Biological Inspirations : Visual Cortex (4.1) :-

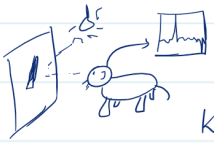
aka CNN / ConvNets.

→ Mainly used for visual tasks such as object recognitions etc,

→ Breakthroughs in (DL + CV) happened in the last 10 years.
↳ Computer Visions

→ Hubel & Wiesel experiments in 1950-80s :-

on animals. certain neurons had more activity when light edge is of a shape.

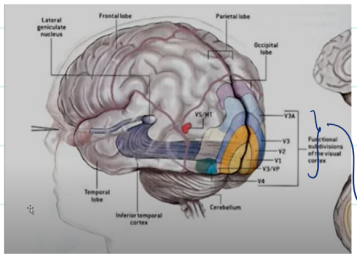


ex :- | → spike in neural activity / → medium spike — → no activity

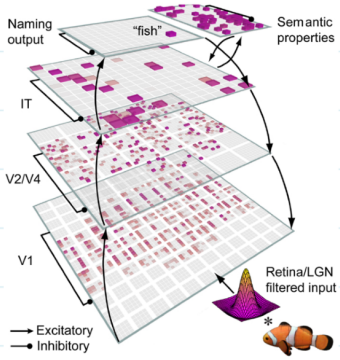
Key findings :- ① Some neurons in the visual cortex are fired when presented with light at a specific angle. ex :- V_1 : primary visual cortex → detects edges.

② There are neurons for edge detections, motions, depth, color, faces, shapes etc,

③ There is a hierarchical structure among visual cortices.



↳ visual cortices.



→ Lots of image processing & CV is inspired by Hubel & Wiesel research.

→ ConvNets take advantage of this research & biological structure.

Convolutions : Edge detection on images (4-2) :-

→ performed by primary visual cortex (V_1)

ex :-  → image

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1020 & 1020 & 1020 & 1020 \\ -1020 & -1020 & -1020 & -1020 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Sobel horizontal edge detector

not multiplication convolution.

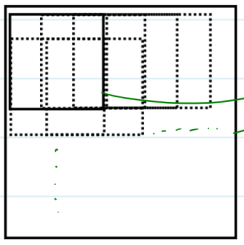
kernel / Filter / Mask operation

min max norm →

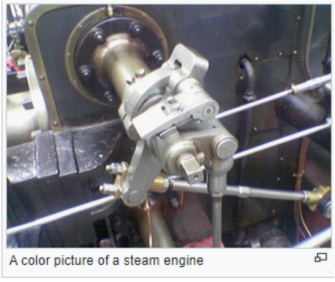
$$\begin{bmatrix} 255 & 255 & 255 & 255 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 255 & 255 & 255 & 255 \end{bmatrix}$$

2 black lines & 2 white lines

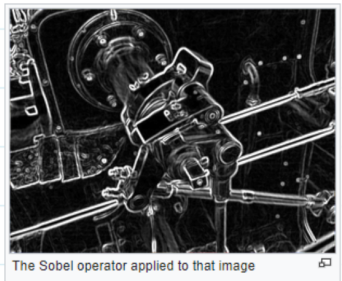
→ Sobel vertical edge detector : $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$



→ kernel
→ input
element wise product and their sum = $\mathbb{R}^1$



→



$$G_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Add matrices obtained from both for both horizontal & vertical

→ Kernel size is not fixed. Not always 3×3

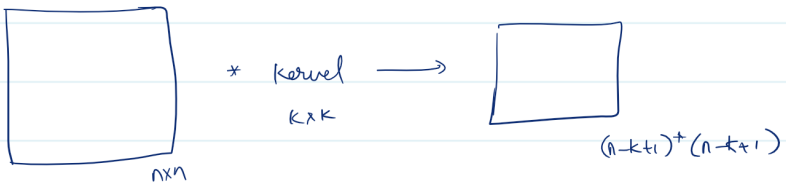
→ Kernel values can also change.

