WEEK 2

In this week we will look at case studies to get a better understanding of CNNs.

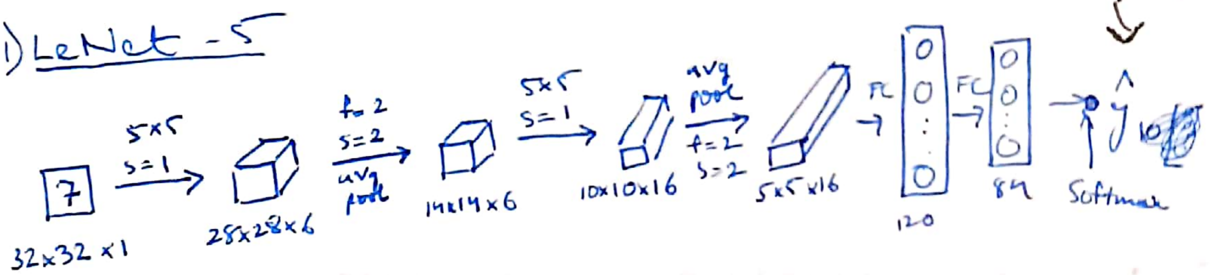
Outline:

- Classic Networks:
  - LeNet-5
  - AlexNet
  - VGG
- ResNet
- Inception

Classic Networks

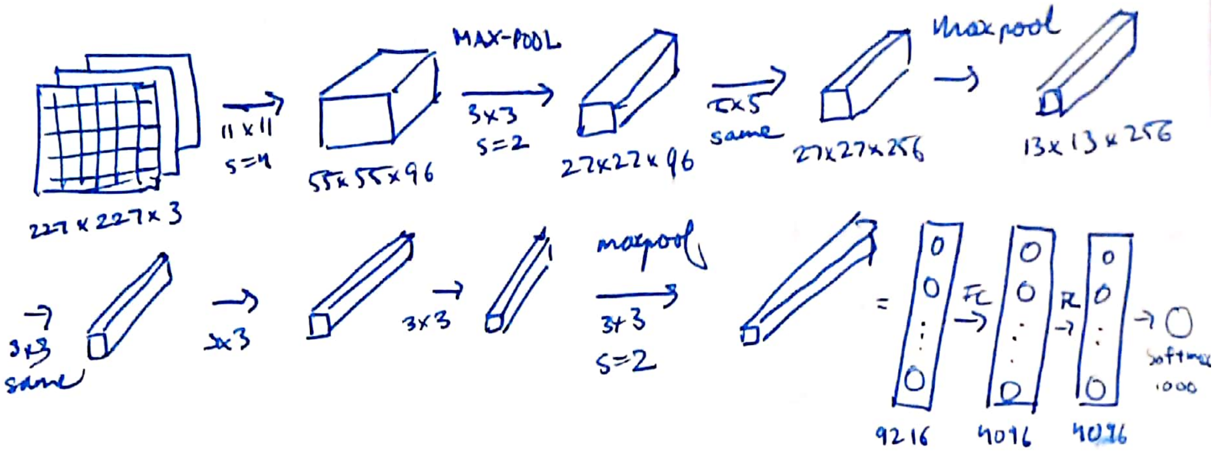
This has 10 elements (cuz 0-9 digits)

1) LeNet-5



- 60 K parameters
- $n_H, n_W \downarrow$ $\quad n_C \uparrow$
- conv→pool→conv→pool→fc→fc→output
- Made for greyscale images
- It was written when computers were slow
- Didn't use ReLu

2) AlexNet



→ similar to LeNet-5, but much bigger

→ ReLU

→ Multiple GPUs

→ ~~Local~~ Used Local Response Normalisation (LRN) – this isn't used anymore since it sucks

→ 60M parameters

3) VGG-16

224x224x3 $\xrightarrow{\text{[CONV 64]} \times 2}$ 224 x 224 x 64 $\xrightarrow{\text{POOL}}$ 112 x 112 x 64 $\xrightarrow{\text{[CONV 128]} \times 2}$

