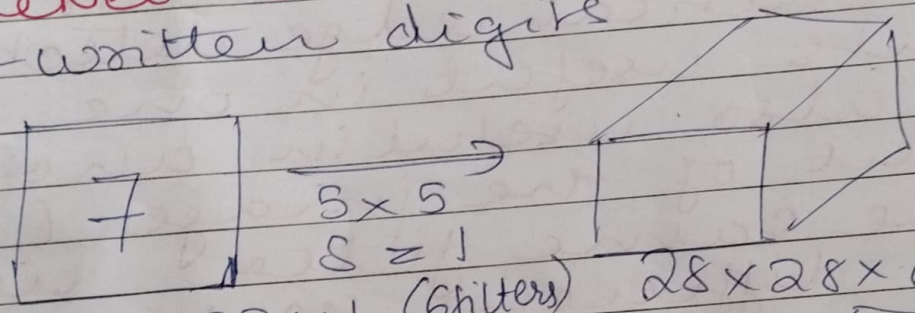
Case Studies

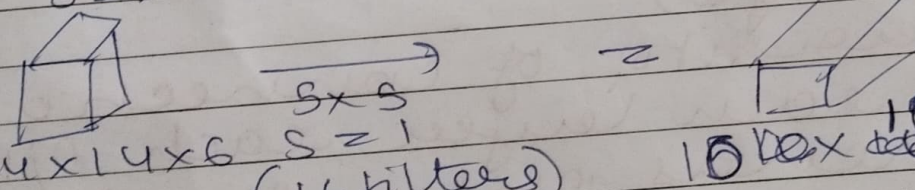
→ Classic Networks:
→ LeNet-5 → AlexNet → VGG
→ ResNet: (152 layer network : very deep)
→ Inception

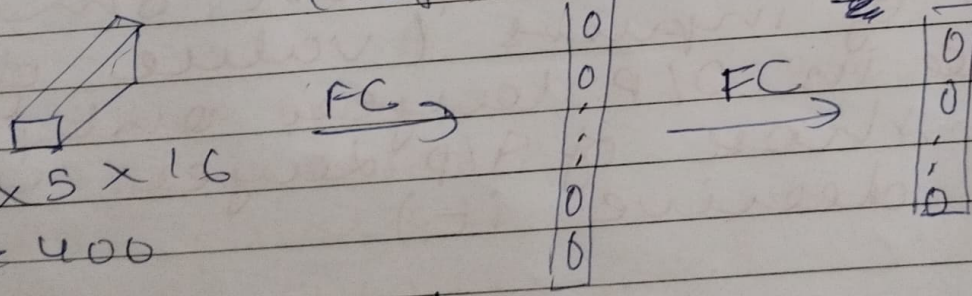
Classic Networks

1) LeNet-5 ⇒ Recognized hand written digits



(back then padding was not used)





→ used sigmoid
→ In a modern version of this layer we use softmax layer with 10 way classification O/P
→ Earlier it used a diff classifier O/P layer that's useless today
→ about 60K parameters (quite small) → $n_H, n_W \downarrow \rightarrow n_C \uparrow$

then pool ... fully con. → fully con → output

PAGE No.
DATE / /

Softmax function – converts a vector of K real nos into probability distribution of K possible outcomes.

→ It is a generalization of logistic func (sigmoid) to multiple dimensions used in **multinomial logistic regression**

→ Often used as last activation function of a neural network to normalize O/P to a probability distribution over predicted O/P classes

→ Takes an input z of K real nos & normalizes it into a probability distribution of K possibilities prop. to the exponentials of input nos

→ Each component after applying softmax is in interval (0, 1)

→ All components add up to 1 so they are interpreted as probabilities

$\sigma : R^K \to (0,1)^K$ where $K \geq 1$

defined by formula

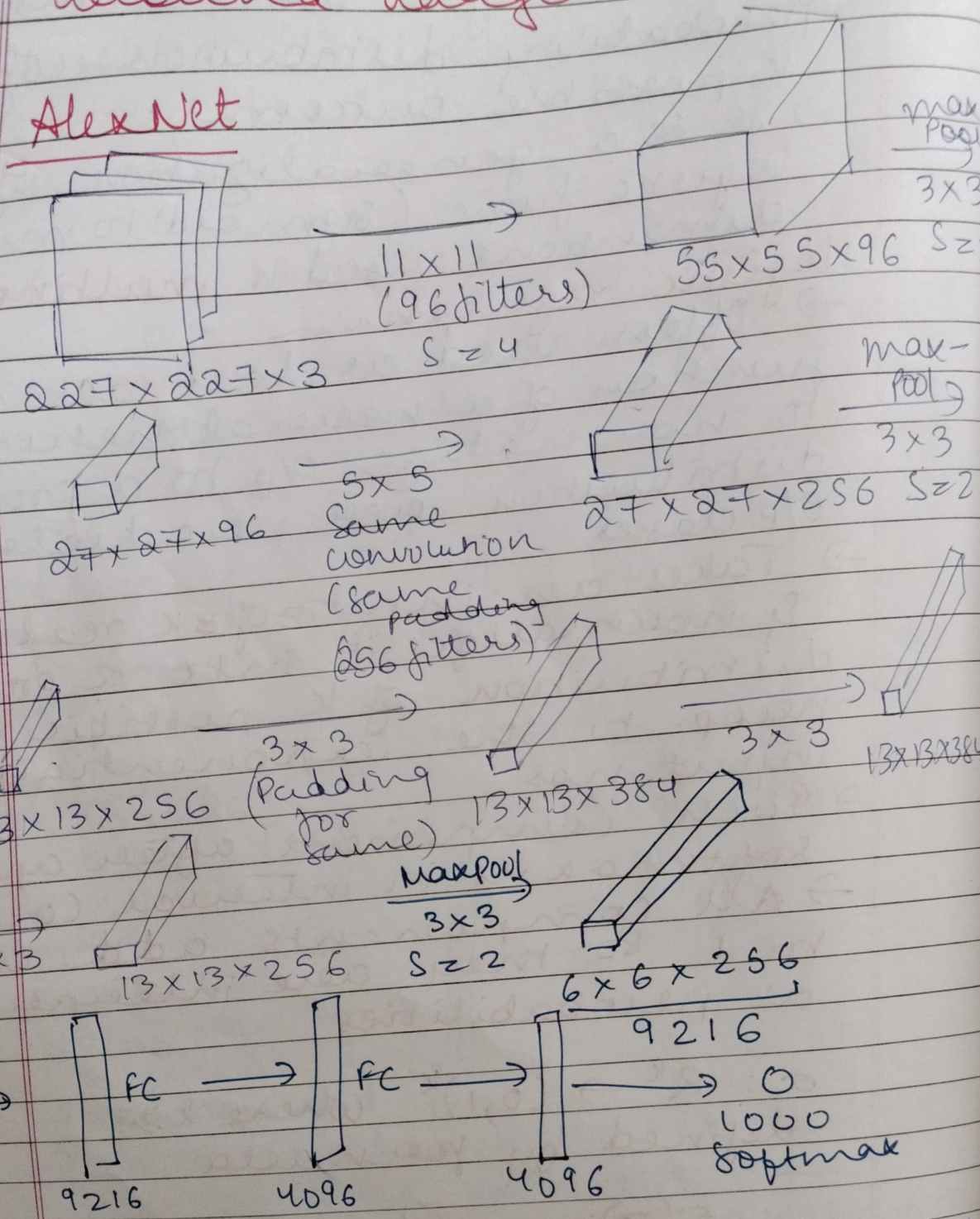
$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots K$$