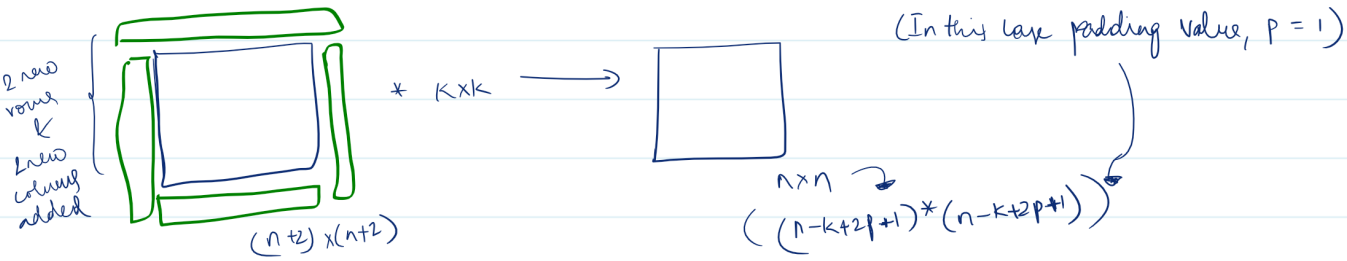
→ Box filter = $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ → blurs the image when sum is divided by number of coeffs.

$\begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}$ → blurs the image distinctly.

Convolutions : Padding & Strides (4.3) :-

→

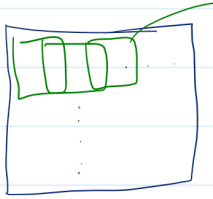




→ Padding can be filled by (i) Zeros : Most common. Adds more edges to the image through. These edges are not learned again, in backpropagating.

(ii) Same Value Padding : Not common. Causes conflicts in case of multiple neighbours

Stride :-



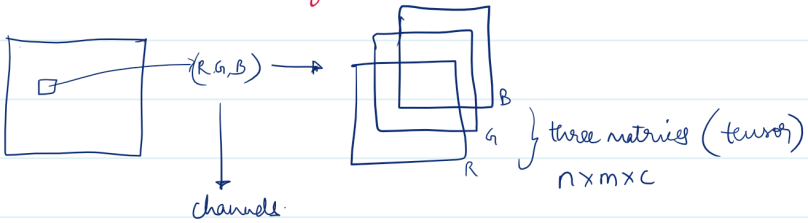
Moving 1 by 1 = stride of S = 1

$n \times n \xrightarrow[K \times K]{\text{stride 's'}} \left(\left\lfloor \frac{n-k}{s} \right\rfloor + 1\right) * \left(\left\lfloor \frac{n-k}{s} \right\rfloor + 1\right)$

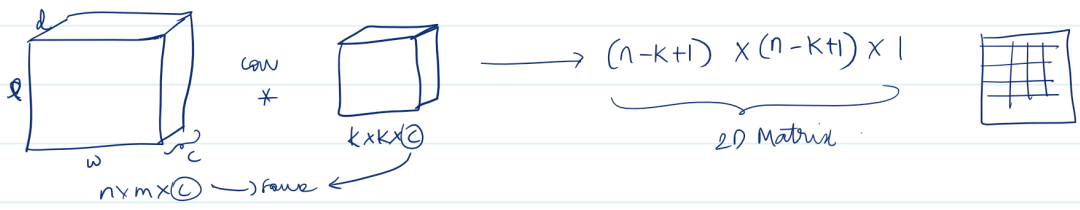
→ $n \times n \xrightarrow[\text{Padding } p]{K \times K \text{ kernel}} \left(\left\lfloor \frac{n-k+2p}{s} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{n-k+2p}{s} \right\rfloor + 1\right)$

→ Padding is bigger output, stride for smaller output.

