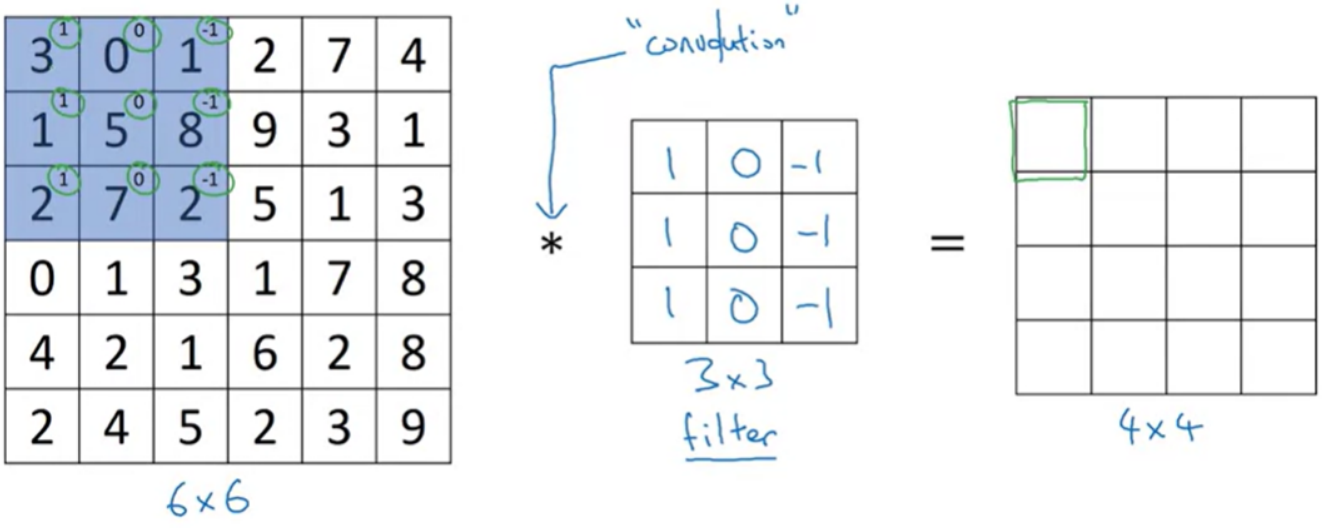
Edge Detection

One of the applications is Edge Detection. Lets say we have a grayscale input matrix of 6x6 as input. We use convolution operation between this matrix and the filter matrix and a filter 3x3 matrix.We get a 4x4 convolution matrix as output. The calculation is as follows :



We choose the 3x3 section of first matrix, multiply each element with corresponding filter matrix element and find their sum. We fill the sum in the corresponding cell of the convolution matrix.

The filter matrices are :

Vertical and Horizontal Edge Detection

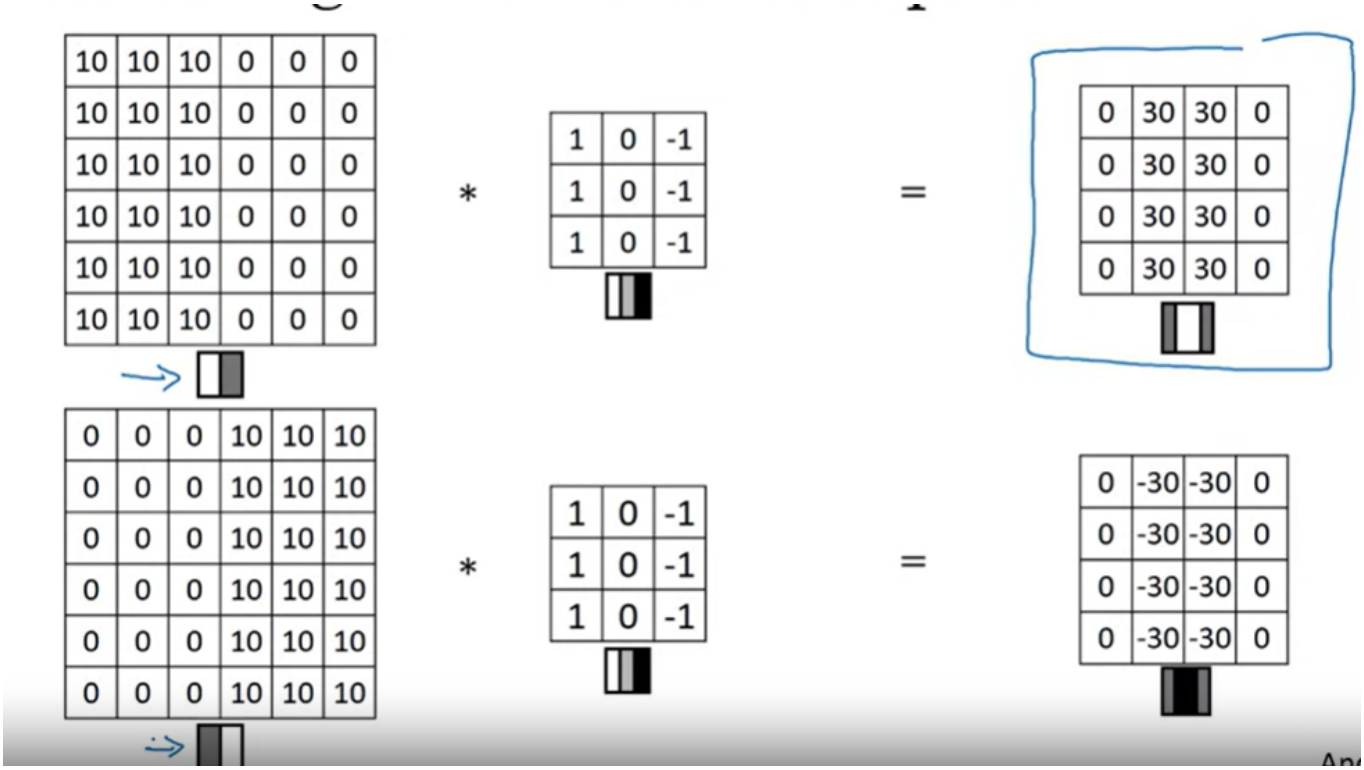
1	0	-1
1	0	-1
1	0	-1

Vertical

1	1	1
0	0	0
-1	-1	-1

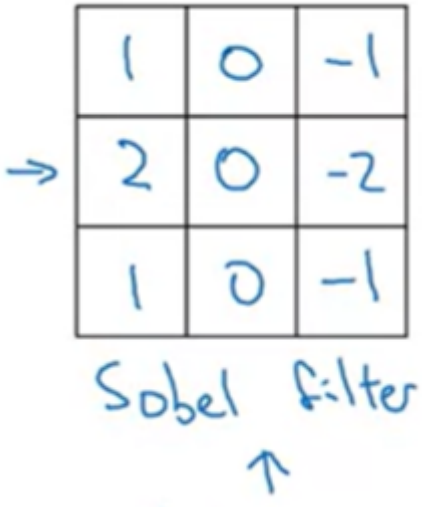
Horizontal

The intuition of edge detection is as follows :



In first image,The 10s show white and 0s show black. There is a sharp vertical edge in the image. After convoluting we get the 4x4 matrix which shows presence of a vertical line in between representing an edge.
On comparing both images we can see that there is a white edge line in first while black edge line in second. This also helps us to find whether there is a light to dark or dark to light transition while going from left to right.

Some popular filters are :

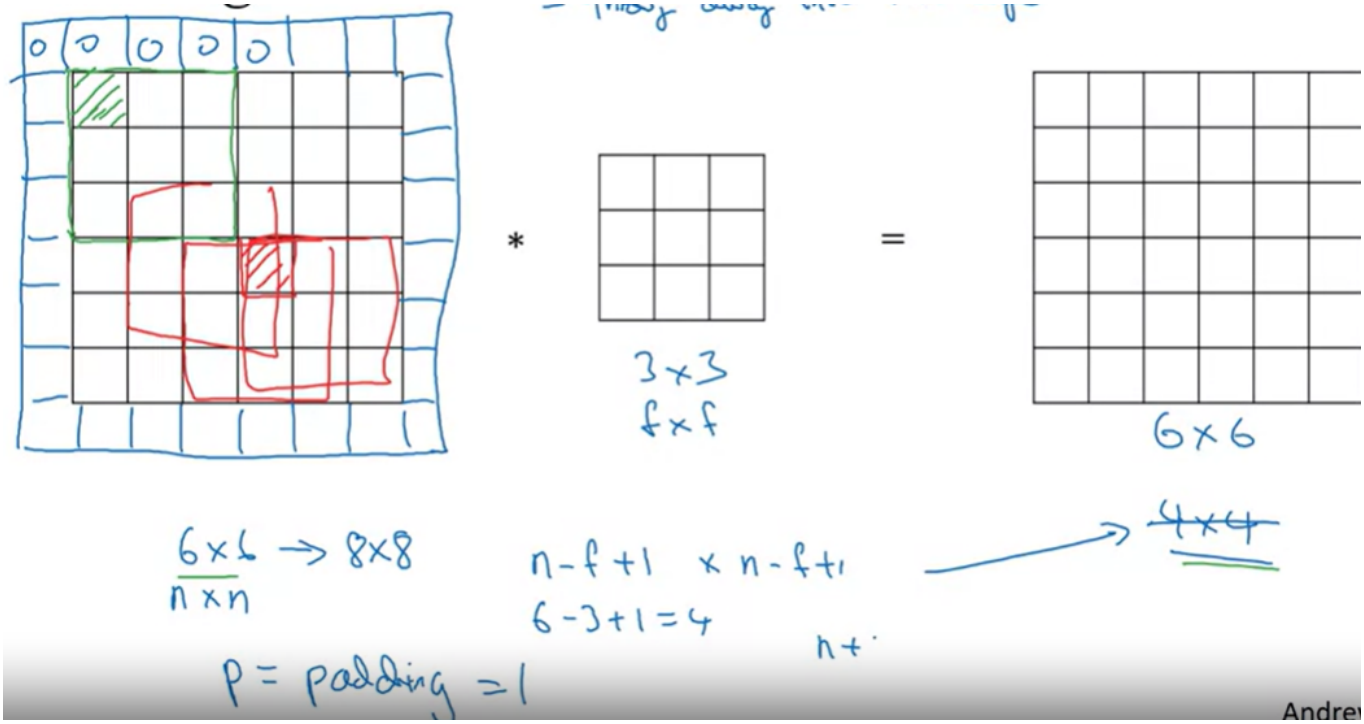


We generally treat the elements of filter as parameters and train them to get a robust model that can detect whatever edges we want.

Function in tensorflow for this purpose is :
"" in keras is :

Padding

In the above edge detection , there is a shrinking in the above convoluted matrix, so to avoid that we add a layer of pixels around the image , which is called as padding



The dimensions of matrices can be characterised and studied as follows :

Valid and Same convolutions

↳ no padding

"Valid": $n \times n \quad * \quad f \times f \quad \rightarrow \quad n-f+1 \times n-f+1$

$6 \times 6 \quad * \quad 3 \times 3 \quad \rightarrow \quad 4 \times 4$

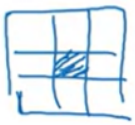
"Same": Pad so that output size is the same as the input size.