112 x 112 x 128 $\xrightarrow{\text{POOL}}$ 56 x 56 x 128 $\xrightarrow{\text{[CONV 256]} \times 3}$ 56 x 56 x 256

$\xrightarrow{\text{POOL}}$ 28 x 28 x 256 $\xrightarrow{\text{[CONV 512]} \times 3}$ 28 x 28 x 512 $\xrightarrow{\text{POOL}}$ 14 x 14 x 512

$\xrightarrow{\text{[CONV 512]} \times 3}$ 14 x 14 x 512 $\xrightarrow{\text{POOL}}$ 7 x 7 x 512 → FC (4096) → FC (4096) → Softmax (1000)

→ Here all filters, CONV = 3x3 filters, s=1, same

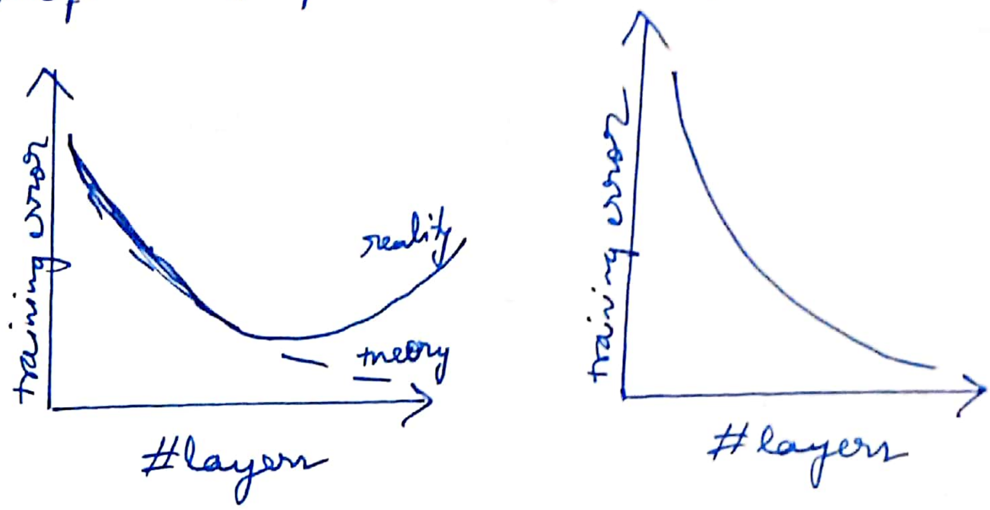
→ MAX-POOL : 2x2, s=2

→ ~138M parameters

→ $n_H, n_W \downarrow$ by 2 while $n_C \uparrow$ – very uniform

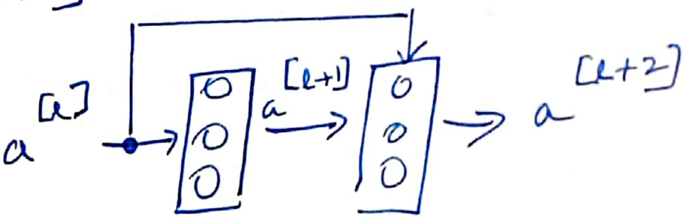
Residual Network

We can train very deep networks without a drop in performance using this.