$z_K = (z_1 \cdots z_K) \in R^K$

→ Instead of e diff base $0 < b < 1$ can be used which concentrate larger O/P probabilities around smaller val

→ b>0 - Larger O/P Probabilities around larger values

AlexNet

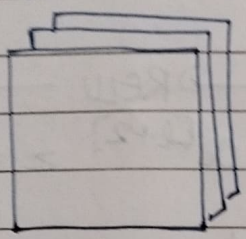


→ similar to LeNet but much bigger
→ Used RELU activation
→ Multiple GPU
→ Local response normalization layer

13 13 256 look at 1 position height & width across all channels and normalize them
↳ Not really useful

VGG-16

CONV = 3×3 filters S = 1
always use padding
MAX-POOL = 2×2, S = 2



224×224×3 → (conv 64) ×2 → 224×224×64

2 conv layers (→ →)
224×224 ×64 224×224 ×64

POOL → 112×112×64 → [conv 128] ×2 → 112×112×128

POOL → 56×56×128 → (conv 256×3) → 56×56×256

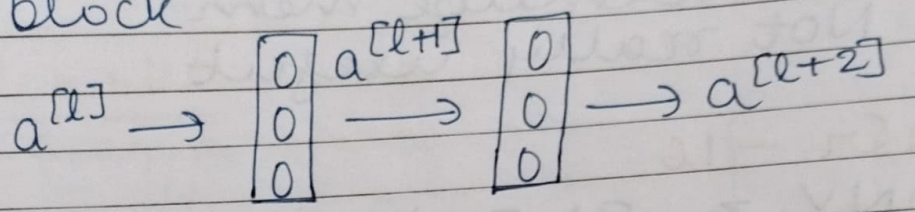
POOL → 28×28×256 → (conv 512×3) → 28×28 ×512

POOL → 14×14×512 → (conv 512×3) → 14×14×512

POOL → 7×7×512 → FC 4096 → FC 4096 → ~~Softmax~~ 1000

Residual Networks (ResNets)

→ Built from a residual block



$a^{[l]} \longrightarrow$ Linear $\longrightarrow$ Relu

$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}$ $\quad a^{[l+1]} = g(z^{[l+1]})$

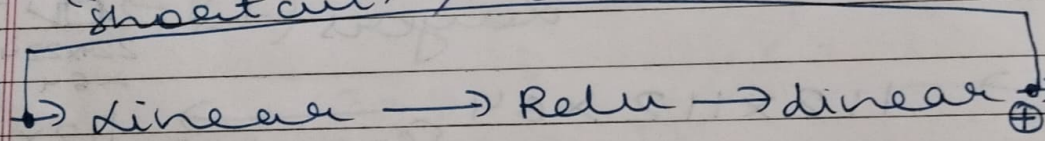
$\longrightarrow$ Linear $\longrightarrow$ RELU $\longrightarrow a^{[l+2]}$

$z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]}$ $\quad a^{[l+2]} = g(z^{[l+2]})$

$a^{[l]}$ to $a^{[l+1]}$ needs 4 steps

(ResNets)

"shortcut" / "skip connection" (2-Lin, 2-RELU)



$\therefore a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

→ Rather than needing to follow the long path the info from $a^{[l]}$ can follow shortcut to go much deeper into neural networks

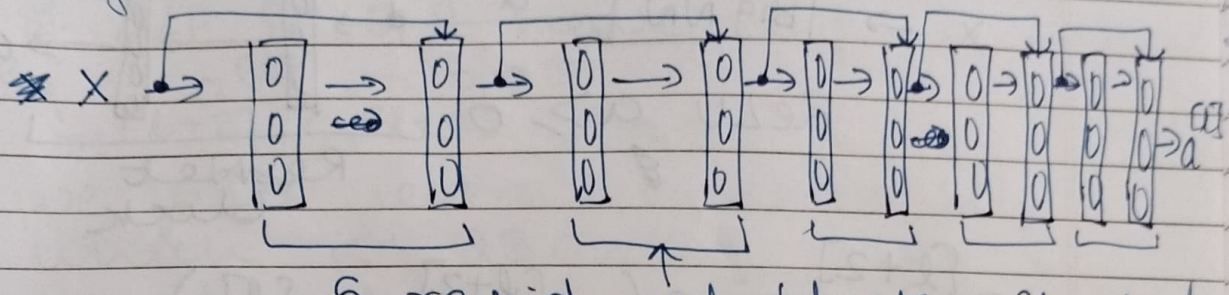
ie instead of $a^{[l+2]} = g(z^{[l+2]})$ we have $g(z^{[l+2]} + a^{[l]})$