Convolution over RGB Images (4.4) :-

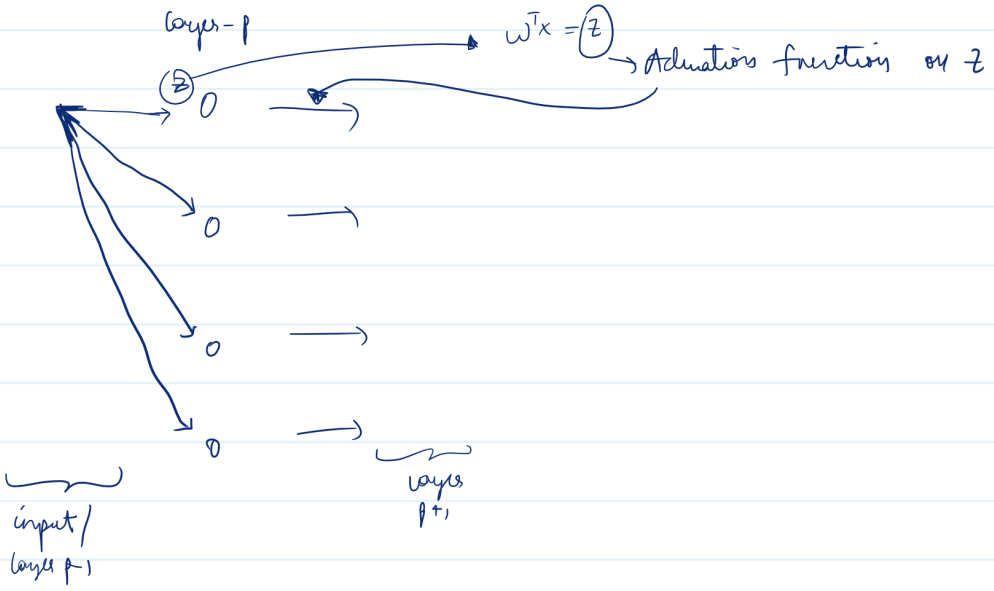


→ Convolution performed same as in 2D ie, component wise multiplication and addition



Convolution layer :- (4.5)

→ In normal MLP,



→ But in CNNs,

multiple edge detectors ⇒ multiple kernels

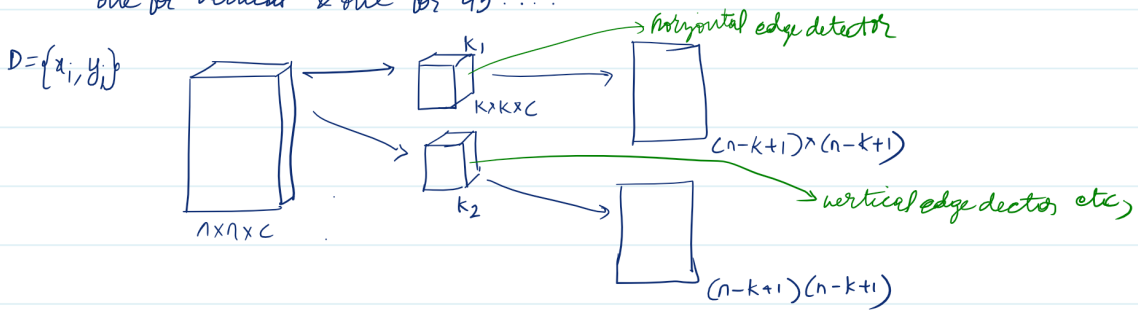
(1 edge detector = 1 kernel)