$n+2p-f+1 \times n+2p-f+1$

$n+2p-f+1 = n \Rightarrow \boxed{p = \frac{f-1}{2}}$

$3 \times 3 \quad p = \frac{3-1}{2} = 1$ | $5 \times 5 \quad f=5 \quad p=2$

f is usually odd

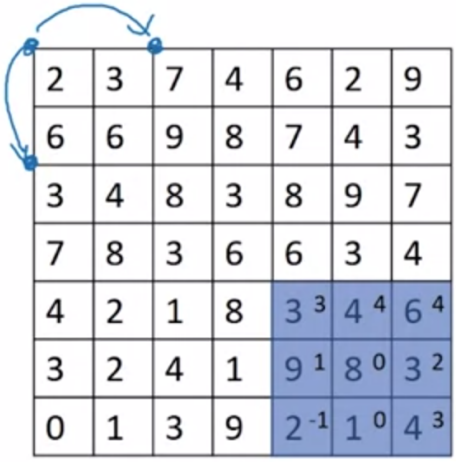


3×3

Andrew N

Strided Convolution

Strided convolution



2	3	7	4	6	2	9
6	6	9	8	7	4	3
3	4	8	3	8	9	7
7	8	3	6	6	3	4
4	2	1	8	3^3	4^4	6^4
3	2	4	1	9^1	8^0	3^2
0	1	3	9	2^{-1}	1^0	4^3

7×7

*

3	4	4
1	0	2
-1	0	3

3×3

stride = 2

=

91	100	83
69	91	127
44	72	74

3×3

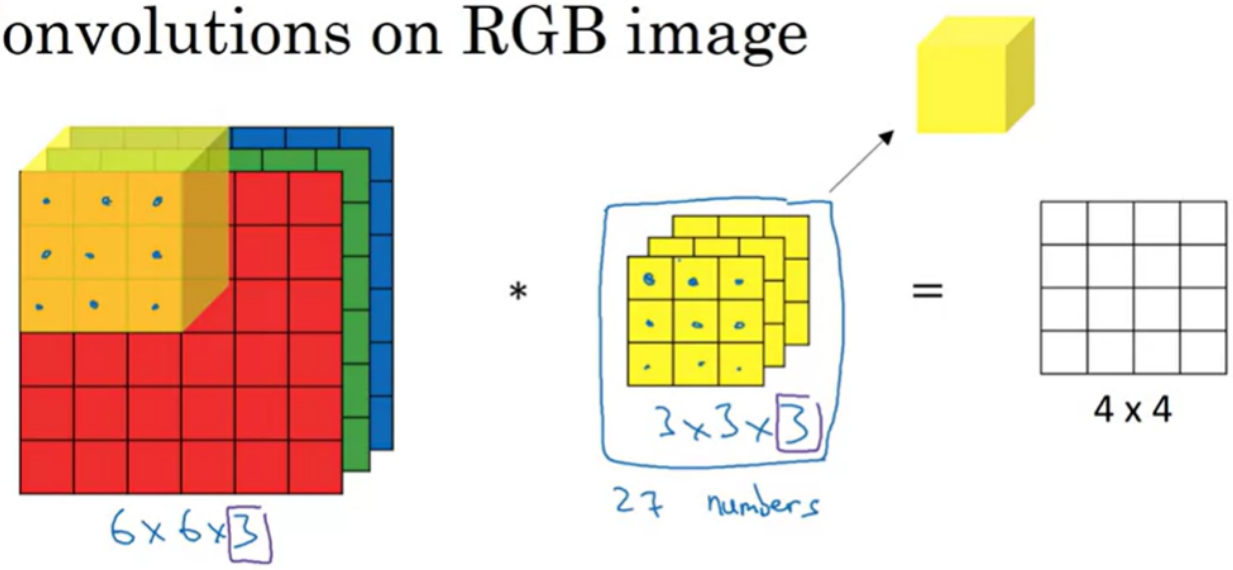
$n \times n \quad * \quad f \times f$

padding p stride s

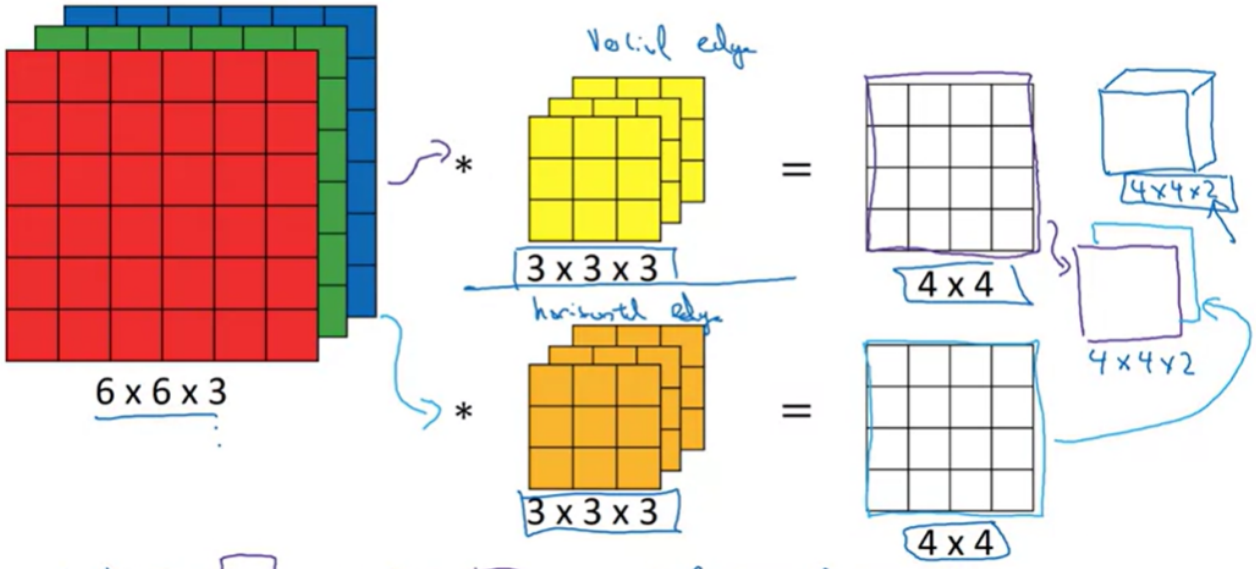
$s = 2$

$\frac{n+2p-f}{s} + 1 \quad \times \quad \frac{n+2p-f}{s} + 1$

Convolutions on RGB image



Multiple filters

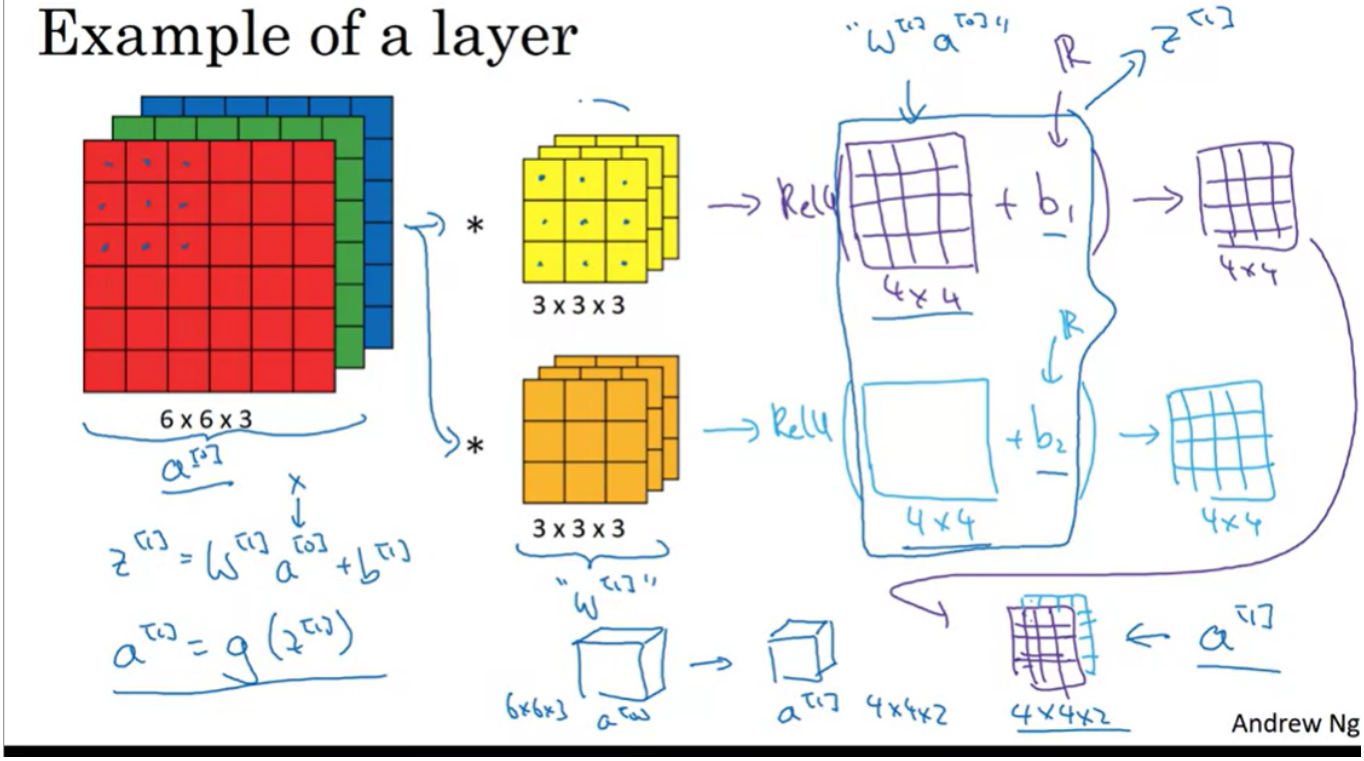


Summary: $n \times n \times n_c$ * $f \times f \times n_c$ → $(n-f+1) \times (n-f+1) \times n_c'$

$6 \times 6 \times 3$ * $3 \times 3 \times 3$ → $4 \times 4 \times 2$ (n_c' = # filters)

Andrew N

Example of a layer



Summary of notation

If layer l is a convolution layer:

$f^{[l]}$ = filter size

$p^{[l]}$ = padding

$s^{[l]}$ = stride

$n_c^{[l]}$ = number of filters

→ Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$

bias: $n_c^{[l]} - (1,1,1,n_c^{[l]})$ ← #filters in layer l.

Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ ←

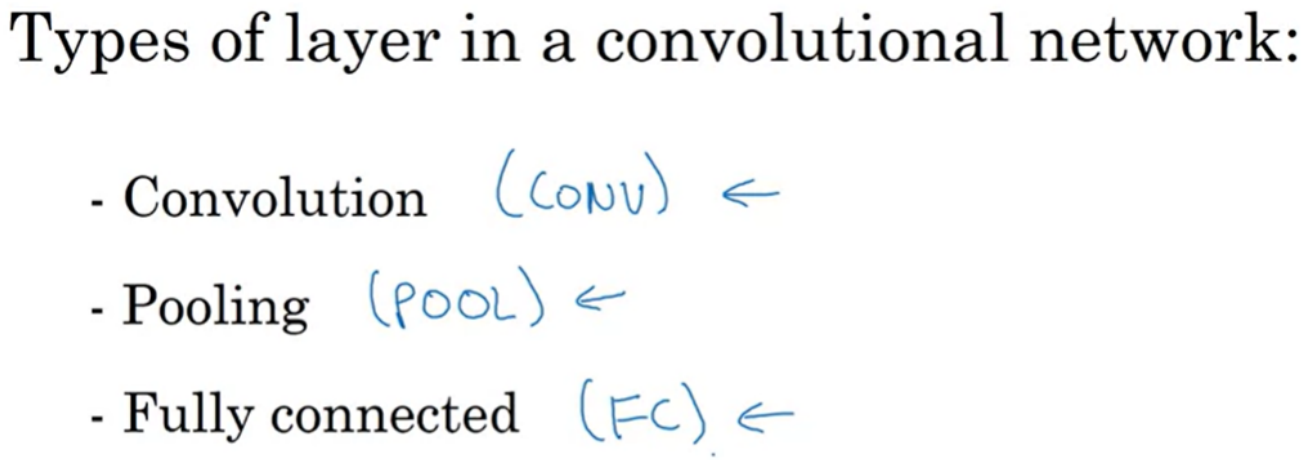
Output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ ←

$$n_{H,W}^{[l]} = \left\lfloor \frac{n_{H,W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