Plain ResNet

We do this by adding an older $a^{[l]}$ to $z^{[l+2]}$

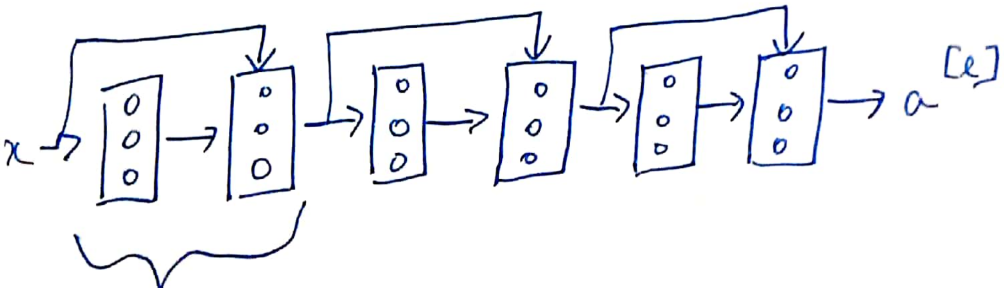


short cut / skip connection

$a^{[l]} \to$ linear $\to$ ReLu $\to$ linear $\to \oplus \to$ Relu $\to a^{[l+2]}$

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

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



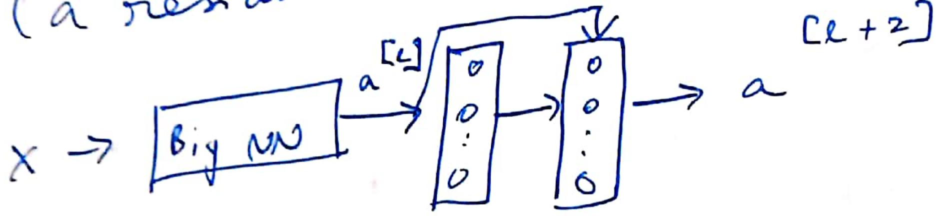
residual block

Why do ResNets work?

Consider a network,

$$x \rightarrow \boxed{\text{Big NN}} \rightarrow a^{[l]}$$

Now suppose we add 2 more layers (a residual block)



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

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

In deeper NN, the layers find it hard to detect parameters, so suppose

$W^{[l+2]} = 0$, $b^{[l+2]} = 0$

Then $a^{[l+2]} = a^{[l]}$

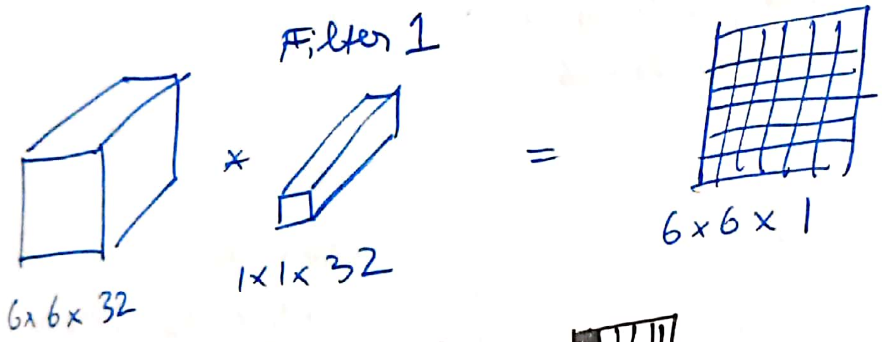
So if no parameters are found, the performance won't be affected.

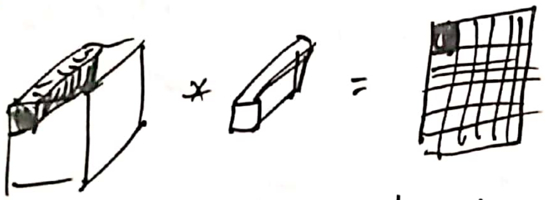
1x1 convolution

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \boxed{2} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

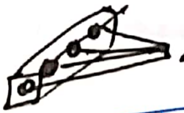
i.e. we multiply all the elements by the 1x1 filter

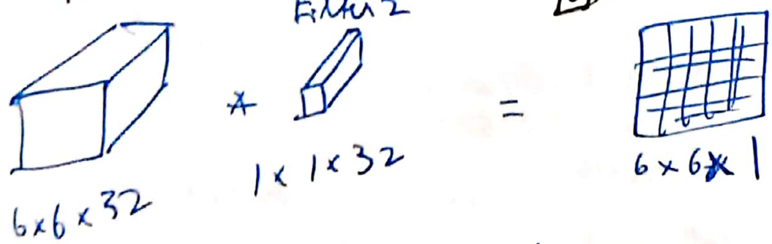
This seems useless, but when we consider a 3D network, its different