→ addition of $a^{[l]}$ makes it a residual block

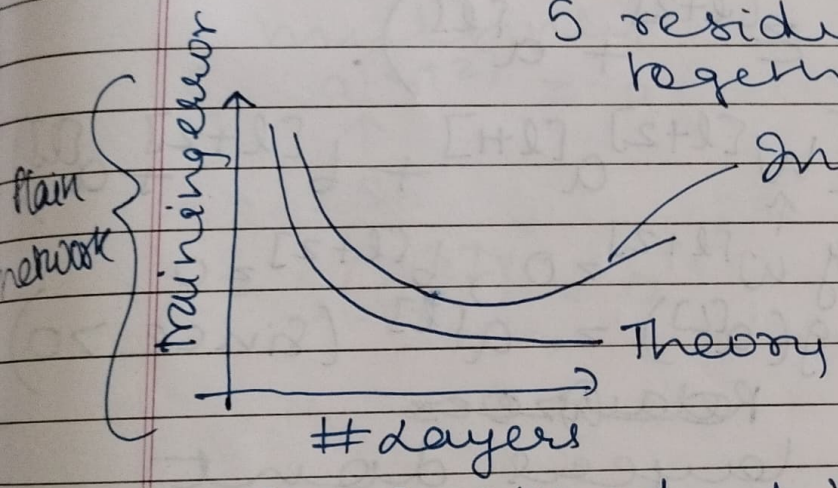
→ Using residual block you can train much deeper neural networks

⇒ The way to build ResNet is by taking many of these blocks like these and stacking them together to form a deep network

→

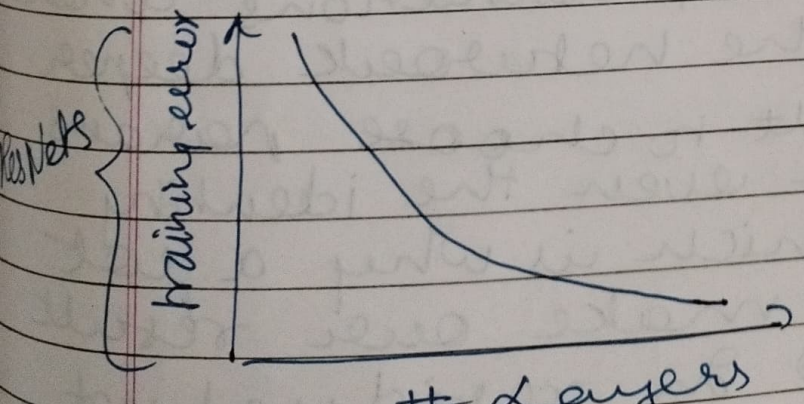


5 residual blocks stacked together

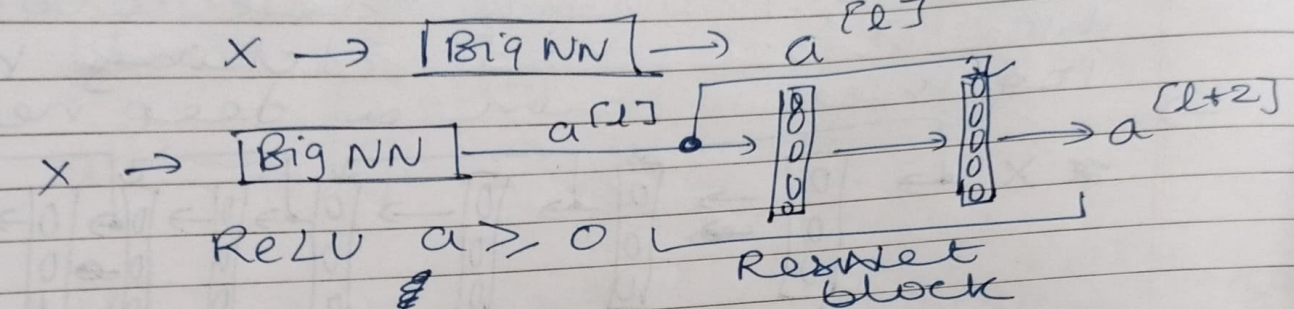


In reality training error gets worse if u pick a network that's too deep

⇒ In ResNet what happens is that even as no of layers get deper u can have performance of training error keep going down



Why Do residual Networks work



$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

$$= g(W^{[l+2]} a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

if $W^{[l+2]} = 0 \rightarrow b^{[l+2]} = 0$

$$= g(a^{[l]}) = a^{[l]} \text{ (since } > 0)$$

∴ adding layers dont cause error increase

→ What goes wrong in deep learning plain nets is without residual of skip connections when we make the network deeper its difficult to choose params that learn even the identity function which is why a lot of layers make our result worse → In residual net. its easy for extra layers to learn identity func so it doesnt hurt performance

→ The dimensions of $z^{[l+2]}$ & $a^{[l]}$ must be same so we see lot of same order convolutions

→ If I/P & O/P have diff dimensions

Eg $a^{[l]} = 128$

$z^{[l+2]} / a^{[l+2]} = 256$

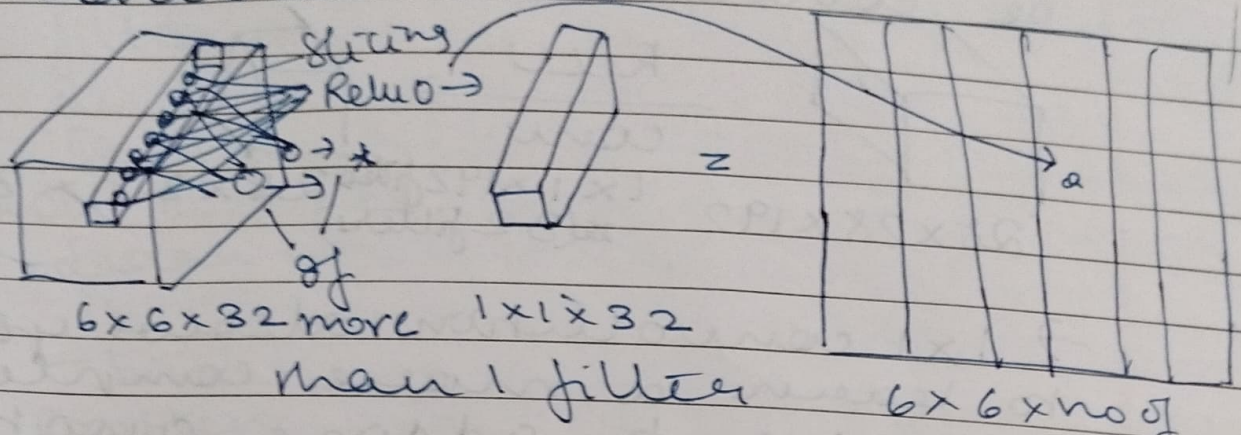
we add extra matrix W_s which is 256×128 & multiply to $a^{[l]}$