$A^{[l]} \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

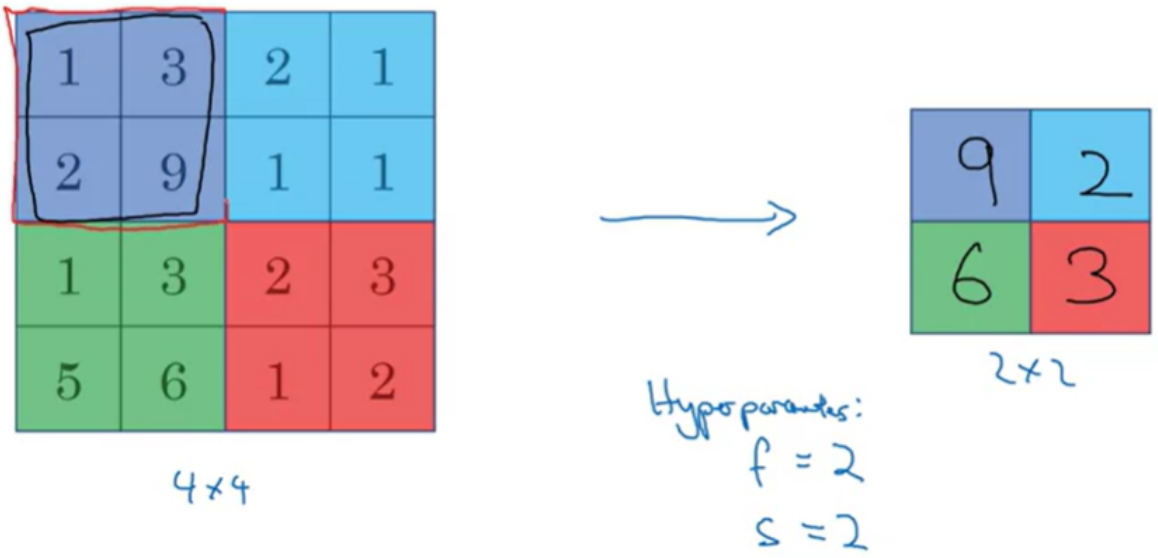
$n_c \times n_H \times n_W$

Depth of output volume = number of filters

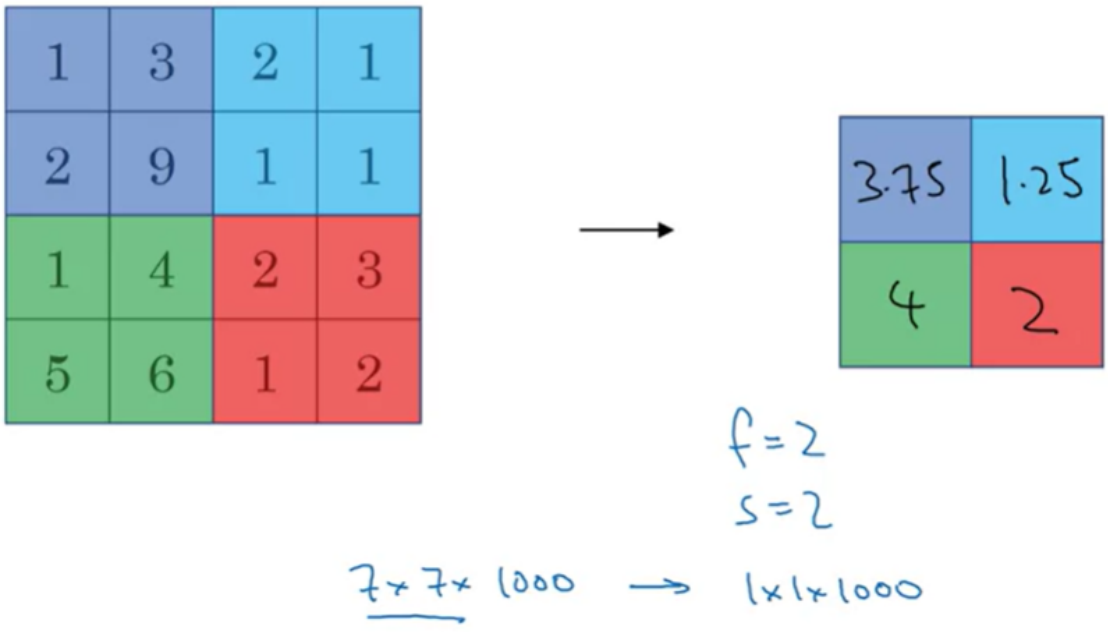


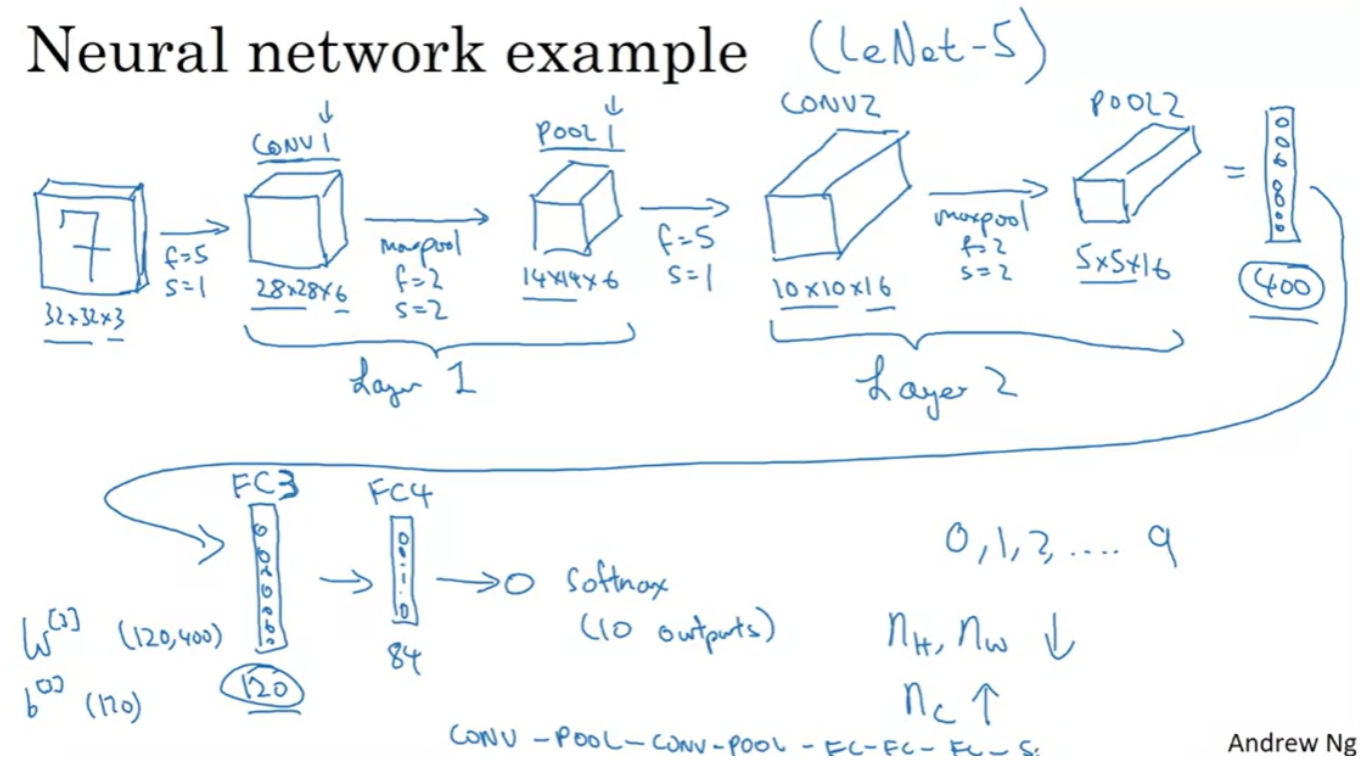
Pooling layers

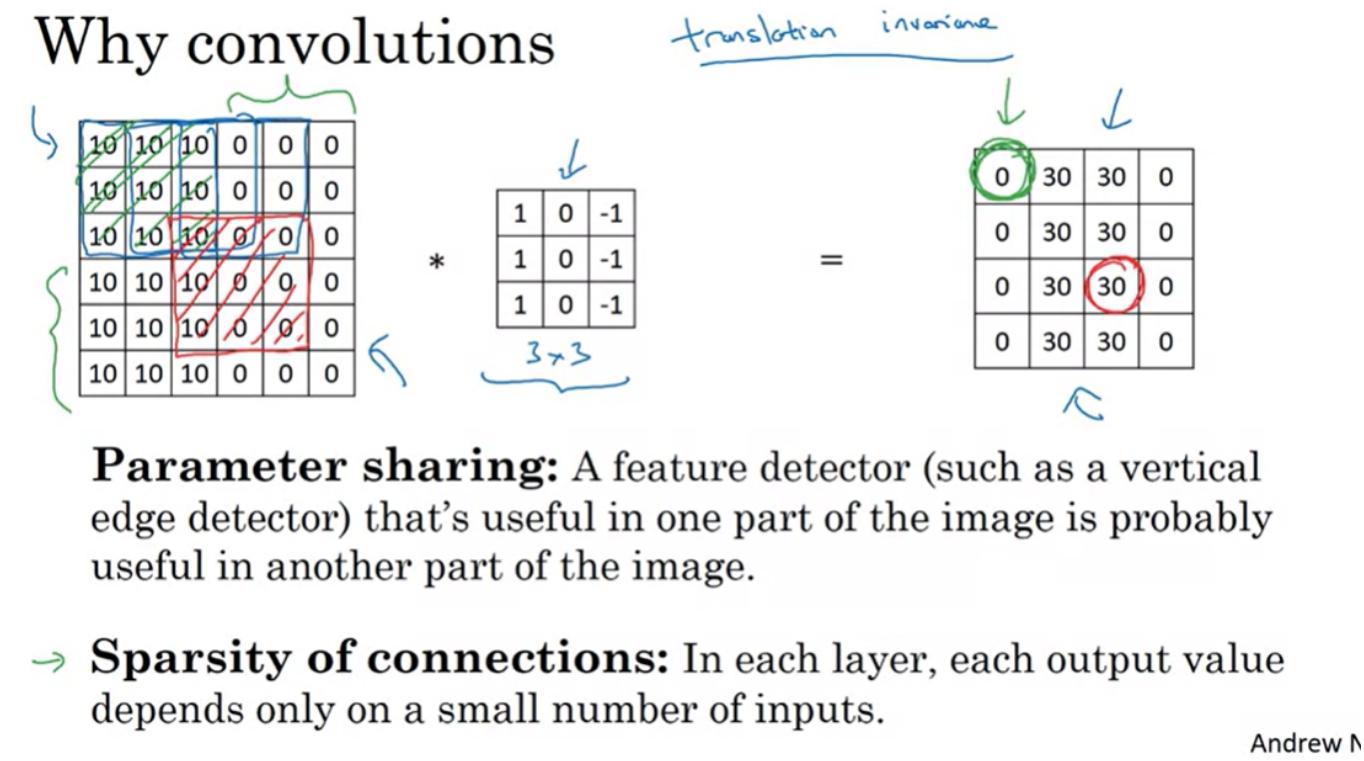
Pooling layer: Max pooling



Pooling layer: Average pooling





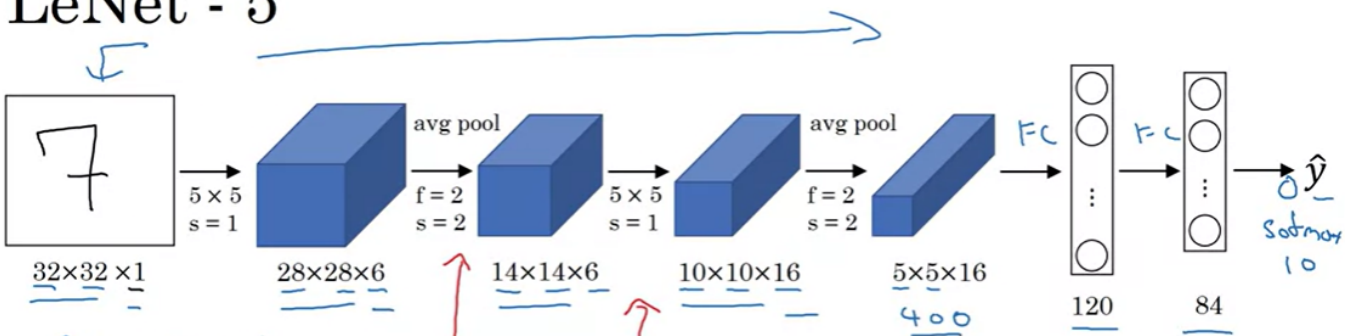


Case Studies

Referring to case studies helps us in formulating our own code.

Classic networks are :

LeNet - 5



60K parameters.