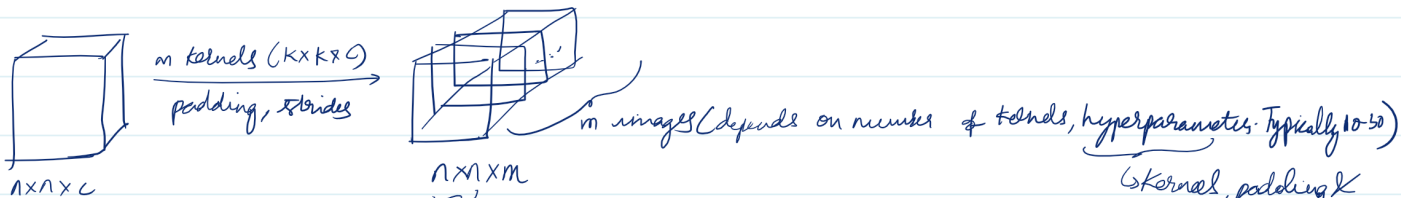
So in CNN, we learn the kernel matrices in backpropagation

MLP : learn weights

CNN : learn kernel matrices. ($I_{k \times k \times c}$)

for ex:- instead of hardcoding one kernel to identify edges, we could have one kernel for horizontal edge, one for vertical & one for 45° ...



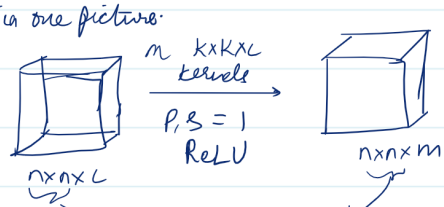


① Input (n×n×c) * Kernel (k×k×c) → p, s = 1 → n×n×1

equivalent to $w^T x$

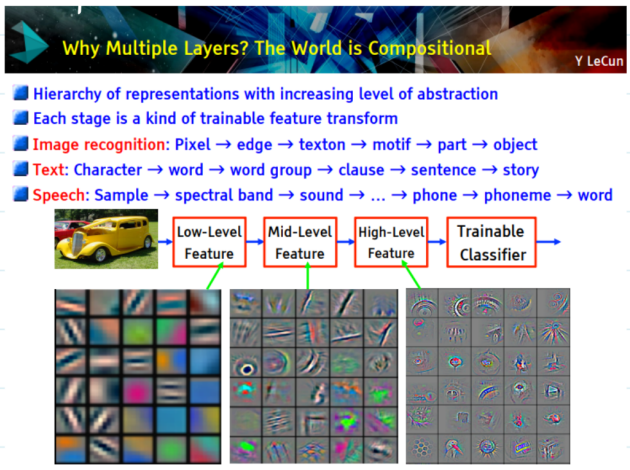
element wise ReLU

→ * in one picture.



need not be the same always.

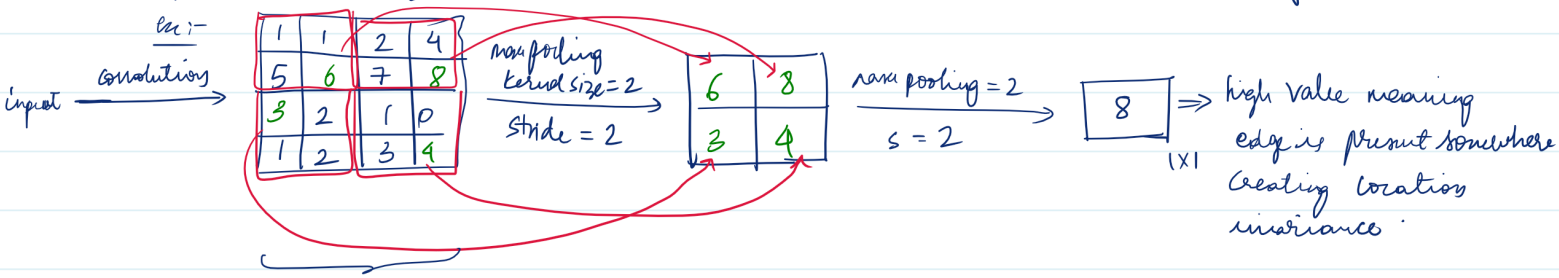
→ In a deep multi layer CNN, works like human visual cortex.
first layer identifies pixels, second edge, third texture, ..., part, object etc)



→ Using ReLU is not mandatory in CNNs.
→ Kernels are initialized in the same way as in MLP.