Here 

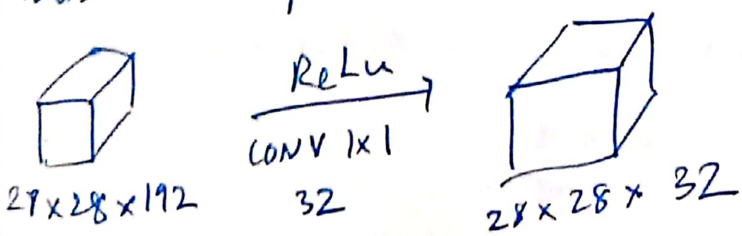
So we multiply element wise and add it all up

This is similar to 



stack these up to get  6x6x #filters

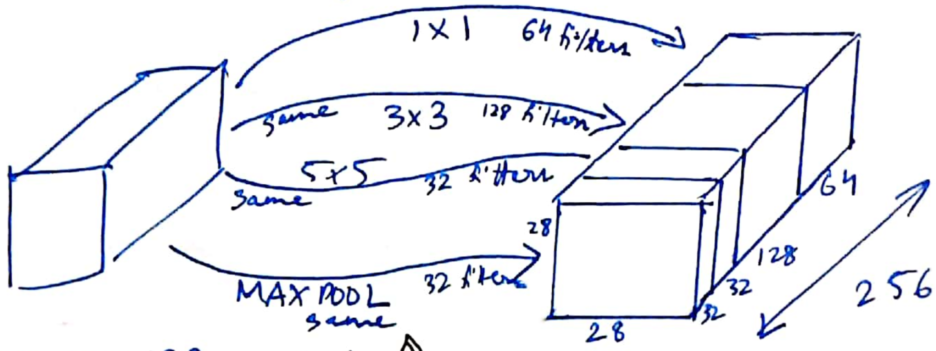
This is useful to reduce no. of filters:



192 reduces to 32

Motivation for inception network

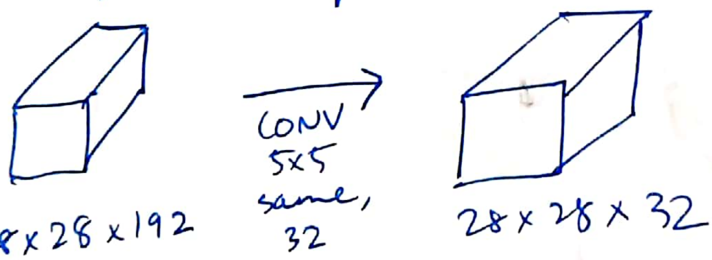
Rather than us choosing the filter size, we try all:



28x28x192   s=1   28x28x256

↑ we need to add padding so it matches 28x28

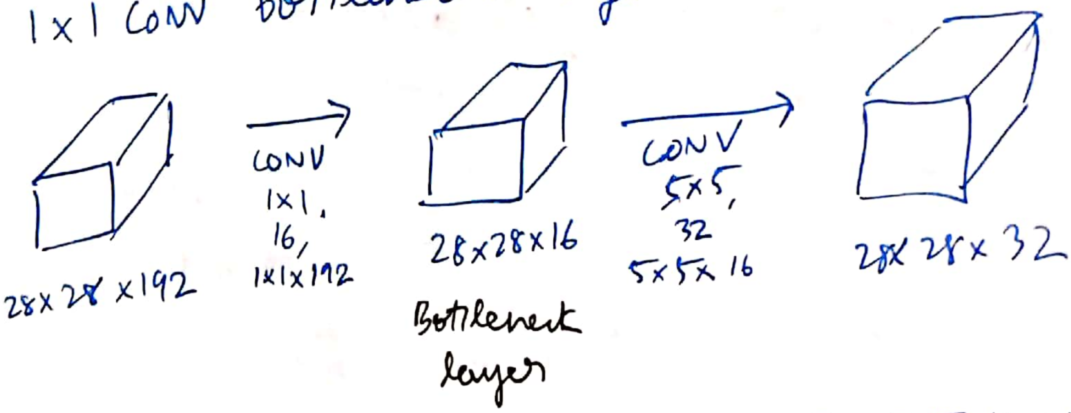
The problem of computation cost:



Consider the 32 5x5x192 filters above

The computation will be $28 \times 28 \times 32 \times 5 \times 5 \times 192 = 120M$

However we can reduce this by using a 1x1 CONV bottleneck layer:

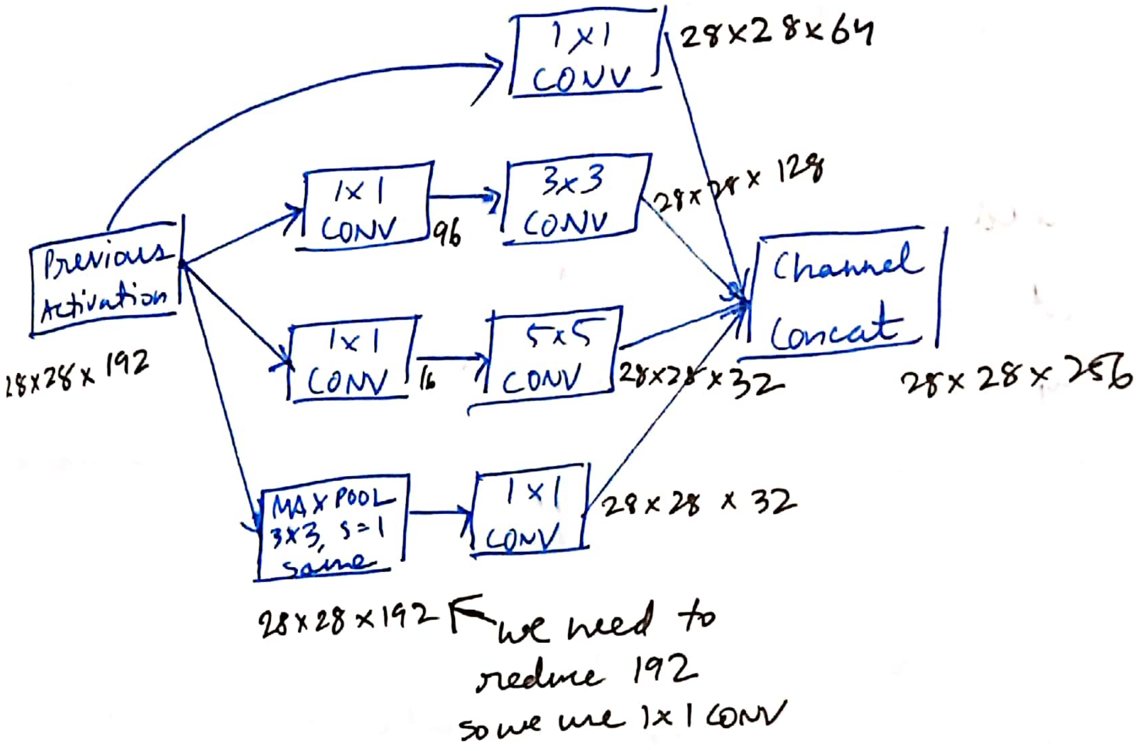


$28 \times 28 \times 16 \times 192 = 2.4M$ + $28 \times 28 \times 32 \times 5 \times 5 \times 16 = 10M$

$= 12.4M$

Hence its reduced.