Non-linearity after pooling

$n_H \times n_w \times n_c$ $f \times f \times n_c$

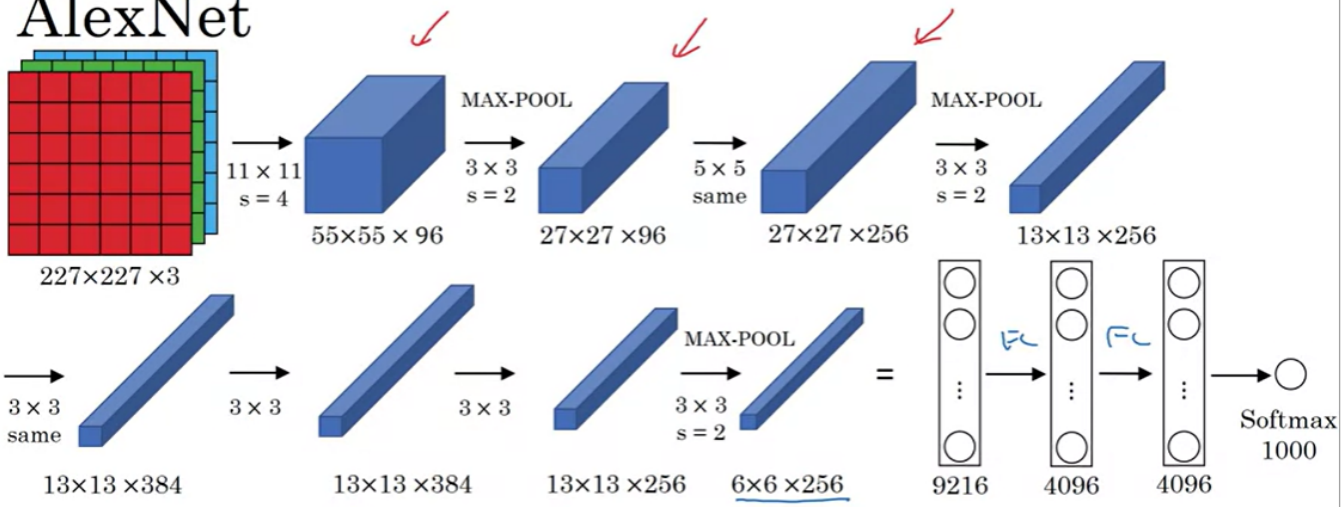
$n_H, n_w \downarrow$ $n_c \uparrow$

conv pool conv pool fc fc output

Activation: Sigmoid/tanh ReLU

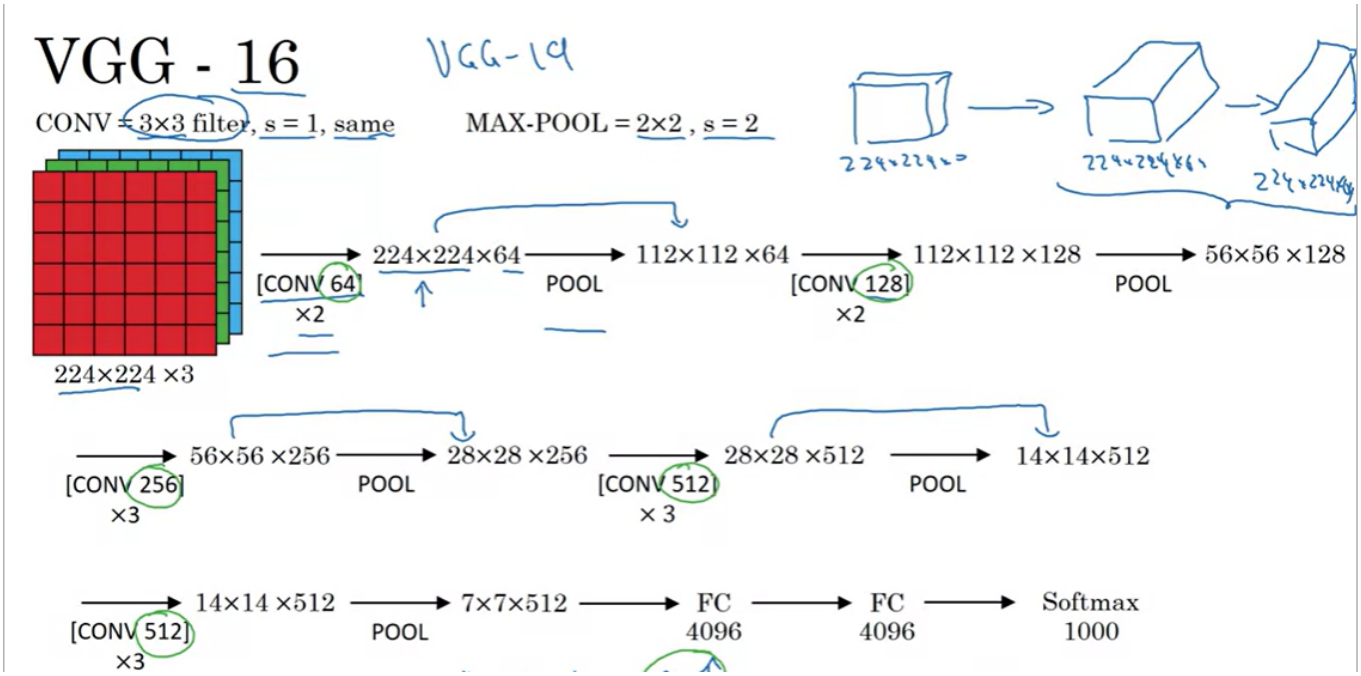
II III

AlexNet

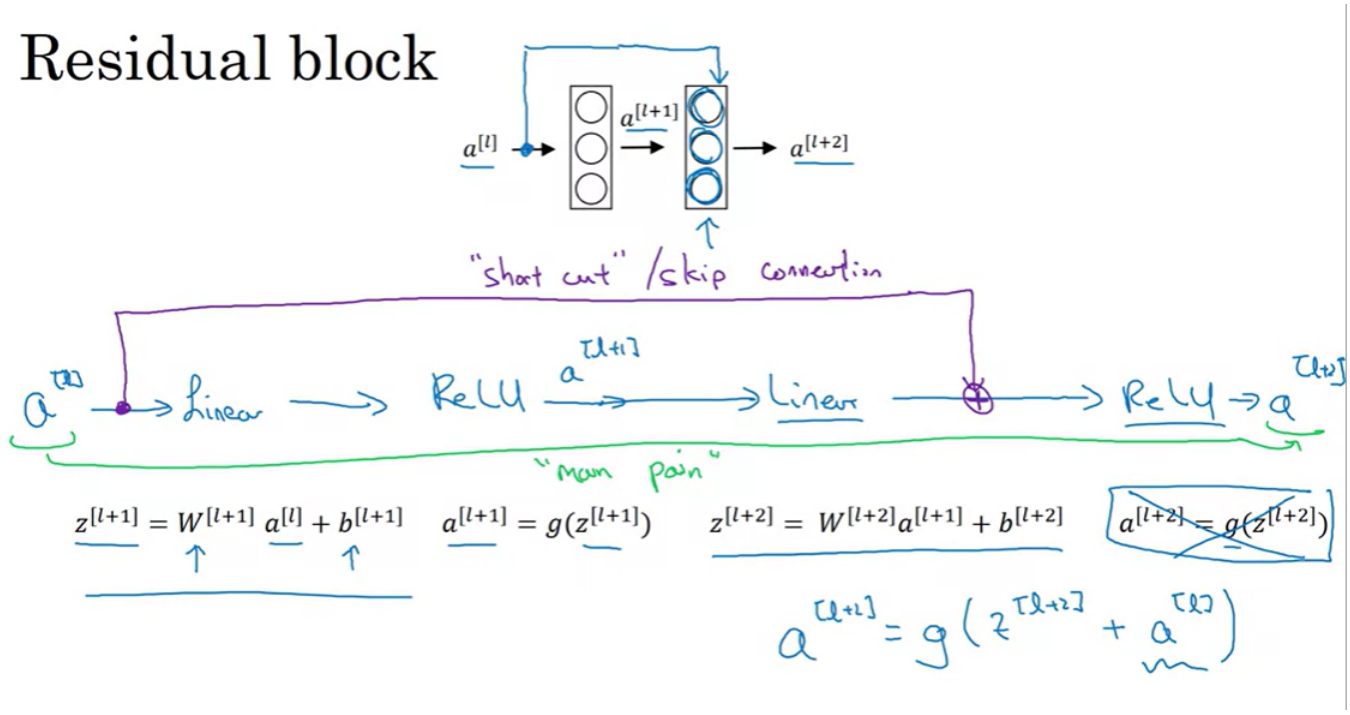


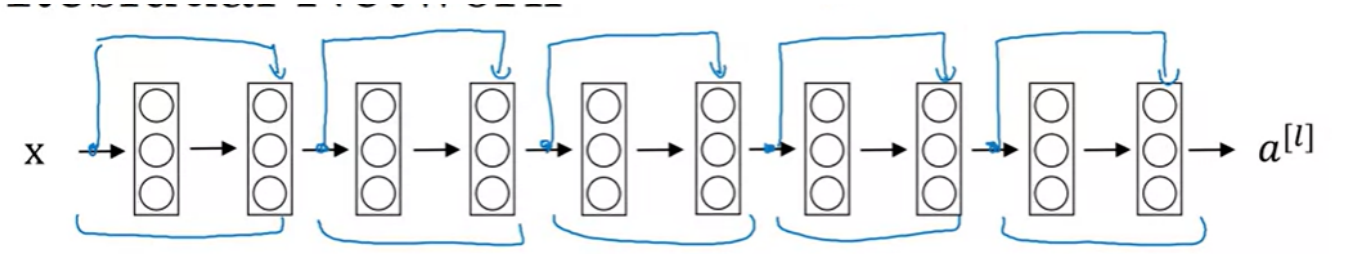
- Similar to LeNet, but much bigger.
- ReLU
- Multiple GPUs.
- Local Response Normalization.

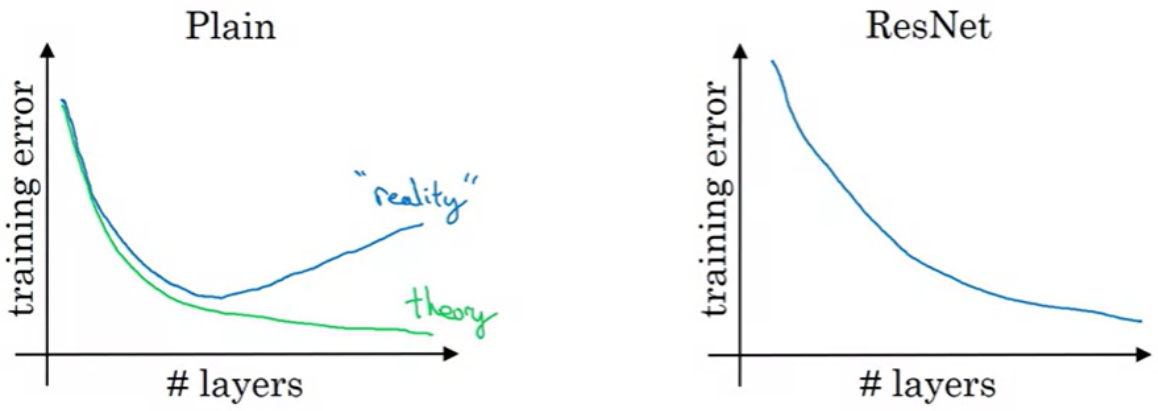
60M parameters



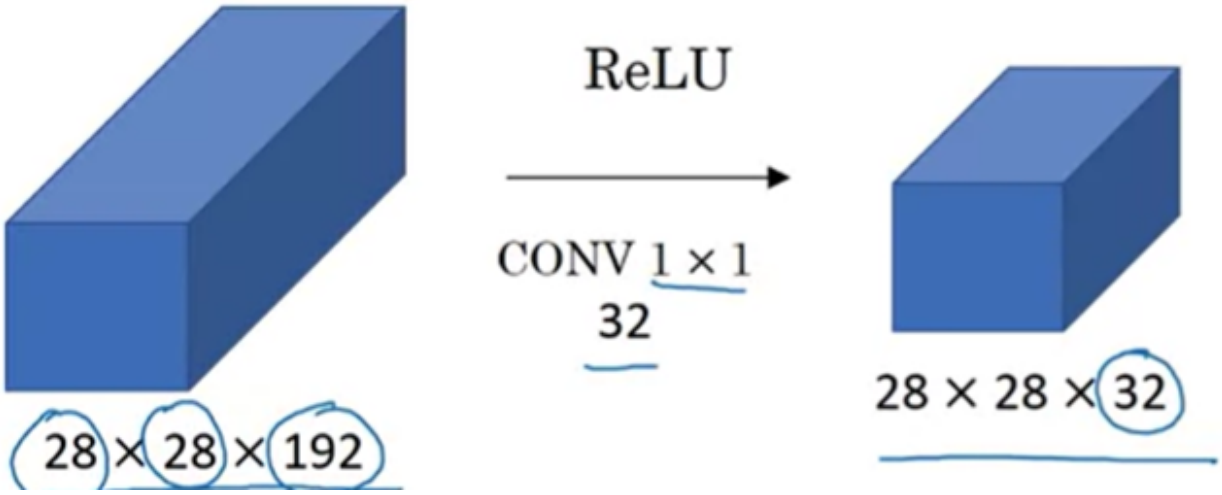
Residual Networks





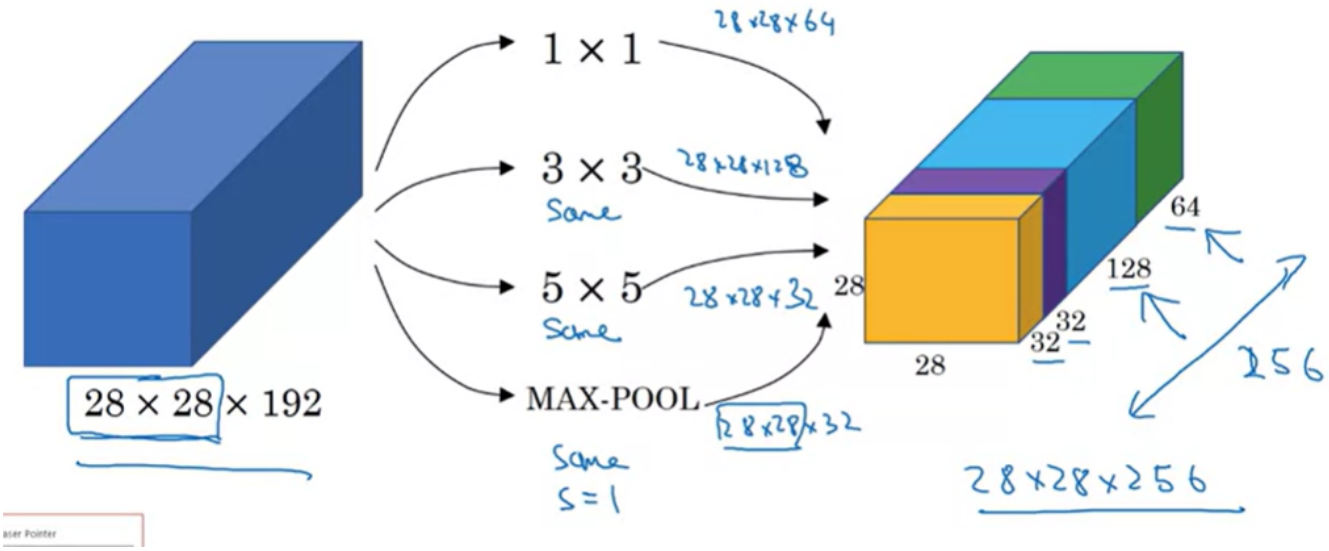


1x1 convolutions help to shrink the number of channels if they have gotten too large. It works as follows :