Max Pooling (4.6) :-

→ Location invariance :- We want model to detect object irrespective of it's location in image. Similarly scale invariance, rotation invariance etc)
In human body at higher levels of visual cortex (V7 etc) neurons are fired as soon as object is recognised irrespective of scale, size etc) Max pooling tries to replicate this
→ Max pooling can also be thought of as a kernel. Finds the max value in the kernel shape & size.

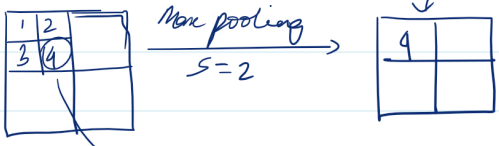


↳ Since this is an output of convolution matrix, the value will be high if there is an edge present. Otherwise it will be a small value.

→ Very popular in ConvNets. Avg-Mean pooling is also gaining popularity.
→ Capsule Network captures both presence of an object & it's locale data.
If eyes, ears, mouth & nose positions are jumbled, CNN will recognise a face, but CapsNet won't.
→ Repeated kernels automatically get rid of scale invariance. Data augmentation takes care of rotation invariance.

CNN Training : Optimization (4.7) :-

→ In order to calculate derivatives, operations need to be differentiable.
CNN operations :- Convolution Layer :- Convolution + (ReLU) → differentiable
Max Pooling
↳ dot product + addition intuitively. In MLP, it's element wise operations. In convnets, it's pixel wise operations.



derivative of 4 is only passed to 4 in our matrix.

ie, $\max(x) = \frac{\partial g}{\partial x_i} = \begin{cases} 1 & \text{if } x_i = \max(x) \\ 0 & \text{otherwise} \end{cases}$

→ derivative is passed only to this value. This is done because no matter what happens to other val, it doesn't affect max pool Thereby the end result.

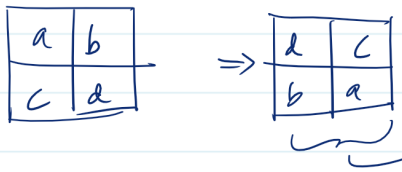
→ Same thing happens to mean pool / avg pool.

→ We can use dropouts on CNN as well just like in MLPs.

Read first comment links for more content.

→ if we invert kernel matrix in correlation by 180°, it becomes convolution operation

ie,



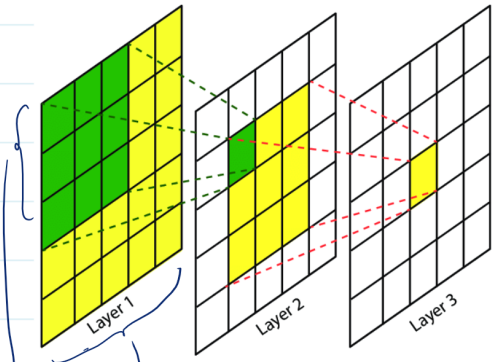
→ convolution.

→ Max pooling selects the brightest pixels from the image. Useful when image is dark & we are interested in only lighter pixels of image. Mean pooling smoothens the image & sharp features may not be extracted.

Receptive Fields and Effective Receptive Fields (4.8):-

Receptive Field :- The region on image on which kernel operates on at a given time 'T.'

Effective Receptive Field :- In deep CNN, the region on original image on which kernel at a deeper layer indirectly works on.

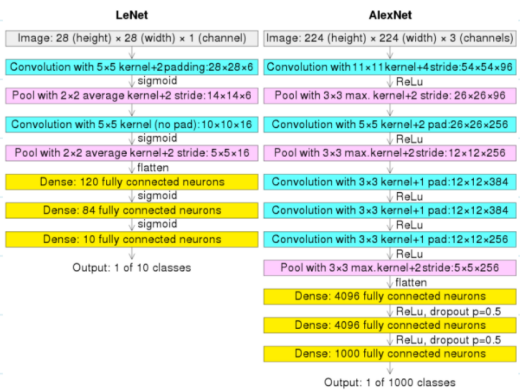


→ Receptive field for layer 2.

