → One hot encoding
    - Good : Bad : Neutral



f_1	f_2	f_3	O/P
.	.	.	.

↓ one hot

[O/P → unique features → own column]

	y_{11}	y_{12}	y_{13}	$[y_{ij}]$
f_1 f_2 f_3	y_1	y_2	y_3	
. . .	[1,	0,	0]	
. . .	[0,	1,	0]	
	y_{21}	y_{22}	y_{23}	

[i° = row number
j = no. of Bits → Colums
c = categories/Bits]

* We use SOFTMAX act. func for multi°-class Problems.

$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{c} e^{z_j}} \rightarrow [P_1, P_2, P_3 \ldots P_c] \rightarrow \text{majority wins.}$$

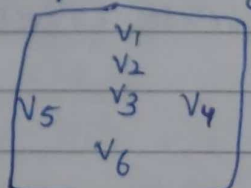
└ Prob value

- To calculate $\hat{y}_{ij}$ → SOFTMAX → $\frac{e^{10}}{e^{10}+e^{20}+e^{30}+e^{40}}$ → Probability

Backtrack ← Calculate ← Put in Loss

CNN

Filters in CNN



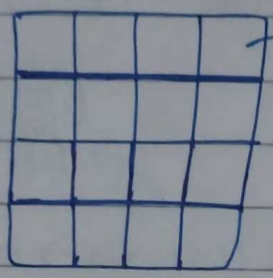
HUMAN BRAIN

image Detection → eyes → Brain → Cerebral Cortex → Visual Cortex → various layers that detect Diff feat of images

Convolution in CNN.

→ Basics About Gray scale Image

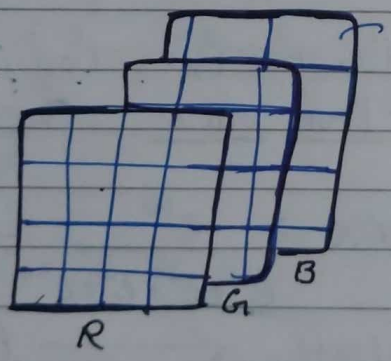
- Define as a x a [eg.→ 4x4, 6x6 etc..] [pixels x pixels]



Values Ranges B/w 0 - 255

→ Basics About RGB / Color Image

- Stacked up windows / matrix / matrices of (values / pixels)



4x4x3 → 3 → Channel

Values Ranges B/w 0 - 255

* These pixel values can be Scaled down.

Normalization ← / → Standardization.

* Convolution: Putting filter over the channel. and moving it by 1 (→ and ↓)

6x6 → Gray scale
↳ 0-255
↳ Normalized
↳ 0 (white)
↳ 1 (Black)

Filter → layer
↳ To detect edges / things in a image.

Here We use Filter

Filter
↳ Vertical edge Detector → To detect vertical edge

1	0	-1
2	0	-2
1	0	-1

filter → vertical edge Detector

- Generalized STEPS For ALL FILTERS

So **STEP 1** → Place filter over the image Pixel Window

So **STEP 2** → multiply respective cell values and apply Σ on it / them

So **STEP 3** → use That value as a cell of the Result Channel / window

So **STEP 4** → Move / **Stride** one Step ~~Left~~ Right / Left and Repeat STEPS 1 to 3. Once hit Boundary move 1 Step Down and Repeat STEPS 1 to 3. [STRIDE AND PADDING]

FINAL STEP ↓

• Output →

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0
0	-4	-4	0

4 X 4

Normalized the Output to convert -4 to 0 (white) and 0 to 255 (Black)

↓

255	0	0	255
255	0	0	255
255	0	0	255
255	0	0	255

vertical edge Detection.

* filter for Horizontal edge Detection will look like → 3X3 →

1	2	1
0	0	0
-1	-2	-1

[Transpose of vertical edge detector]

Formula → $n - f + 1$ (6 - 3 + 1 → 4)

• We loss some Info about the Image if we Stride by 1 Step. (from 6x6 to 4x4).