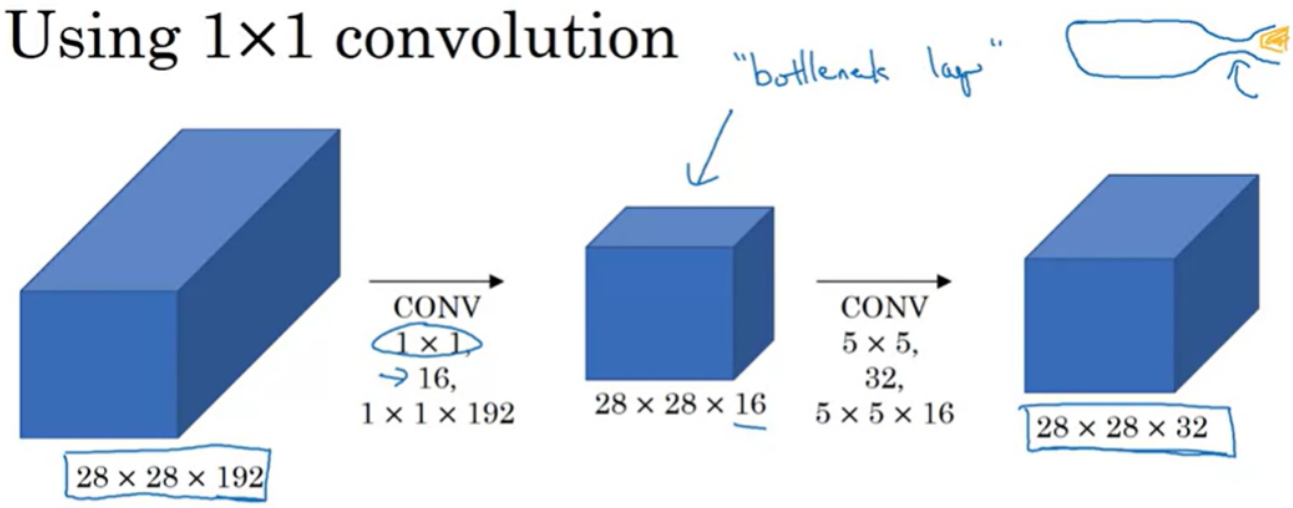


Inception Network

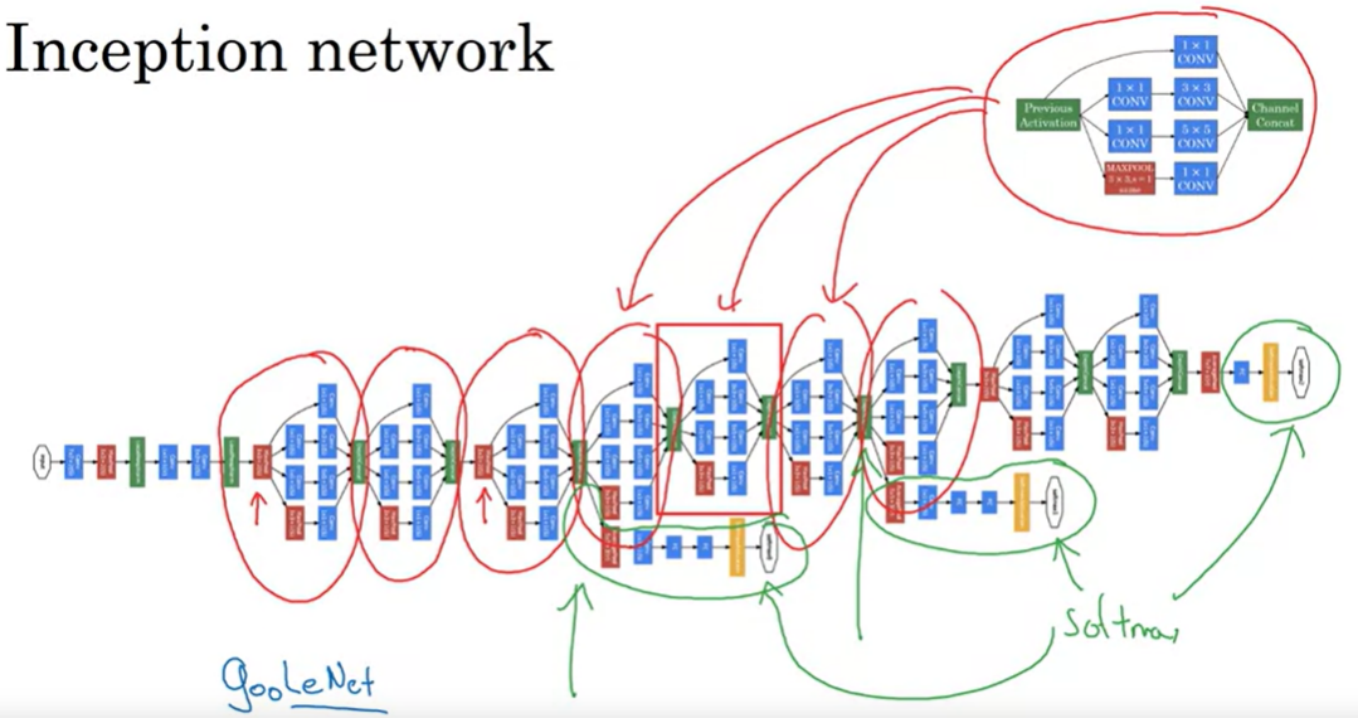
When we are unsure as to what filters and pooling layers to use, we perform all the possibilities at the same time , condense results into one block and concantate the results



However we run into the problem of high computational cost in for eg 5x5 layer above. So we use 1x1 convolutions with an intermediate bottleneck layer as shown below to reduce the computational cost



Example of an inception network



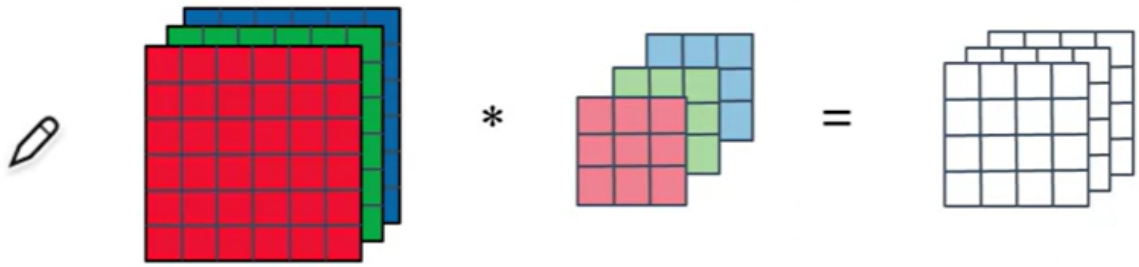
MobileNets

Motivation for MobileNets

- Low computational cost at deployment
- Useful for mobile and embedded vision applications
- Key idea: Normal vs. depthwise-separable convolutions