→ effective receptive field for layer 3.

ImageNet (4.10):-

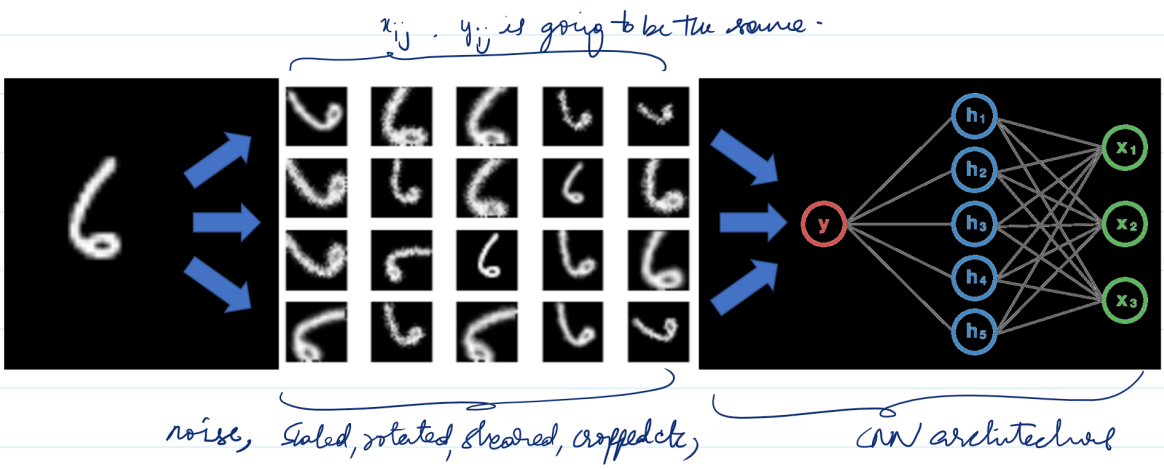
→ CNN used AlexNet in 2012 gave very good result.



→ AlexNet vs LeNet.

Data Augmentations (4.11):-

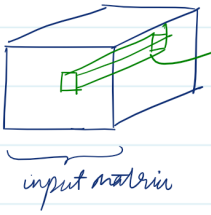
→ We want the models to be robust & work even when image is rotated, scaled etc.,



→ Why? We can get invariances like scale invariance, rotation invariance etc., Also can increase the size of the dataset.

→ Augmentations prevents model from overfitting by introducing divergence to the dataset.

AlexNet (4.13):-



→ only this data normalized = Local response normalization. Not used these days. Replaced by BatchNormalization.

→ AlexNet used dropouts, ReLU & multiple GPUs to train. Gave good score.

→ Objective of dense layer is to take results from convolution/pooling and use them to classify image into a label.

VGG Net (4.14):-

→ By Andrew Zisserman (key figure in CV). AlexNet uses all sorts of kernel sizes. VGG tries to use simple fixed kernel sizes ie, 3x3 kernels, $S=1$, $P=$'same' and maxpool $S=2$, 2×2.

→ VGG = Visual Geometry Group.

→ 2 types VGG-16 & VGG-19. Default networks used these days.
16 layered & 19 layered network.

→ Pretrained available in Keras.

→ In training, weight decay ie regularization = 5×10^{-4} & dropout ratio = 0.5.

→ Regularization can be applied at all layers including conv layers because they contain parameters too.

→ In AlexNet MaxPooling done after almost every conv layer. In VGG, it's done after 2-3 conv layers.

ResNet (4.15):-

→ Residual Networks. By Microsoft

← Even after adding lots of layers, error not reducing even after dropout, lots of iters etc.)