• To prevent It from Happening we a Use The Concept of PADDING

Let's say → we want [6 x 6 → 6 x 6] (No loss)

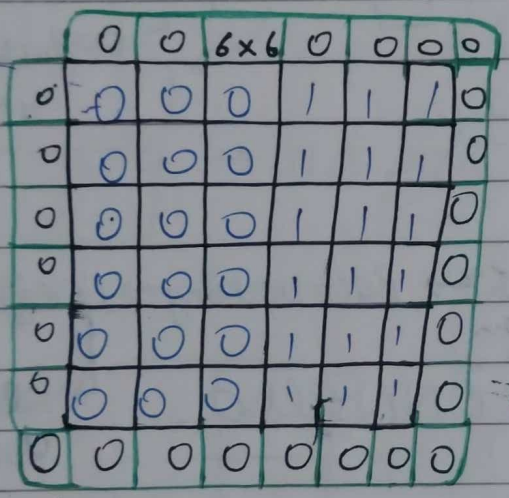
So → $n - f + 1 = 6 \Rightarrow n = 6 + 3 - 1 \Rightarrow n = 8$

• So we have to Convert our 6x6 to 8x8 in order to prevent loss of Data.

• **PADDING** → convert our image to desirable dimension so to prevent any loss of Data during convolution.

→ P = 1 [Adds 1 column to Right and left AND Adds 1 rows to Right and left of our original Data] image]

Padding P=1



To fill values in New col and rows
- 0 Padding
- Neighbour Padding

• 0 Padding → Fill 0's everywhere in new cells
• Neighbour Padding → Fill the nearest value in the new cell (comp to values in orignal image).

---

n-f+1 → After Padding → $n+2P-f+1$

• Inf loss → $n-f+1$ [Stride]
• No Inf loss → $n+2P-f+1$ [Stride + Padding]

less edges

more edges.

Backpropagation in CNN → Update Weights/Value of Filters.

- **Operation of CNN** (Backpropagation and filter updation)
  - Values
- filter = Kernel
- Multiple filters

- Just like ANN → $Z = w_1 x_1$ → $Z = Act(z)$ (relu)
  - in CNN → Image ☐ * filter ☐ → output
    - output = Act(z) (relu (values of output))

$$n - f + 1$$

→ Same (loss func → optimizer → Backtracking → weight upd)
- happens in CNN also.
- Value updation

MAX POOLING

→ either we can stack multiple convolution operations or stack multiple filters, or we can also apply MAX POOLING.

* Location Invarient → some neurons get automatically updated when they see an Image.

* MAX POOLING → A layer or Kernal or Tensor (n-dimensional matrix/Array) Applied after 1/multiple convolutional operations.

$n - f + 1$ $\quad (P=0) \rightarrow$ info loss

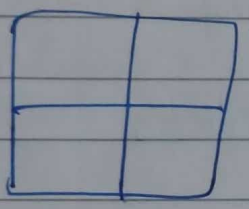
$4 - 2 + 1 \Rightarrow 3$

↳ Let's ⇒ 2×2

$4\times4 \xrightarrow{\text{filter}} 2\times2 \xrightarrow{\text{Result}} 3\times3 \xrightarrow{\text{MAX POOL LAYER}}$

- In maxpool → stride = 2 (or [stride = size of maxpool layer])

max (1 2 3 / 4 3 6 / 2 8 4) → Apply maxpool layer and consider the maximum value from that Tensor or Kernel.



→ maxpool of 2×2, so stride = 2

↓

output → 4 6 / 8 4

Max Intensity value is picked from the convolution operation.

↳ So Proper detection.

- These can be stacked. (Horizontally or Vertically)
- Some ex → Min Pooling / Avg Pooling / Max pooling.

DATA AUGMENTATION

- Transforming the same Image / Data into Diff forms.
- Flip → Horizontal shift → Vertical shift → Zooming → ADDNOISE.

Why? → To ↑ Variance in data so that our model performance can ↑ and can be Robust to any way the Data is presented to it.

• Code snippet — Data Augmentation

• from keras.preprocessing.image import ImageDataGenerator
↓ main library.