$$a^{[l+2]} = g(z^{[l+2]} + W_s \times a^{[l]})$$

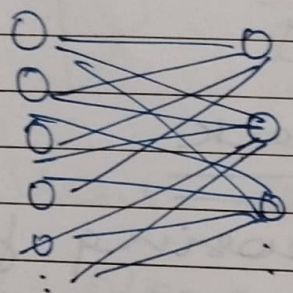
$a^{[l+2]}$: 256 ; $z^{[l+2]}$: 256 ; W_s: 256×128 ; $a^{[l]}$: 128

$$= g(a^{[l]}) = a^{[l]}$$

Network in network 1×1 convolutions



⇒ It is having a fully connected neural network for each input slice ie 36 networks



Inputs 32 units & O/P no of filters (fully connected NN)

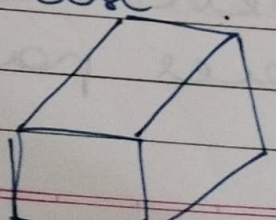
32 inputs of slice

no of filters $n_c^{[l+1]}$

This occurs 6×6 times to give total of $6 \times 6 \times n_c^{[l+1]}$ outputs

⇒ This is called a network in network

Eg of use:

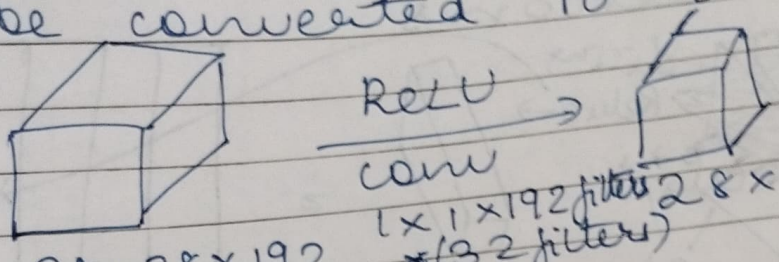


28×28×192

to shrink it we can use Pooling but if no of channels

Same convolution
— O/P size = Input size by padding

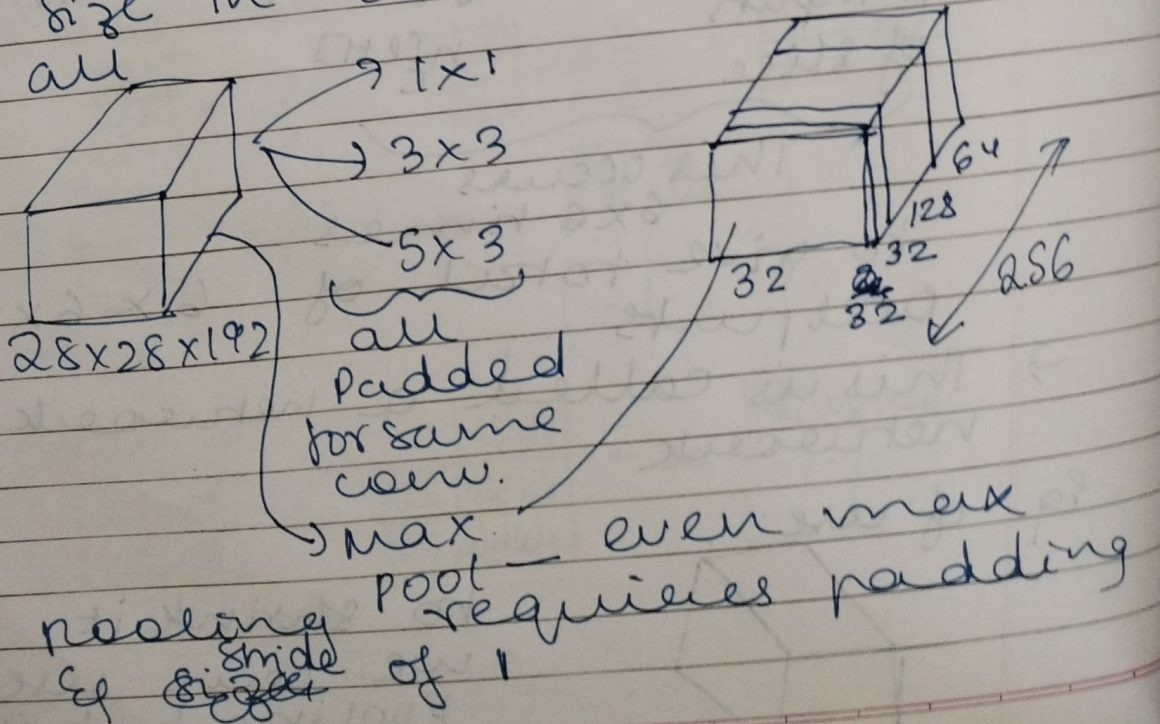
is too big ie it has to be converted to 28 × 28 × 32



→ 1×1 convolution allows you to learn a more complex function by adding another layer (even if input & output dim. are same). The effect of 1×1 conv is it just has a nonlinearity

