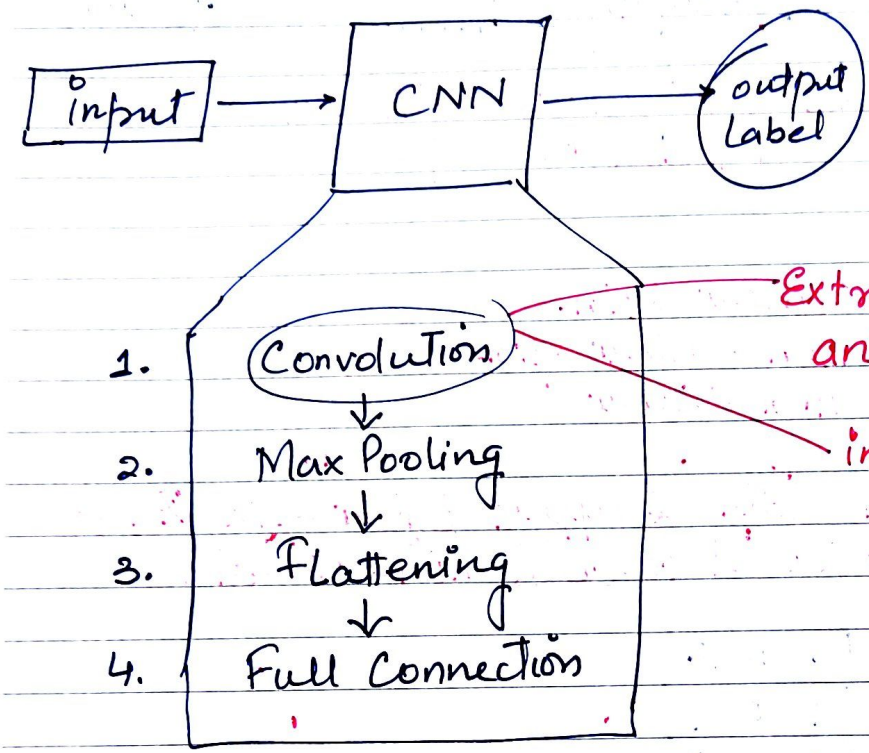
Convolution NN.

- supervised
- task: classification

] visual cortex

probability of classes that best describes the image.



Extract features from an image

image ⊗ [filter/kernel / feature detector] = feature map

# of filters = depth

dot product (matching)

Convolved feature or Activation Map or Feature Map

Each kernel produces a feature map.

e.g if kernel depth = 3, then # of feature maps = 3

depth of feature map = 3

filters ∝ accuracy

depth ∝ accuracy

Stride = slide. filter

Stride = 1, move one pixel ~~from~~ from left to right.

Stride = 2, move two pixels

greater Stride, smaller feature map.

~~down sampling~~.

Padding.

① zero

②

Padding

4×4 input → 6×6 → 3×3

output = 4×4

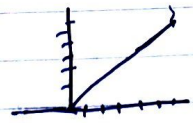
(non-Linear operation)

conv. ⊗ $y[n] = \sum_{k=-\infty}^{\infty} x(k)\, h(n-k)$

ReLu Layer? why? introduces non-linearity in ConvNet

Rectified Linear Unit ← used to remove or avoid vanishing gradient problem (Slow learning ↓ sigmoid)

Output = Max (0, input)



replace (-ve values) by zero. in feature map.

Convolution — Linear Optn.
ReLu — non-Linear Optn.
Real data — non-Linear

∴ conv takes real data & apply Linear optn ⊗, so ReLu is introduced To add non-Linearity.

-1	3	2
1	-2	1
-1	1	3

ReLu →

0	3	2
1	0	1
0	1	3

Max Pooling : input : feature map.

(downsampling / subsampling) of feature map.

retain most important info & reduces dimensionality.

Types : Max, Min, Average, Sum etc.

works better in practice

Pooling work:

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

4×4 feature map

max pool with 2×2 kernel & stride = 2 →

6	8
3	4

4 stride = 2
kernel size = 2×2 →

6	7	8
6	7	8
3	3	4

Flattening :- input : pooled feature Map.