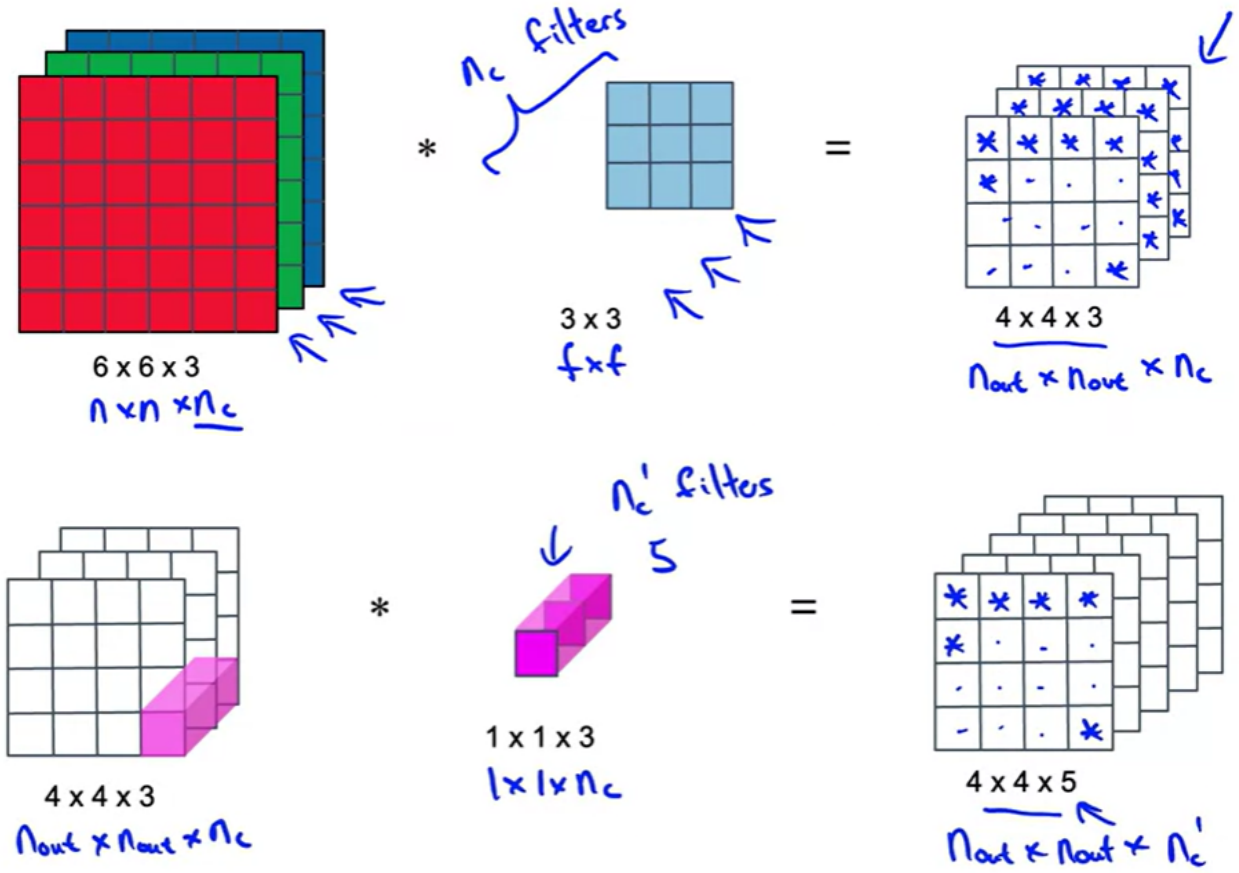
Depthwise Separable Convolution

Depthwise Convolution

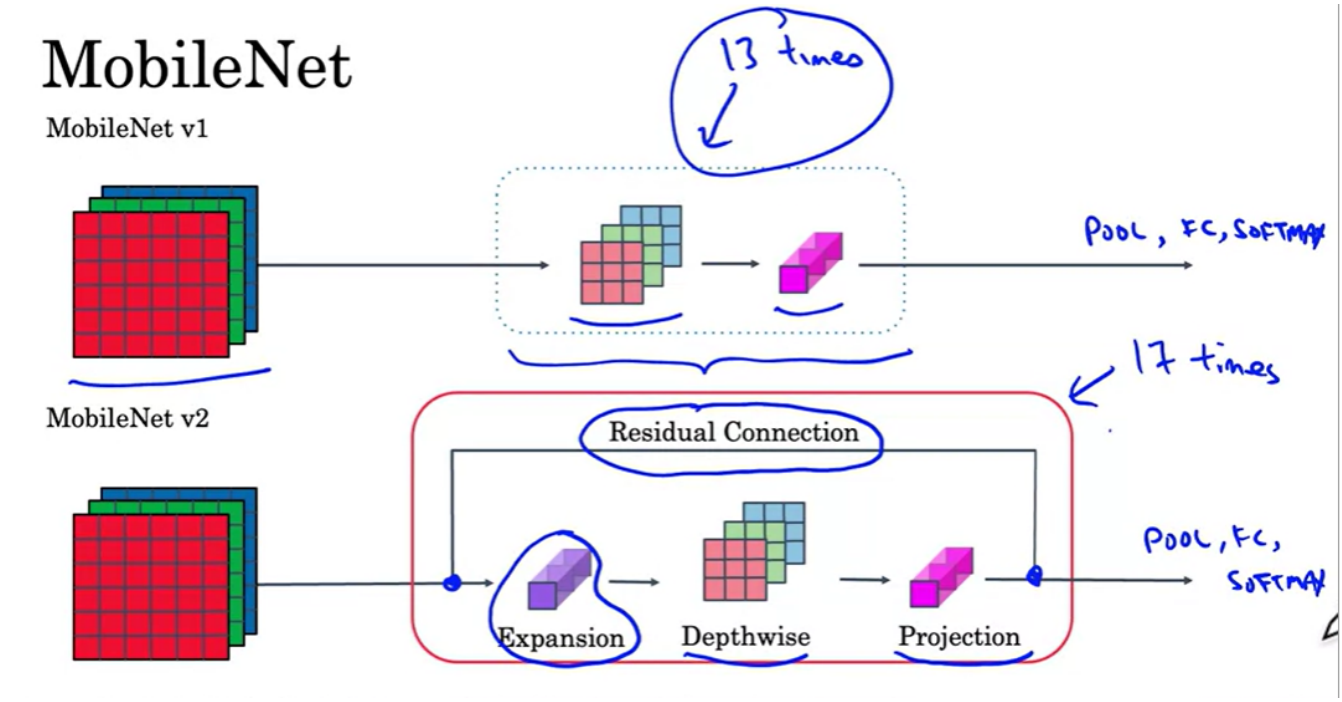


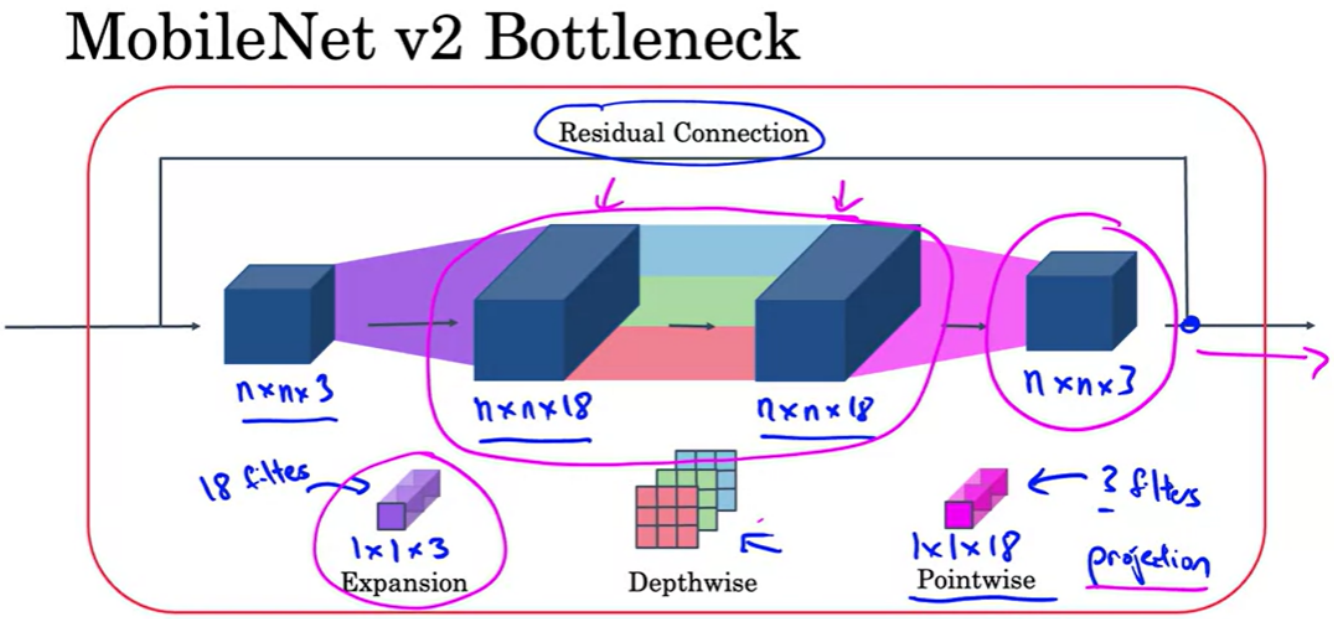
Pointwise Convolution

Depthwise Convolution



In depthwise convolution we convolve the individual coloured layers with corresponding same coloured filters to get an output . It is observed that this convolution has a much lower cost than normal convolution





EfficientNet

Efficiency of a cnn can be improved by

- Increasing resolution of input image
- Increasing depth of the network
- Vary the width of individual layers

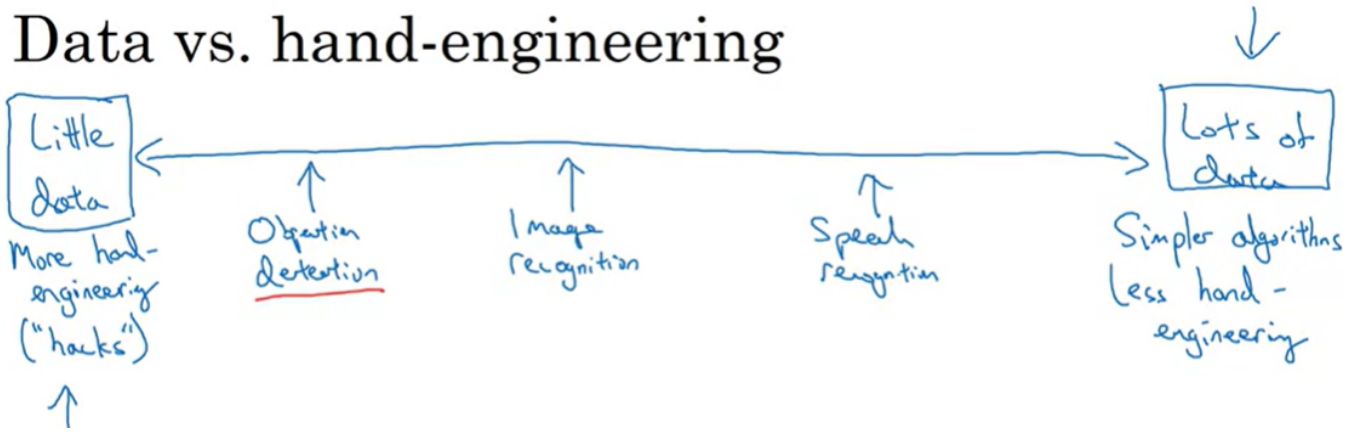
Transfer learning

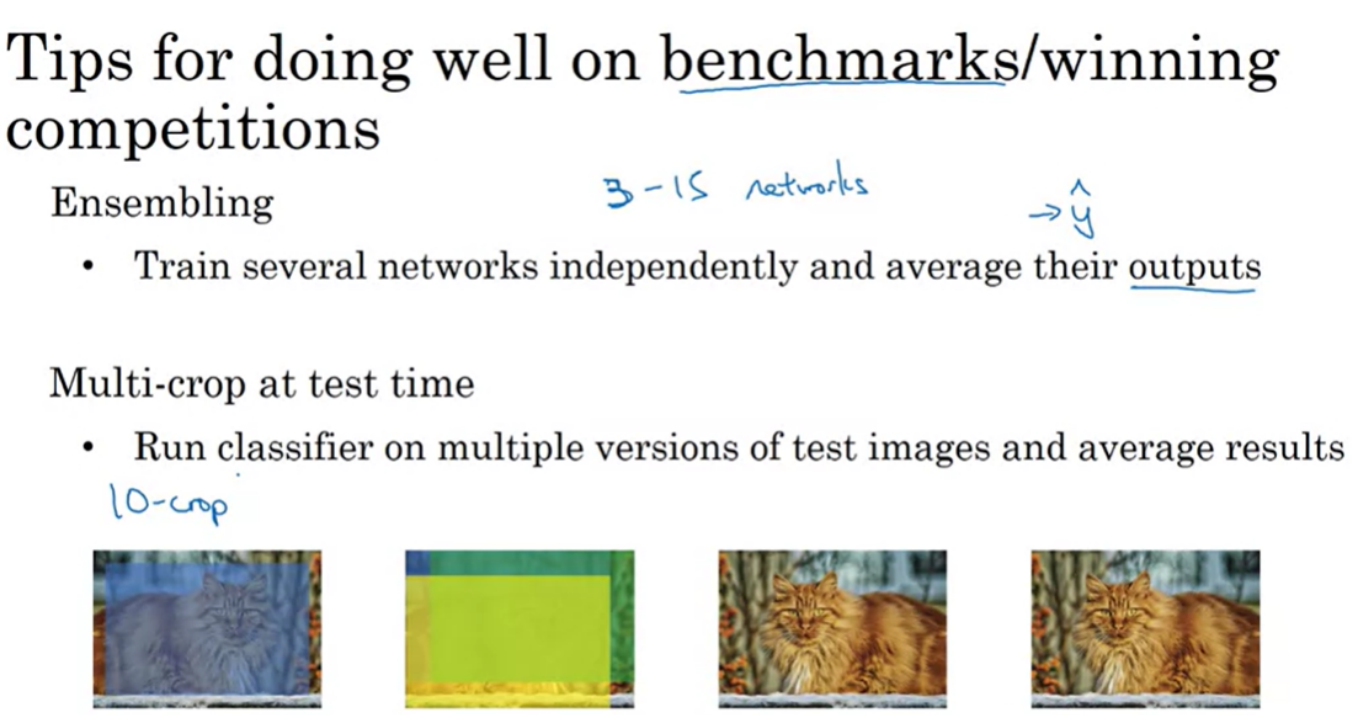
We import pretrained weights of most of layers of network by someone else and freeze those parameters. We only work in the last layers iterating their values. This is generally preferred in

computer vision

Data Augmentation

Many a times in computer vision we come across a scarcity of training examples. So to tackle this we perform various operations on available set of data. This includes mirroring , random cropping , and using colour distortions(adding or subtracting RGB values)

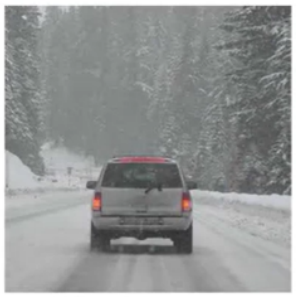




It is better to use open-source, pretrained neural networks and finetune accoriding to your requirements

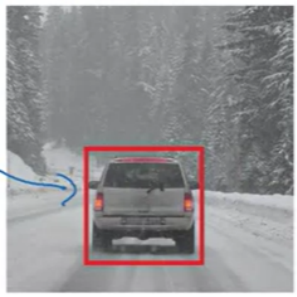
Object Detection

Image classification



"Car"

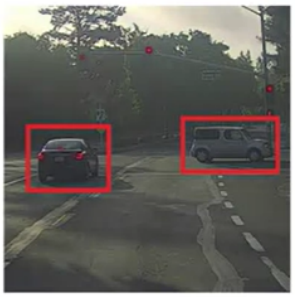
Classification with localization



"Car"