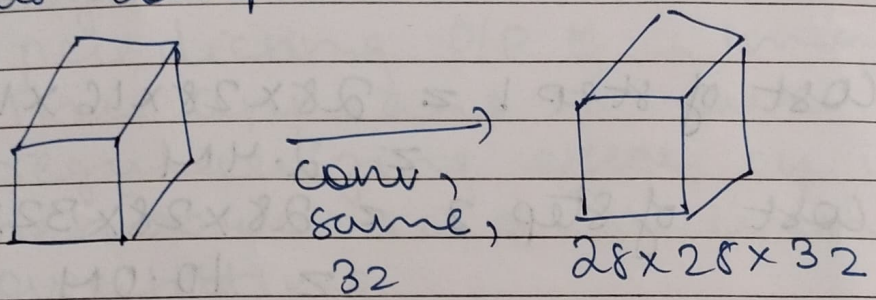
Inception network

→ Instead of choosing filter size in conv layer, do them all



→ basic idea is instead of you needing to pick one of these filter sizes or pooling you want and committing to that you can do them all, concatenate all outputs and let the network learn whatever parameters it wants to use, whatever combinations of filter sizes it wants

→ problem with inception layer is computational cost



filters are 5×5×192

∴ O/P size is 28×28×32

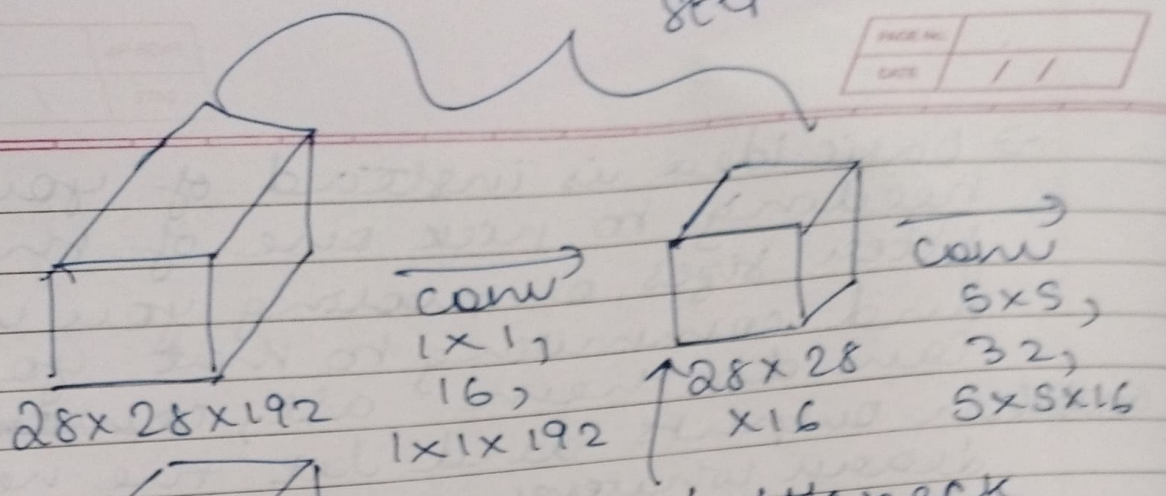
So we need to compute 28×28×32 nos, and for each we do 5×5×192 multiplic

∴ total multiplic = 28×28×32×5×5×192

≈ 120 mill. (This is expensive)

Total multiplies = no of multiplies for each O/P × no of O/P

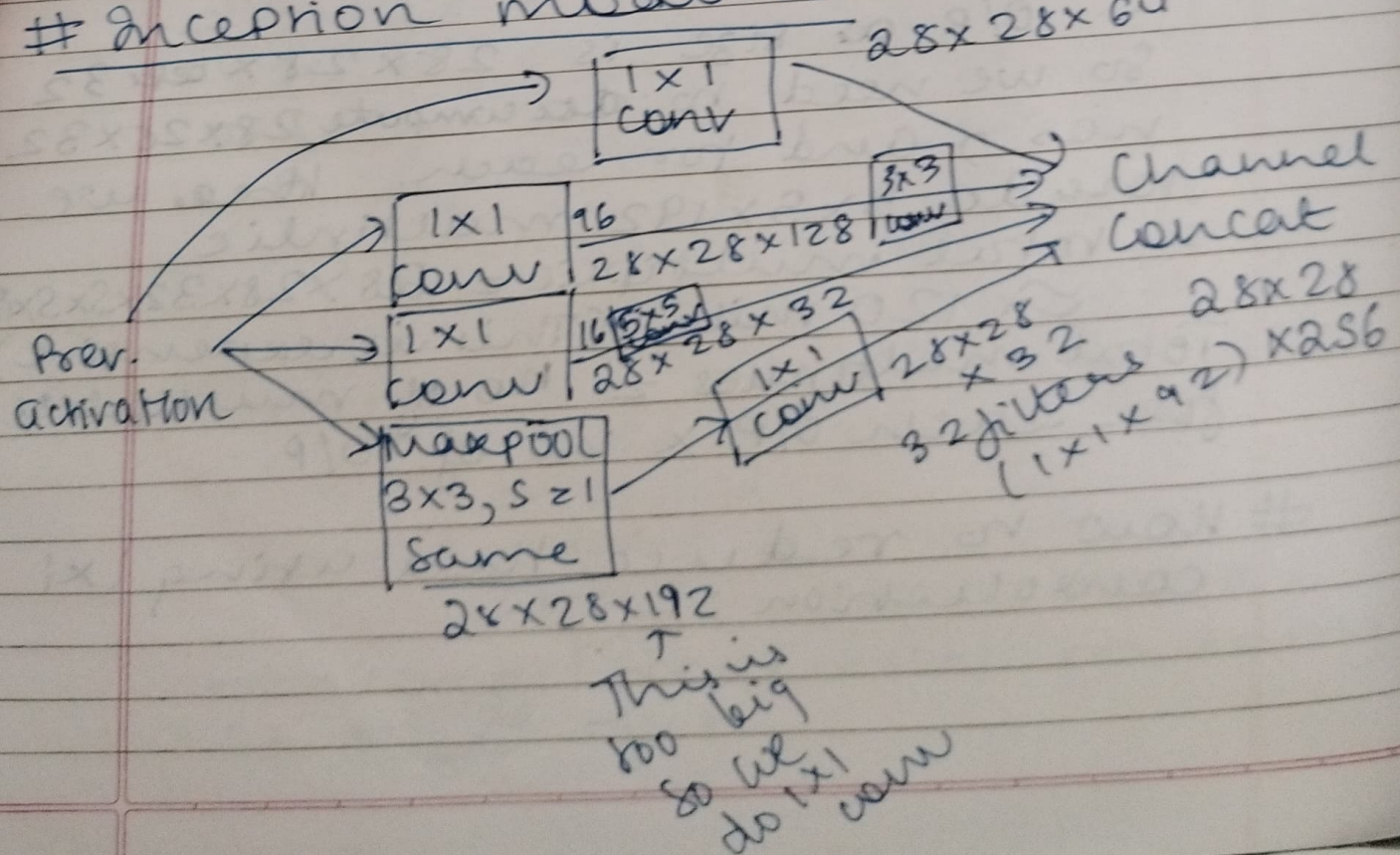
How to reduce this using 1×1 convolution :-