Inception module / GoogleNet

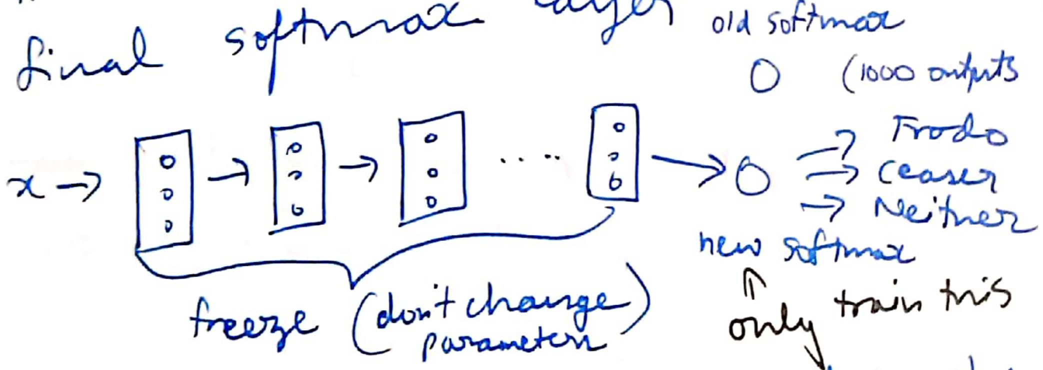


- The inception network contains many inception modules connected to each other in series
- It also contains additional side-branches that each make a prediction, and all these predictions are later connected to a softmax output
- The name comes from the movie Inception. "We need to go deeper"

Practical Advice for using ConvNets

1) Using Open-Source implementation:
Rather than studying a paper and building the network from scratch, check out github to find the open source implementation. In most cases, the authors of the paper would have made their code open-source.

2) Transfer Learning:
Let's say you need to make a dog detector (Frodo, Ceaser, Neither). You can download an existing pre-trained model from the internet, and only replace the final softmax layer

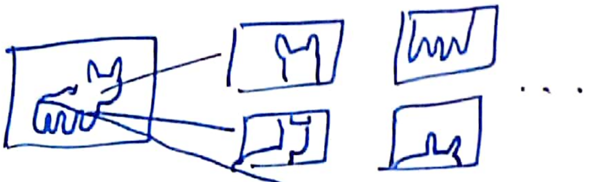


If we have more data, we can reduce the no of layers we freeze

3) Data Augmentation:

We can use this to increase our data.

- Mirroring 

- Random cropping 

However this may not be good ~~since~~ what if it crops to something like 

But usually it works fine

- Rotation
- Shearing
- Local Warping

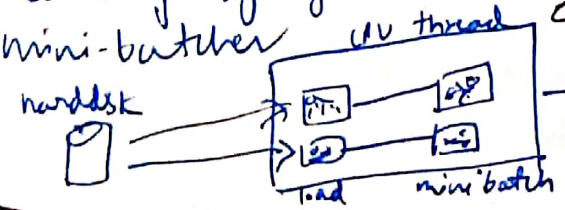
} Used less

- Color shifting - we add/minus RGB values by small values

This makes your model more robust to colour changes (sunlight, etc)

- PCA color augmentation: If the image has more R & B values, it'll subtract a lot to R & B than G. It's used in Alexnet paper.

We can implement distortions during training by generating a stream of distorted mini-batches ← These 2 happen parallely



→ Training

h) State of Computer Vision

Little Data ⟷ Lots of data

Object detection — Image Recognition — Speech recognition

Little Data:
- complex algos
- More hand-engineering
- Can use transfer learning

Lots of data:
- Simpler algos
- Less hand-engineering

Image Recognition: Although we have a lot of data, the problem is so complex that we wish we had more

Two sources of knowledge:
- → Labeled data (x, y)
- → Hand engineering features, network architecture, other components

Tips for doing well on benchmarks/winning competitions
- → Ensembling: Train several networks independently (simultaneously) and average their outputs
- → Multi-crop at test time: Run classifier on multiple versions of test images and average results (10-crop method)

Use open source code:
- → Use architecture of networks published in the literature
- → Use open source implementations if possible
- → Use pretrained models and fine-tune on your dataset