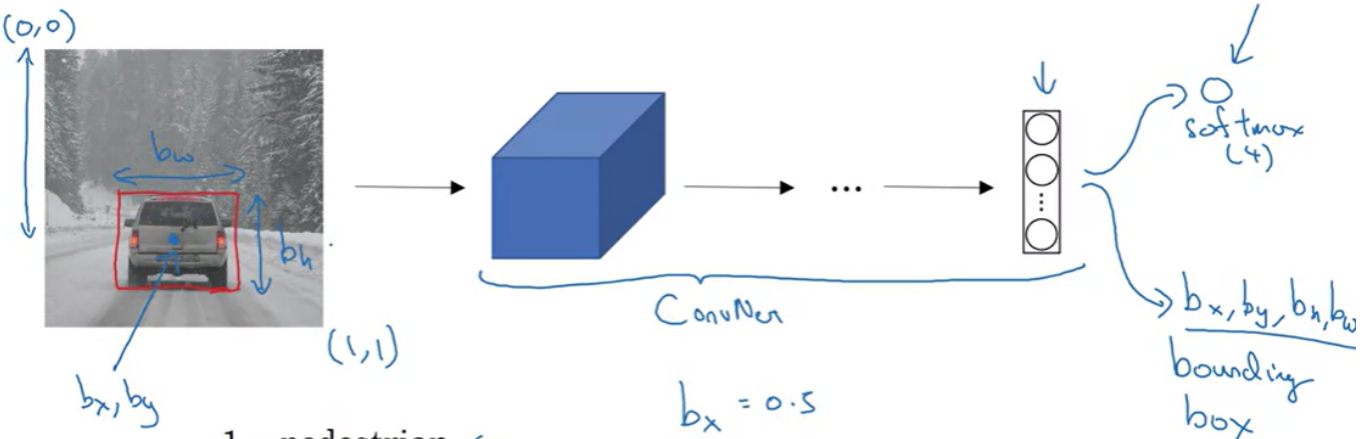
1 object

Detection



multiple objects

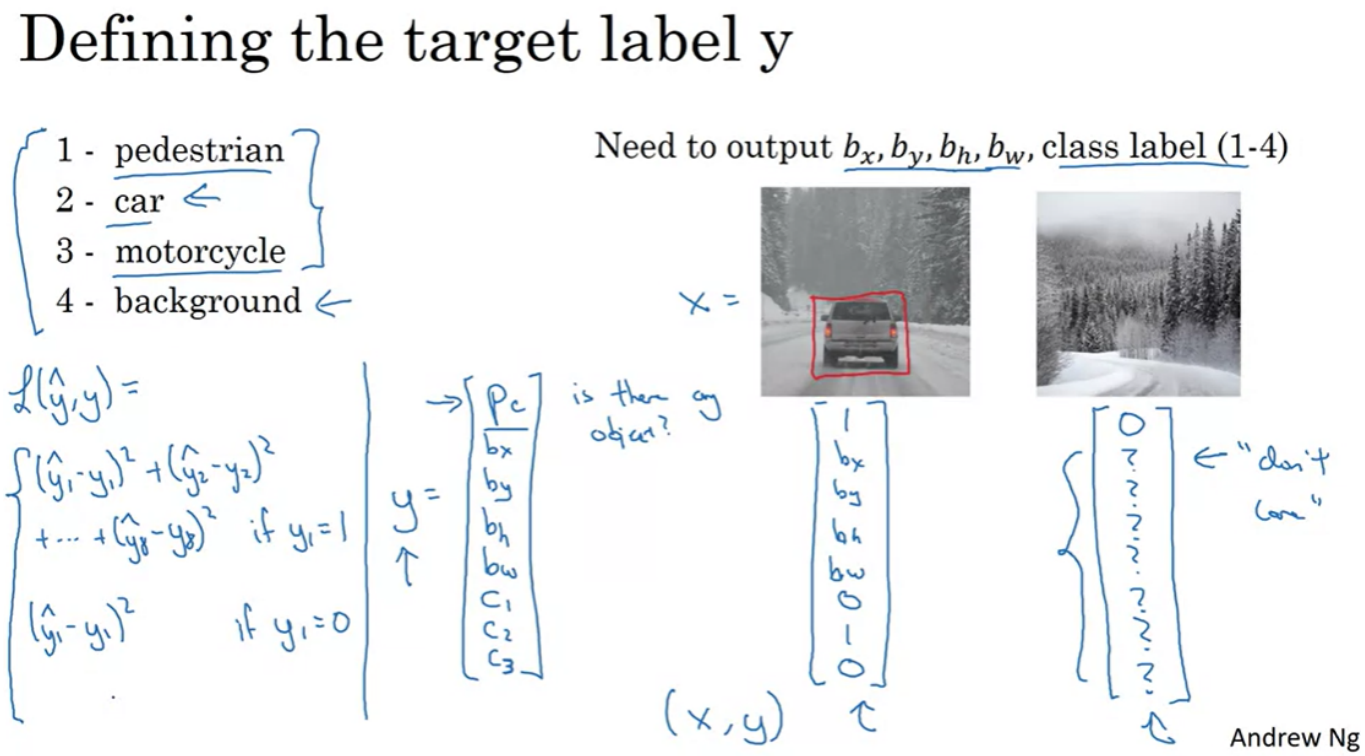
Classification with localization



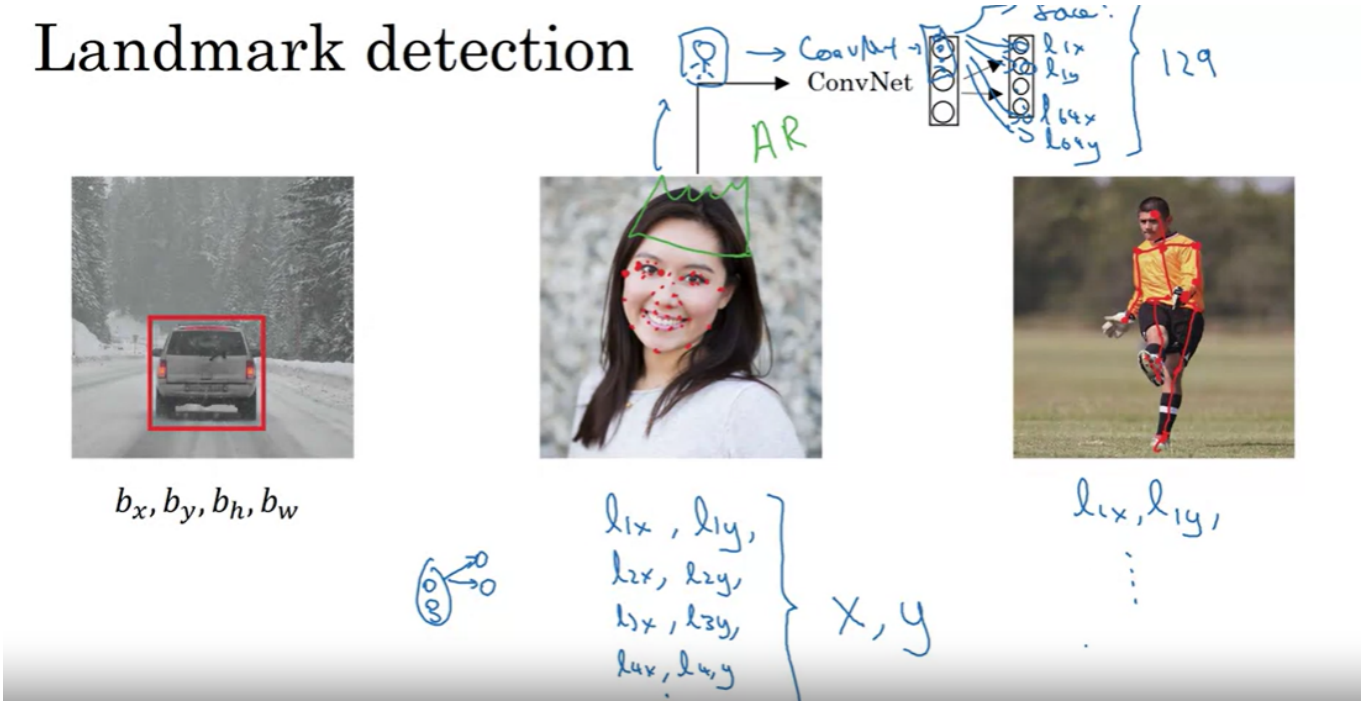
1 - pedestrian ←
2 - car ←
3 - motorcycle ←
4 - background

$b_x = 0.5$
$b_y = 0.7$
$b_h = 0.3$

We can define the centre of object and its dimensions as above.



We can define the y vector as above, with pc being the probability of an object being in the image. We don't care about other parameters if there is no object, hence the loss fucntion is declared accordingly



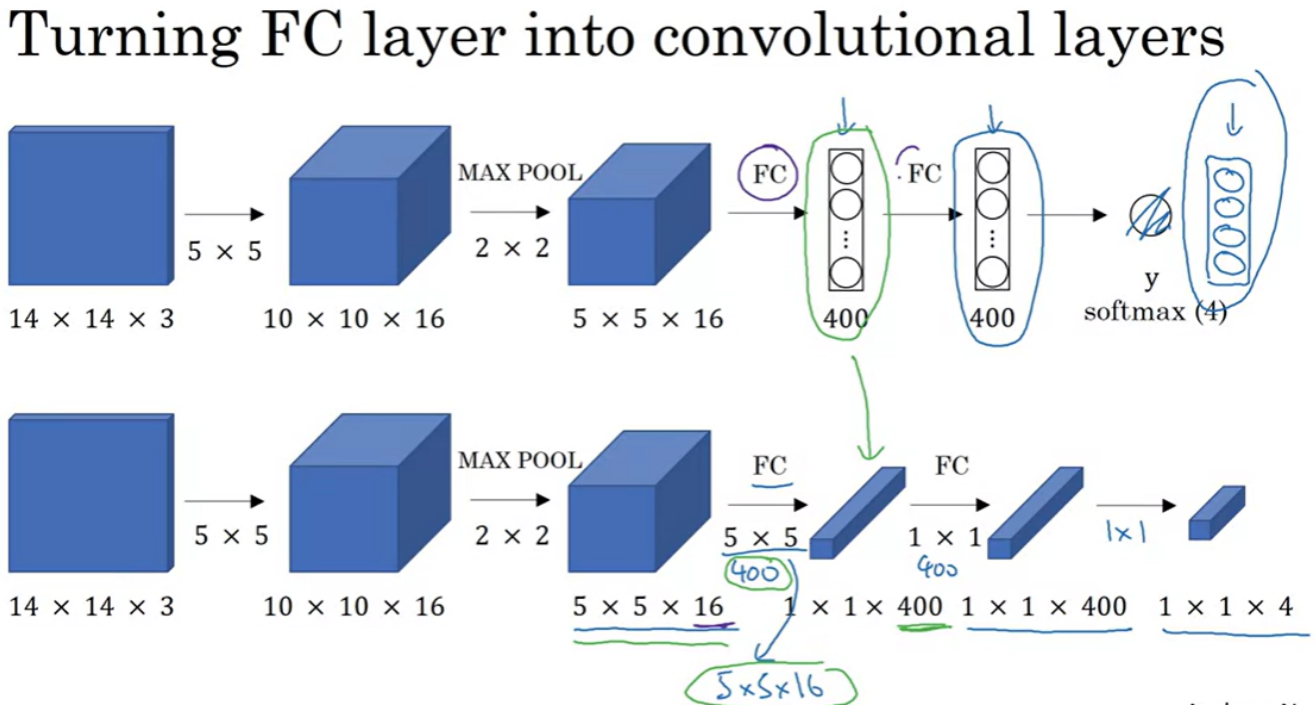
Certain points can be detected . Care has to be taken while labelling

We will first train our model on various car images. Then we use sliding windows method as shown below where we take boxes and pass them over the images and seeing if the pixels

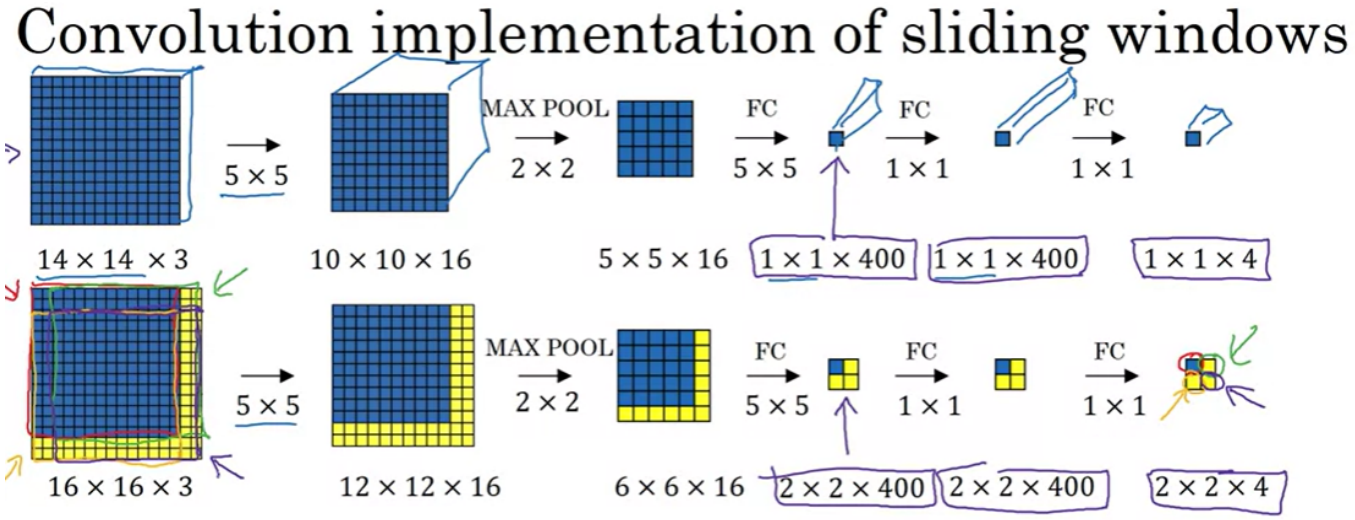
enclosed each time contains a car or not. we will gradually go on increasing size of the boxes

However if the stride is less then we will get more accuracy but computatonal cost will also increase