6	8
3	4

pooled feature map

→

6
8
3
4

or

6
3
8
4

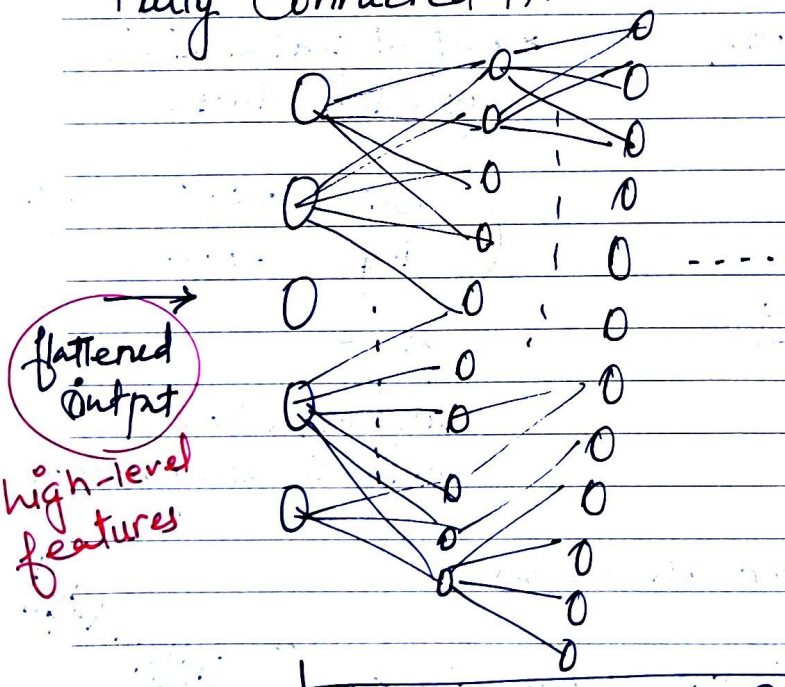
depends on how we choose to do it

input to ANN.

↑ used for Backpropogation

Fully Connected ANN

[Fully Connected Layer is a cheap way to learning non-linear features]