cost of step 1 = $28 \times 28 \times 16 \times 1 \times 1 \times 192$
= 2.4M

cost of step 2 = $28 \times 28 \times 32 \times 5 \times 5 \times 16$
= 10.0M

Inception module

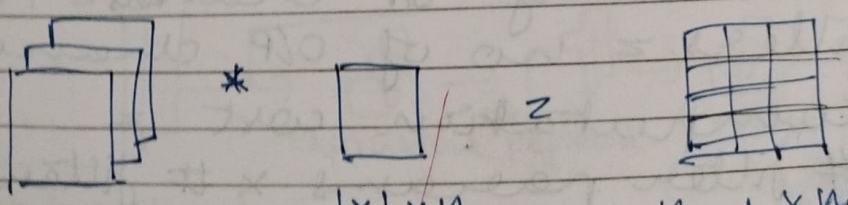


→ Inception module network combines several such modules together

→ There are sidebranches that try to evaluate results using softmax
→ This ensures that features computed even in the hidden units even at intermediate layers are not too bad for predicting O/P of a image. This appears to have a regularizing effect on inception network and prevents overfitting
→ Dev. by google (GoogleNet)

MOBILENET

→ Lot of N.N are quite computation -ally expensive can't run on phones / embedded vision
→ Key idea : Normal vs depthwise seperable conv.
→ Normal conv



⇒ Inception module network combines several such modules together

⇒ There are sidebranches that try to evaluate the results using softmax

⇒ This ensures that features computed even in the hidden units even at intermediate layers are not too bad for predicting O/P of a image. This appears to have a regularizing effect on inception network and prevents overfitting

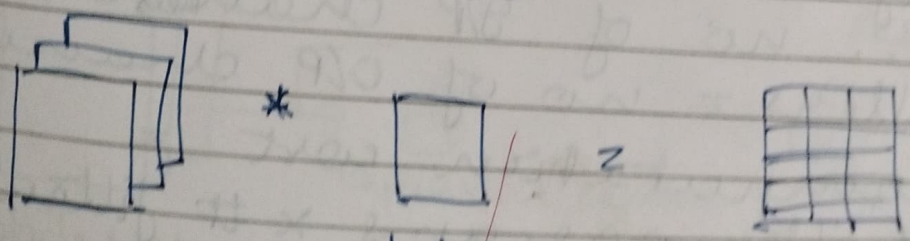
⇒ Dev. by google (GoogleNet)

MOBILENET

⇒ Lot of N.N are quite computationally expensive can run on phones / embedded vision

⇒ Key idea : Normal vs depthwise Seperable conv.

⇒ Normal conv



$n \times n \times n_c$ $\quad f \times f \times n_c$ $\quad n_{out} \times n_{out} \times n'_c$

(n'_c filters)

computation cost = filter params × filter positions × no of filters

$f \times f \times n_c$

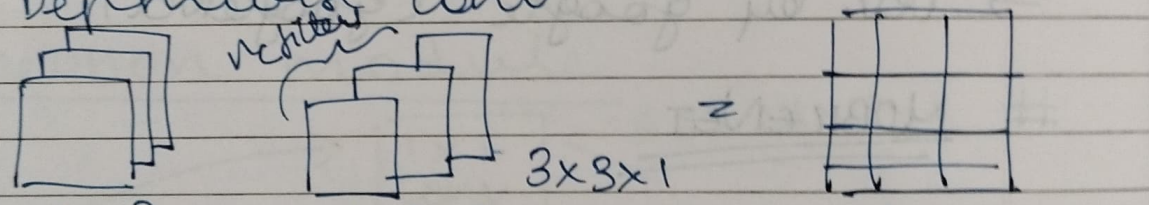
↑ how many times filter is placed on input img ie $n_{out} \times n_{out}$

$= f \times f \times n_c \times n_{out} \times n_{out} \times n_c$

Depthwise Seperable Convolution

→ Depthwise conv. followed by a pointwise conv.

1) Depthwise conv



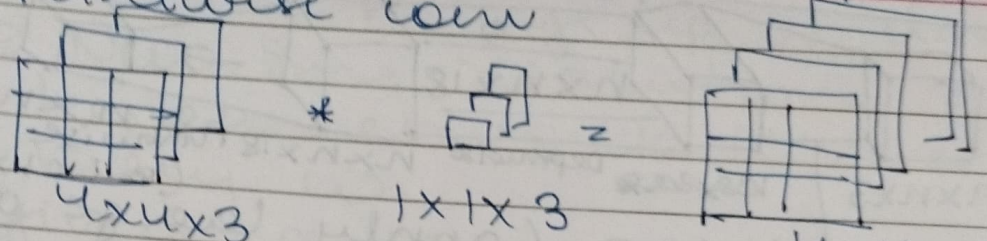
→ 1st filter placed on 1st channel of I/P to give 1st channel of O/P and so on

∴ no of I/P channels = no of filters = no of O/P channels

∴ computation cost = # filter params × # filter positions × # filter no

$= 3 \times 3 \times 4 \times 4 \times 3 = 432$

2) Pointwise conv



$n_{out} \times n_{out} \times n_c$ $\quad$ $1 \times 1 \times n_c$ $\quad$ no of filters $= n_c' = 5$