• from tensorflow.keras.utils import img_to_array, array_to_img, load_img

• datagen = ImageDataGenerator (→ #parameters
rotation_range = 40,
width_shift_range = 0.2,
height_shift_range = 0.2,
shear_range = 0.2,
zoom_range = 0.2,
horizontal_flip = True,
fill_mode = 'nearest')

• img = load_img("Directory towards Image")

→ x = img_to_array (img) → R, G, B → 3D → 3 Channels
↓ make it 4D

→ x = x.reshape ((1,) + x.shape)
(1, 3D size).

flow generates batches of randomly Transformed images and saves the result to the specific directory.

• for batch in datagen.flow (x, batch_size = 1, save_to_dir = " ", save_prefix = "", save_format = "").

Transfer Learning → Pre-trained model → our model.

- Using Pre-trained Models to inc the performance of our model.
- The Pre-trained models are trained on millions of images so it ↑ the accuracy of our Model.
- We can use a pretrained CNN, one trained on Millions of images, as start of our Model. This will allow us to have a very good convolution Base Before Adding our own Dense Layered classifiers.

*Fine tunning → Applying Transfer learning, we often want to Tweak the final layers in our Conv Base to work Better for our problem.

→ Only adjusting the final few layers → Does not affecting the Prev layers

→ To look for only features relevant to our very specific problem.

TL → Pre trained Model → Our Model → Fine Tunning

[After Training the Base Model]

Freezing The Base (Do not train the layers of Base model)

The term freezing refers to Disabling the training property of a layer

→ we won't make any changes to the weights of any layers that are frozen during Training.

→ IMP AS WE DON'T WANT TO CHANGE THE CONVOLUTIONAL BASE THAT ARE ALREADY HAS LEARNED WEIGHTS

ADD OUR CLASSIFIER

Binary class → binary_cross entropy (loss)

Multiclass → Sparse_Categorical_Crossentropy (loss)

Model

Basic ANN Prep

→ Always Normalized / Standarised X_train and x_test.

→ Import tensorflow as tf.

model = tf.keras.models.Sequential()

Input layer ↴

model.add (tf.keras.layers.Dense(units=10, kernel_initializer="he_normal", ← optional
activation="relu",
input_dim = 13))

model.add(tf.keras.layers.Dropout(value)) —— ①

Further layers. (N-layers)

↳ model.add(tf.keras.layers.Dense(units, activation)) *
— ① (use ①)

Output layer

↳ model.add(tf.keras.layers.Dense(units, activation)
↳ sigmoid/softmax

— X — X — X —

model.compile (optimizer, loss, metrics).

— X — X — X — ← optional

model.fit (X_train, Y_train, validation_split = 0.30, Batch_size,
epochs = 30)

— X — X — X —

Basic CNN Prep

Normalize img array → /255.0

x_train and x_test → (1)

reshape.(len(ara), size)

* (2) eg → [28, 28, 1]

model = tf.keras.model.sequential()

model.add(tf.keras.layers.Conv2D(filters, kernelsize, activation, input_shape)

Further layers → model.add(tf.keras.layers.MaxPool2D(size)) → (1)

↳ similar to Dropout

↳ model.add(tf.keras.layers.Conv2D(filter, size, activation)

use (1)

model.add(tf.keras.layers.Flatten()) → makes 1D for ANN

Add ANN layers → Dense and Dropout and output layer

Same model.compile and model.fit

* Hyperparameter tunning using Keras_tuner

!pip install keras-tuner --upgrade

CNN → all this in dog json(hp). | in layers we add hp

filters = hp.Int("name", min_val, max_value, step)

Kernel_size = hp.Choice("name", values = [a, b]) (axa or bxb)

activation and input_shape remains same

ANN

units = hp.Int("name", min_value, max_value, step)

activation remains same

from keras_tuner import RandomSearch

from keras_tuner.engine.hyperparameters import HyperParameters

ts = RandomSearch(func_name, objective='val_accuracy', max_trails = 5, directory="", project_name)

ts.search(X_train, Y_train, epochs, validation_split)

* ts.get_best_models(num_models=1)[0] (last step)