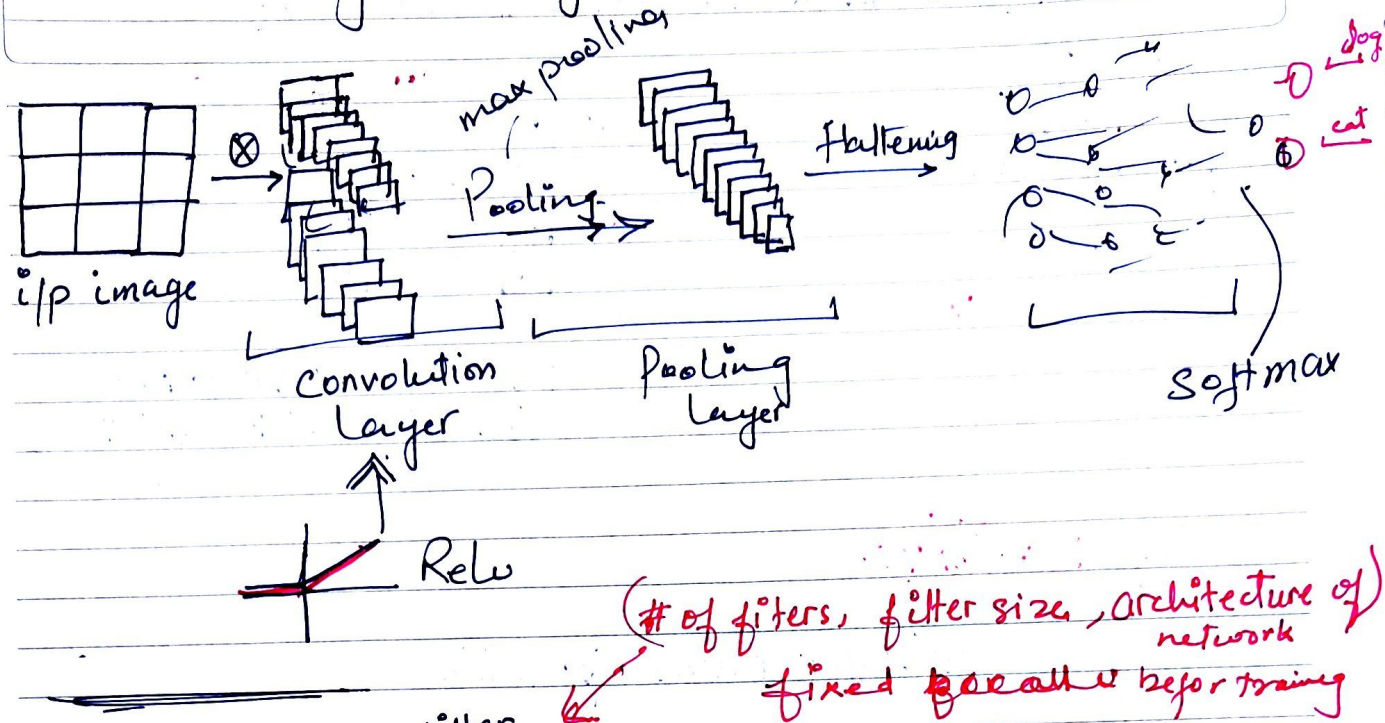
Multi-Layer Perceptron

Arbitary vector of values → Softmax( ) → Squashed values of vector b/w 0 & 1 to sum all values to 1

Assign prob. to each class & sum of all prob = 1

(used for multiclass classification)

Training Using Backpropogation



(# of filters, filter size, architecture of network) fixed beforehand before training

① Initialization of filter parameters (weights) — randomly

② i/p image (one at a time) → Network → output probabilities of each class.

forward propagation
- convolution
- Relu
- pooling
- fully conn.

eg.
cat — 0.8
dog — 0.2

random, b'coz weights are random

③ Calculate total error over all classes

$$\text{Total error} = \sum \frac{(\text{target prob} - \text{output prob})^2}{2}$$

simply derivative of this

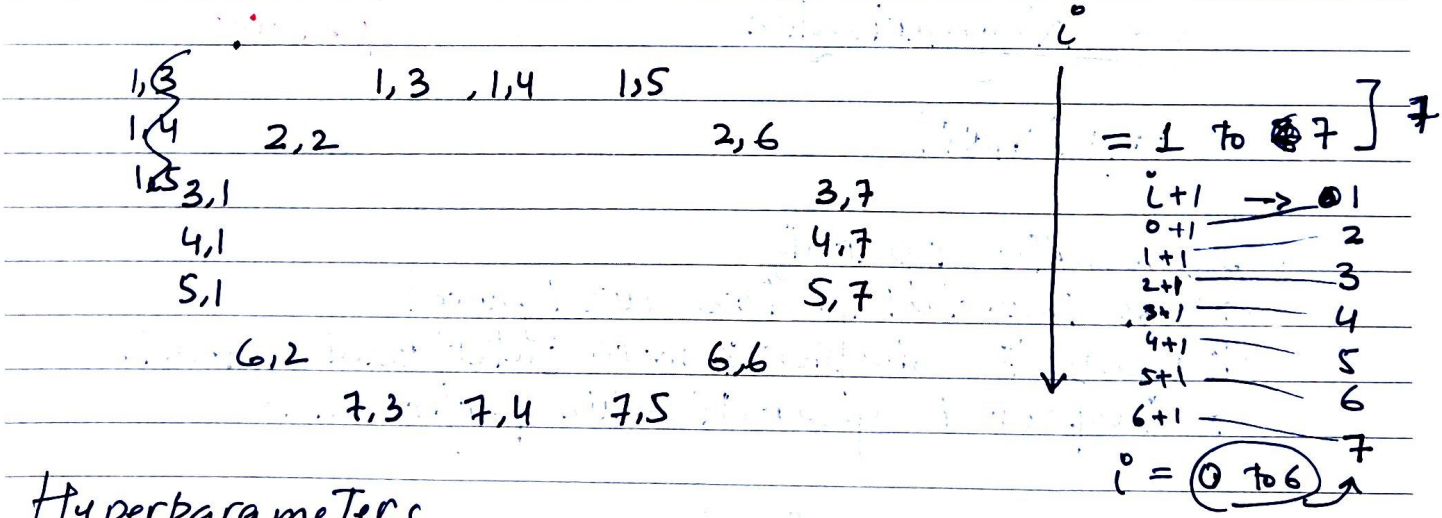
③ Use back propogation to calculate the gradients of error concerning all weights in the network.
- Use gradient descent to update all filters weights & parameter values to minimize the output error
(weights are updated in proportion to their contribution to the total error)