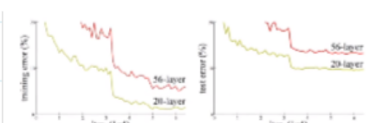
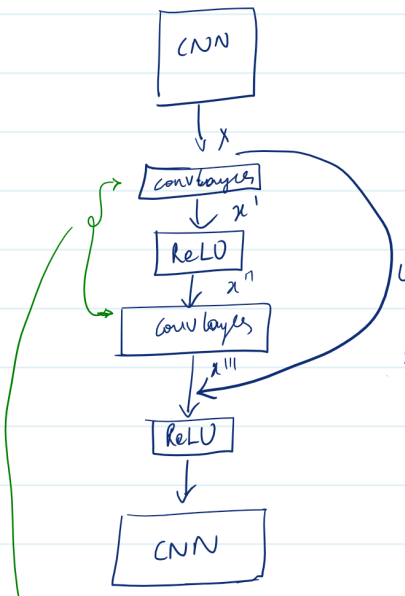


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.



→ Doesn't necessarily have to be conv layers. Can be dot prod (MLP) too.

- if x is +ve, $ReLU(x) = x$
  $ReLU(ReLU(x)) = x = ReLU(x)$
- if x is -ve, $ReLU(x) = 0$
  $ReLU(ReLU(x)) = 0 = ReLU(x)$

Identity / residual block. Skip connections.

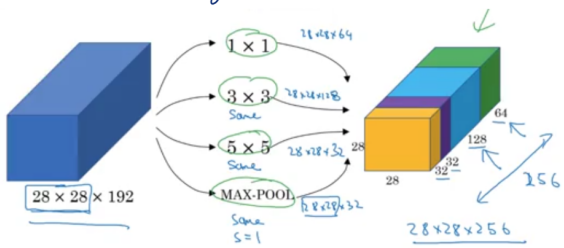
x is added to x'''

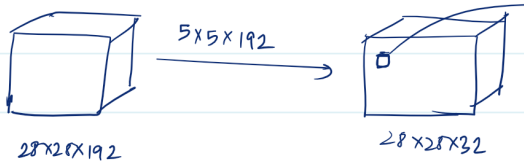
→ if because of some regularization, these 2 become 0 (weights), then $a''' = x$ instead of 0. ie, output will not be impacted & adding additional layers would not hurt performance - will skip over them.

→ Must be before ReLU & x & x''' must be same size ie, padding should be taken care of in conv layers.

Inception Network (4-16) :-

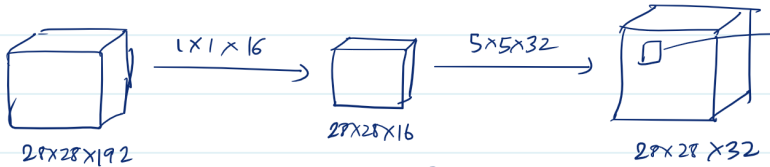
→ Instead of using just one kernel, use multiple kernels at once.



→ 

→ In this all $(28 \times 28 \times 32) \times (5 \times 5 \times 192)$ computations = 120 Millions

But instead

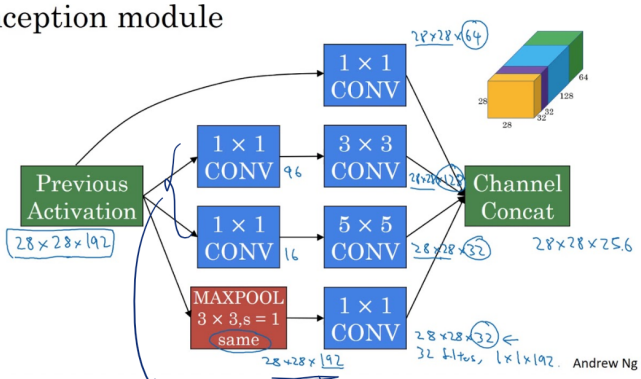


$((28 \times 28 \times 16) \times (1 \times 1 \times 192)) + ((28 \times 28 \times 32) \times (5 \times 5 \times 16))$

= only 12.4 millions

→ Bottleneck layer.