→ cost of normal conv = 2160

→ cost of depthwise = depthwise + Pointwise = 432 + 240 = 672

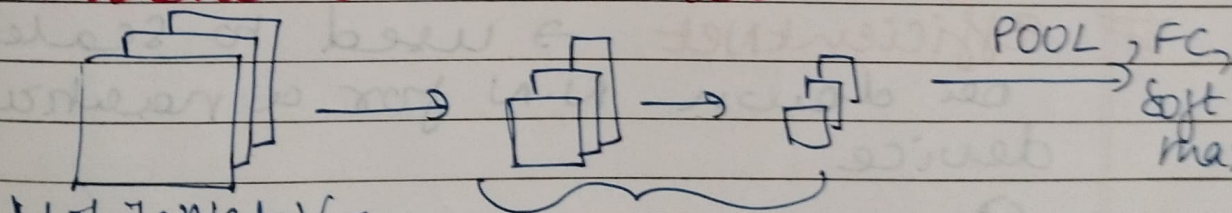
→ $\frac{\text{cost of depthwise+pointwise}}{\text{cost of normal conv}} = \frac{1}{n_c'} + \frac{1}{f^2}$

$= \frac{1}{5} + \frac{1}{9}$

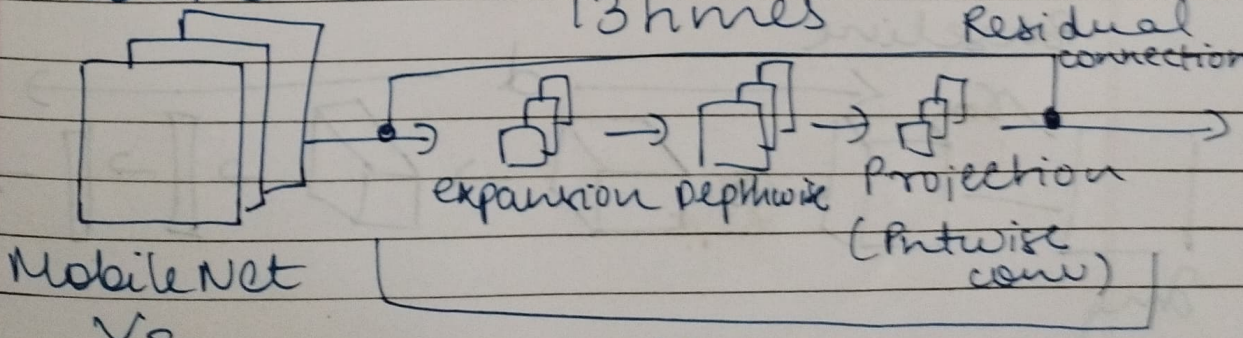
$= \frac{672}{2160}$

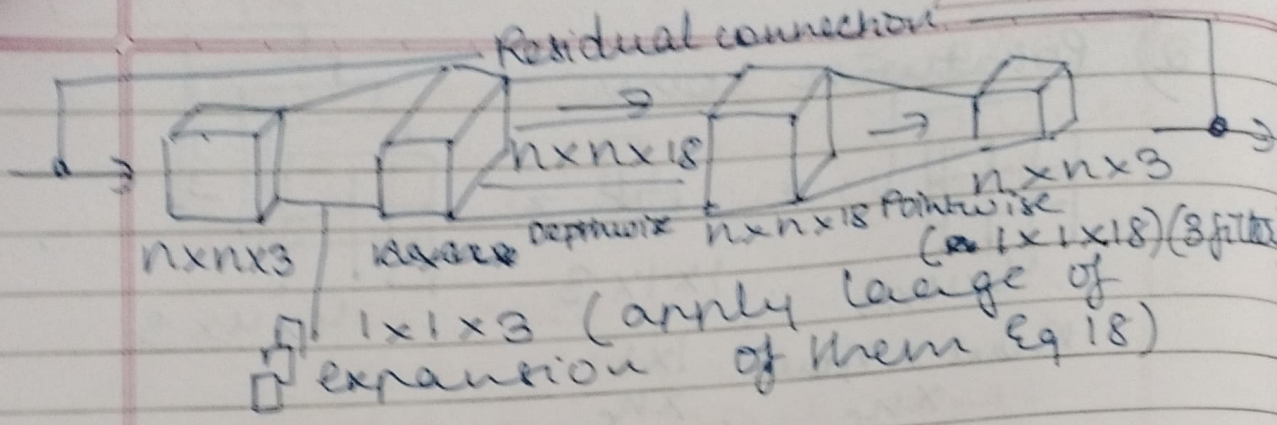
MobileNet Architecture

1)



2)





Bottleneck block

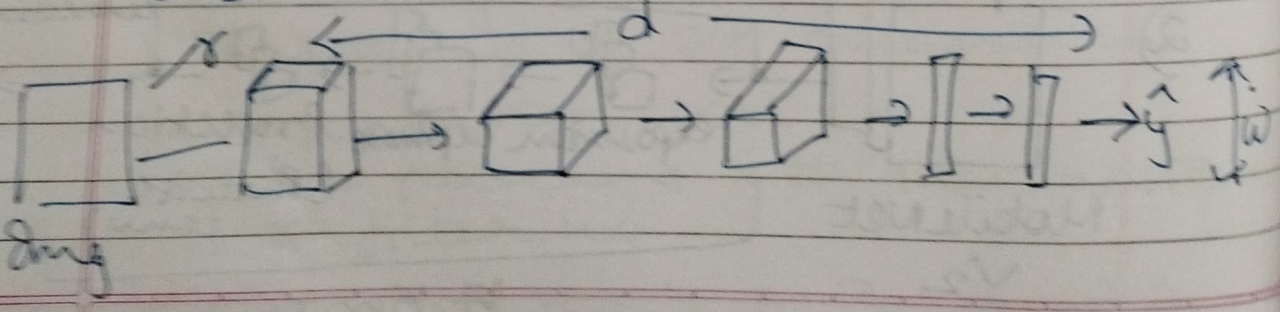
1) By using expansion operation increases size of representation within bottleneck block this allows N.N to learn a richer function