④. When the network is inputted with this image again & the output probabilities are nearly equal to the target probs, then the network learns to classify this particular image correctly.

⑤ Repeat step 2 to 4 for all images in training set.

How CNN works.

→ Build small CNN as defined in the architecture
→ Select images to train the CNN
→ Extract feature map
→ Implement convolution layer
→ Apply Relu on con. layer
→ Apply max pooling on Con layer
→ Make a fully connected layer
→ Then input an image to CNN to predict the image cond
→ Back propogate to calculate error.



Hyperparameters

① # neurons in dense layer.* ⟺ → can be chosen based on the output of flattened layer
② Filter size
③ # layer
④ Pooling operation
⑤ Activation fxn
⑥ Non-linear fxn.
⑦ Padding Type
⑧ Stride

overfitting ← more neurons
underfitting ← less neurons

→ usually greater than # of features (flattened)

Penalizing = impose cost on certain aspects to encourage desired behavior

often applied to weights of model's layers (fully connected as well) conv. layer Recc. Lay.

L2 Regularization

↓ to avoid under overfitting.

$= (\lambda) \sum w_i^2$ ← weights

↑ Hyperparameter

↓ added to original loss function to avoid overfitting during training.

Regularized loss = original loss + L2 Regularization term

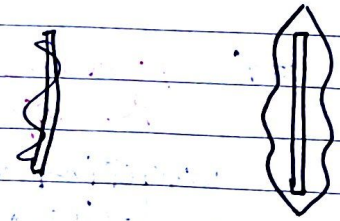
penalized large weights. (b'coz Σ is used) & learns simpler patterns

L1 Reg. } other methods.
Dropout }

L1 Regularization

$= \lambda \sum |w_i|$

→ promotes sparsity
→ less important features are ignored
→ leads to simpler & more interpretable model.
→ improved generalization performance.



Default Batch Size in Keras = 32

Vanishing Gradient Problem. : gradient of sigmoid becomes nearly zero.

46:20
ML Tech Talk
Deep Learning

Tensorf

Autoencoders → image reconstruction

Tensorflow.

keras → High level API (used for experimentation by ML practitioner & research engineer)

modules. —— eg: nn . : nn layers, operations & utilities
sets
signal
sparse
lite.
image — image processing & manipulation operation

- Classes.
  - Functions
  - Data

in keras. module.

conv2D
relu
Conv3D ...

API. High-level.

modules.
- ~~Classes~~
- losses
- backend
- activations
- preprocessing

These are keras modules.

Losses module of keras.

Classes :- Binary crossentropy
CTC
Hug.

Functions : MSE
get
Loss.