Inception module



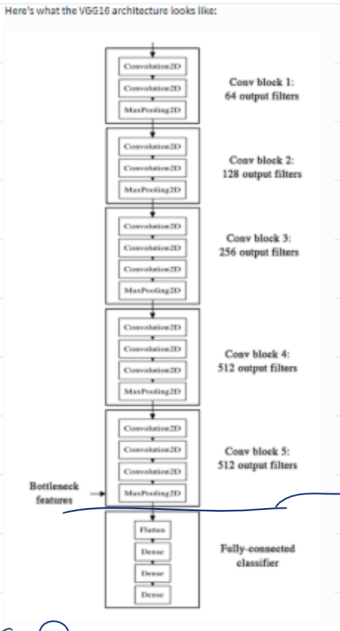
Bottleneck layers.

This would dominate the conv kernels above it. So we use 1×1 to reduce 192 to 32.

Transfer Learning (4:17):-

→ Instead of building NN classifies from scratch, we can reuse existing models trained on diff dataset (like VGG16). We "transfer" learnings from one model & use it for our use case.

Case ①:-

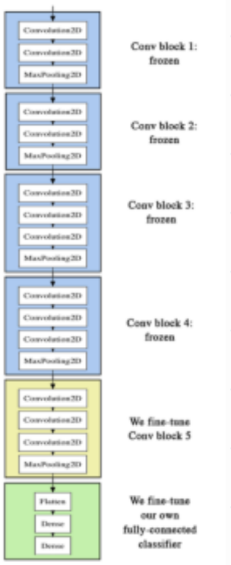


We only use these layers

→ Bottleneck layer. By this layer we have a vector representation of each image.

} X ie, we use VGG as a sort of feature engineering tool. Till flatten if classification problem or till bottleneck if segmentation problem (goal is to divide image into segments).

Case ②:-



We "freeze" these layers as they are anyway needed.

We modify these layers acc to our needs using our dataset D. We should ensure learning rate is small so things dont change drastically.

Case ③:- VGG16 + Imagenet to initialize new model & train model entirely using our dataset D. Learning rate small

Case ④:- retrain model from scratch. (Bad & expensive)

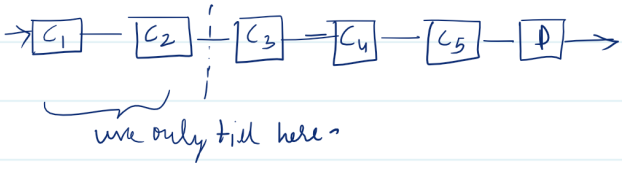
→ which case to choose depends on size of our dataset

Case ①:- When D ≈ Imagenet & |D| is small

Case ②:- |D| ≈ Imagenet & |D| is large. finetune the NN

Case ③:- |D| = reduced D ≈ Imagenet finetune last layers.

Case ④:- |D| = small & D ≠ Imagenet



Case ⑤:- |D| = large & D ≠ Imagenet, init model using VGG16 & retrain.

fine tune = starting from a trained network, retraining on new dataset using very small update changes.

General trend in designing NN architecture

CNN → pool → CNN → pool → Dense → Dense (sigmoid/softmax)

Can replace with any number of dense layers as long as it makes sense.

Can also be refined as CNN – BN – Activation – Dropout – pool → . . . . → Dense → Output.

Face Recognition from scratch in CNN:-

→ 1000 images per class is good for CNN (if doing without transfer learning)

→ last couple of layers with 50-100 images per person. Data Augmentation as well.
(Transfer learning) ↳ of VGG. ↳ Realistic Setting (glasses, lighting etc.) and various angles.

→ Good loss would be cross entropy & softmax last layer.

→ If no time, use cloud APIs that are pre trained. (ex- MS Advanced Facial Recognition System)