2) But in mobile there is memory constraints. Bottleneck uses pointwise conv as projection to project it back to smaller set of values so when we pass this to next block amnt of memory to store these values is reduced back down

EfficientNet → used to scale up or down N.N for a particular device

→

Baseline



to scale things up or down
→ New Img resolution r ↑ deeper network ie vary d/ vary w
→ Given a computational budget — what's the good choice of r/d/w or depending on computational resources, we can also use compound scaling where we simultaneously scale up or down the resolution of the img & depth & width of NN
→ Rate at which we shld scale up r, / d / w — look at one of the open src. implementations of EfficientNet which will help choose a good trade off btw r, d & w

Practical Advice for Using ConvNets

Transfer Learning

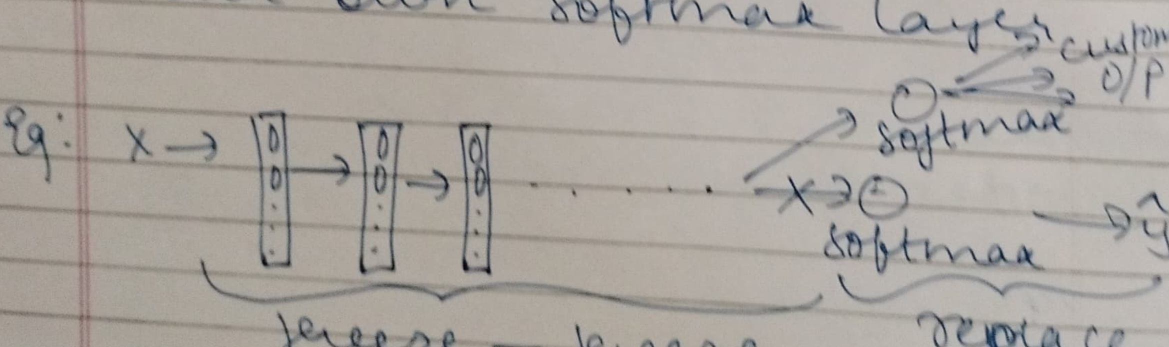
→ For building comp vision application rather than training the ways from scratch random initialization its faster to download ways that someone has already trained and use as pre training and transfer that to new task

→ Types of data sets — ImageNet, MS COCO or Pascal

→ If we dont have lot of training data then we download the open src implementation of a NN (code + weights)

→ Then we can get rid of softmax layer and create our own softmax layer

Eg:



freeze — freeze parameters in all of these layers

replace

→ We get pretty good performance with small data set

⇒ Depending on framework it might have things like trainable parameters so we might set that for some early layers

⇒ There are diff ways by which deep learning frameworks let u specify whether or not to train the weights associated with a layer

⇒ If we have a larger training set we can freeze few initial layers and train the rest of the layers along with our O/P unit

⇒ Couple of ways to do this:

⇒ take last few layers and use that as initialization and do gradient descent ⇒ throw away the last few layers and ~~use gradient descent~~ our hidden units with our softmax O/P

⇒ If we have lot of data we can use the network as initialization and train whole network

Data Augmentation

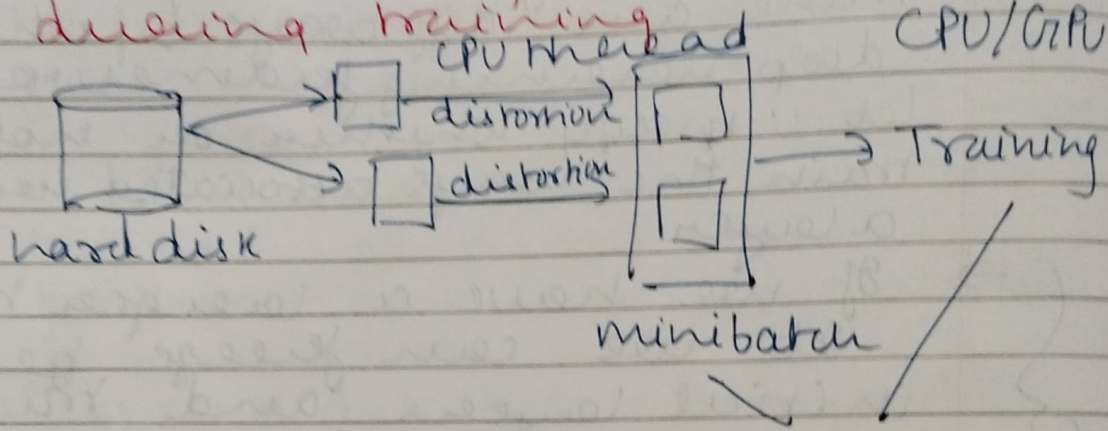
⇒ The simplest form is mirroring on vertical axis

⇒ Another is random cropping

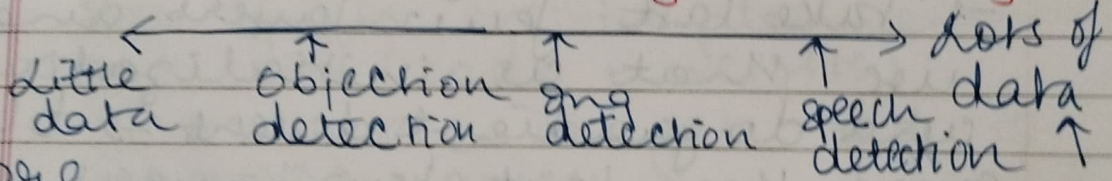
⇒ Rotation, shearing, local warping

⇒ 2nd type commonly used is colour shifting → add distortions to R, G and B channels

Implementing distortion during training



Data vs hand engineering



more hand engineering ("hacks")

Simpler algo less hand engineering

transfer learning

2 sources of knowledge

⇒ Labelled data (x, y)

⇒ Hand engineering features/ network architecture / other components

→ Computer vision relies more on hand engineering & has developed a more complex network architectures.

* Ensembling – Train several networks independently & avg their O/P ($\hat{y}$)
* Multicrop @ test time – Run classifier on multiple versions of test images & average results