Import Libraries

→ import tensorflow as tf
~~print~~

→ from tensorflow.keras.layers import Dense, Flatten, Conv2D
from tensorflow.keras import Model

module module

tf.keras.layers ← Dense — class
Flatten — class
Conv2D — class
Conv3D — class

tf.keras
core data structures
Layer
Models → directed Acyclic Graph (DAG)

framework
tf → API Keras.
Layers
models
data structures

input/output transformation

layers.Layer
class
recursively composable
Layer1 output → input to Layer2
weight
call() & computation
trainable or non-trainable

Layer(layer) → output of this = input to this

Layers
→ handle data prepro
- normalization
- Text vectorization

deep learning model is directed A.C.G. of layers

models
DAG
object
groups layers together

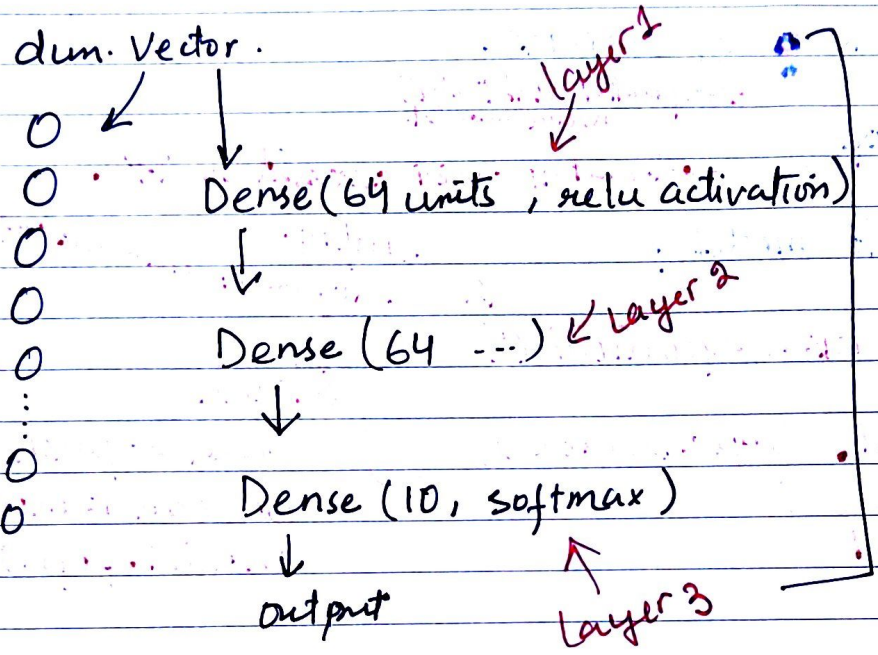
e.g. simplest model:
Sequential model.
↓
Linear Stack of layers

Complex model:
- functional API
(non-linear topology
- Shared layer
- multiple inputs or outputs

Preprocessing
↓
include directly into model.
during training or ~~testing~~ after training

DAG ex.

784 dim. Vector.



Sequential API

↳ used when layers have 1 i/p tensor & 1 o/p tensor.

ex. 3 layer model

```
model = keras.Sequential(
    [   layers.Dense(2, acti, name= ),
        Layers.Dense(3, ___ ),
        Layers.Dense(4, ___ ),
    ]
)

x = tf.ones((3,3))
y = model(x)
```

```
layer1 = layer.Dense( )
layer2 = layer.Dense( )
Layer3 = layer.Dense(4, name=layer3)

x = tf.ones((3,3))
y = model(keras Layer3(lay2(lay1(x))))
```

```
model = keras.Sequencial(
        [ Layer.Dense (4, activ-fxn, name=layer2),   ← # of neurons
          L       - - -
          Lay - - -
        ]
)     }
```

How to check # of layers.

model.layers   ← sequential object with name model. / element of object accessed by 'dot'.

adding Layers to model ?

model = keras.Sequential( )   ← empty. argument: name='my-seq'

model.add(Layer.Dense( ))
model.add( )
model.add

Stack
add = push
we can remove layers.
.model.pop()

weights can be accessed from layers.

Layer = layers.Dense(4)
~~Layer~~ layers. weights

object of Dense class from layers module.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

~~Z~~ 14

XW + b

input, weight, bias

Three neurons in dense layer.

(1,4) (4X3) + (3,)

[a b c d] [a - - -; b - - -; c - - -; d - - -] + [-; -; -]

Dense(3) → (4,3)

(input (1,4))

Bais