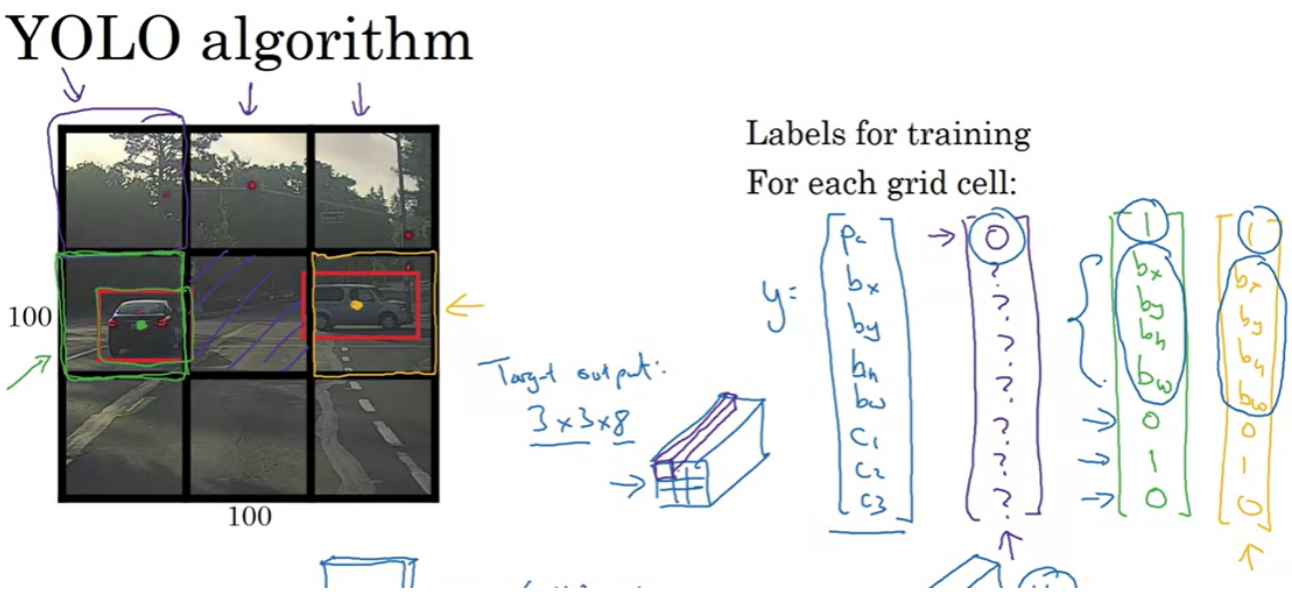
TO implement this in code,



We convert a fc layer to conv layer

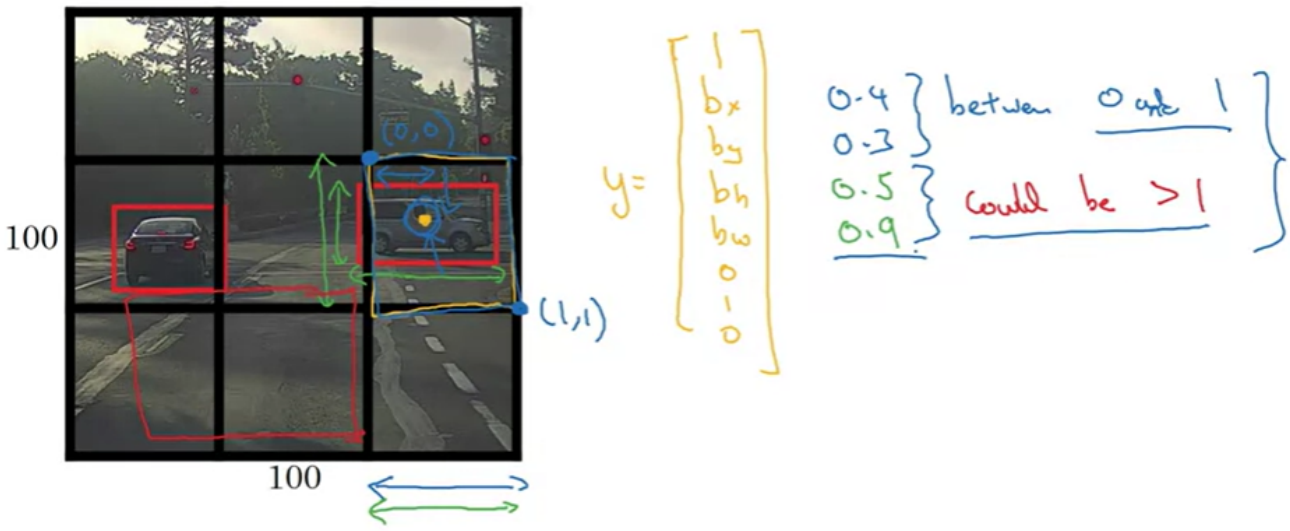


The blue region's output is the final blue top left cell. This helps to compress all the computation into one process. Thus individual sliding window's outputs can be found



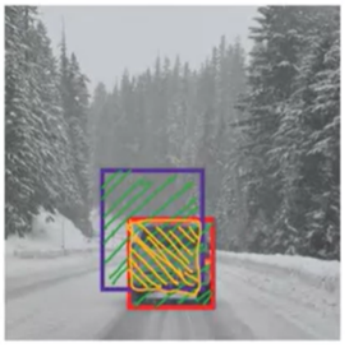
To improve the accuracy and speed, we draw a grid across the input images and perform object detection in each cell. This the output vectors are as shown above in right

Specify the bounding boxes



Higher the IoU, more accurate are the bounding boxes

Evaluating object localization

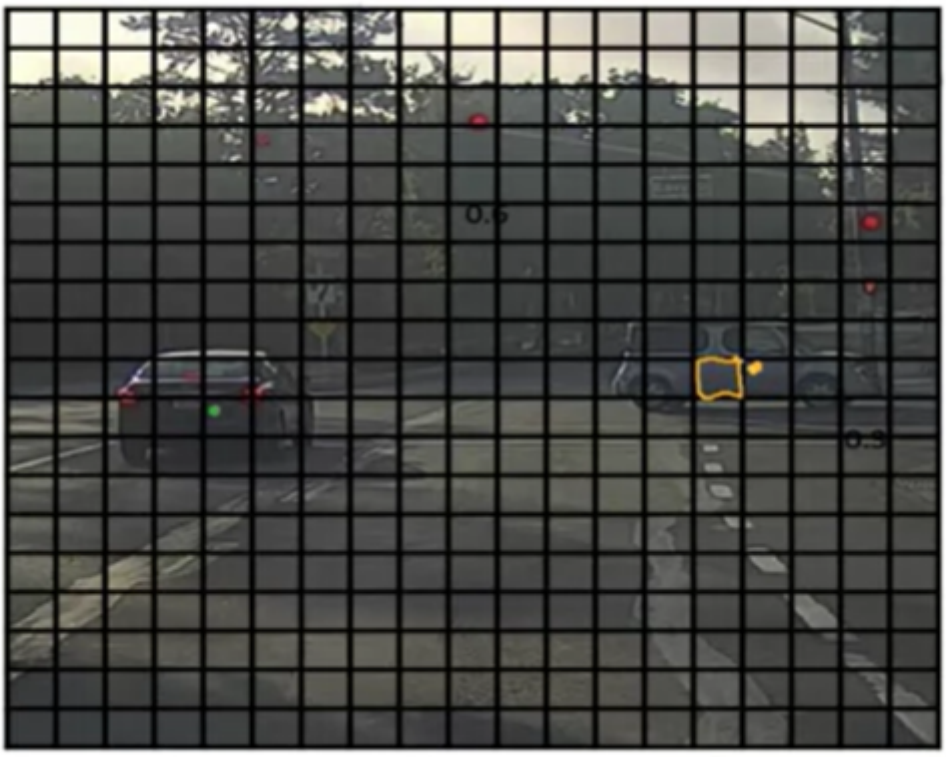


Intersection over Union (IoU)

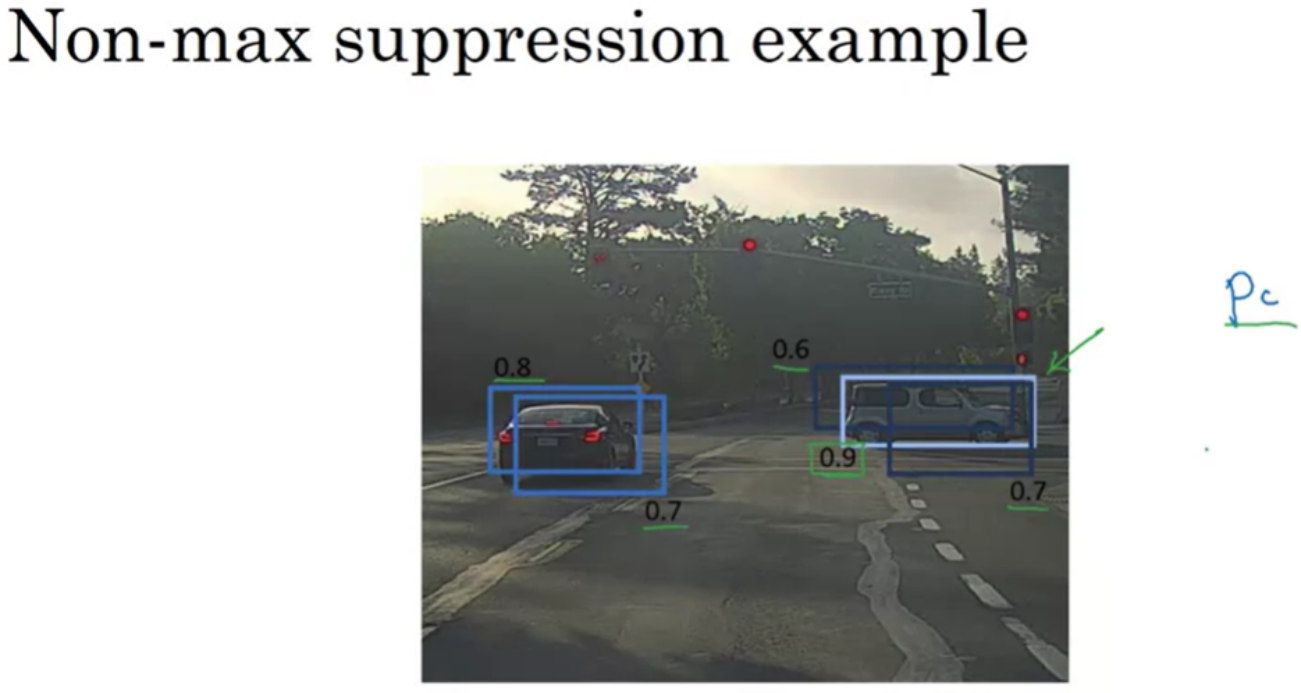
$$= \frac{\text{size of } \square}{\text{size of } \square}$$

"Correct" if IoU $\geq$ 0.5 ←

Non-max suppression



Here all the boxes having car might give multiple detection of car. So what nonmax suppression does is :



The prediction with highest IoU will be selected.

The algorithm to be followed is as follows :

19× 19

Each output prediction is: $\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \end{bmatrix}$

Discard all boxes with $p_c \leq 0.6$

While there are any remaining boxes:

- Pick the box with the largest p_c
  Output that as a prediction.
- Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step

Andrew Ng

Anchor Boxes

Overlapping objects:

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Anchor box 1:

Anchor box 2:

$y =$ $\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ p_c \\ b_x \\ \vdots \\ c_3 \end{bmatrix}$ Anchor box 1 / Anchor box 2

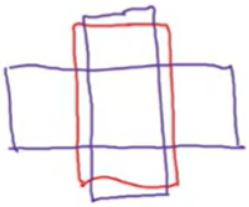
Anchor box algorithm

Previously:

Each object in training image is assigned to grid cell that contains that object's midpoint.

Output y:

$3 \times 3 \times 8$



With two anchor boxes:

Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

(grid cell, anchor box)

Output y:

$3 \times 3 \times 16$

$3 \times 3 \times 2 \times 8$

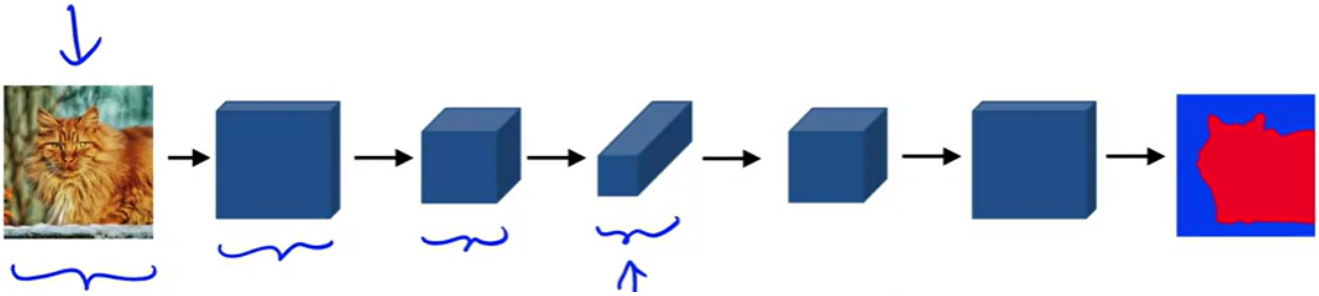
Outputting the non-max supressed outputs

- For each grid call, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class (pedestrian, car, motorcycle) use non-max suppression to generate final predictions.

Semantic Segmentation

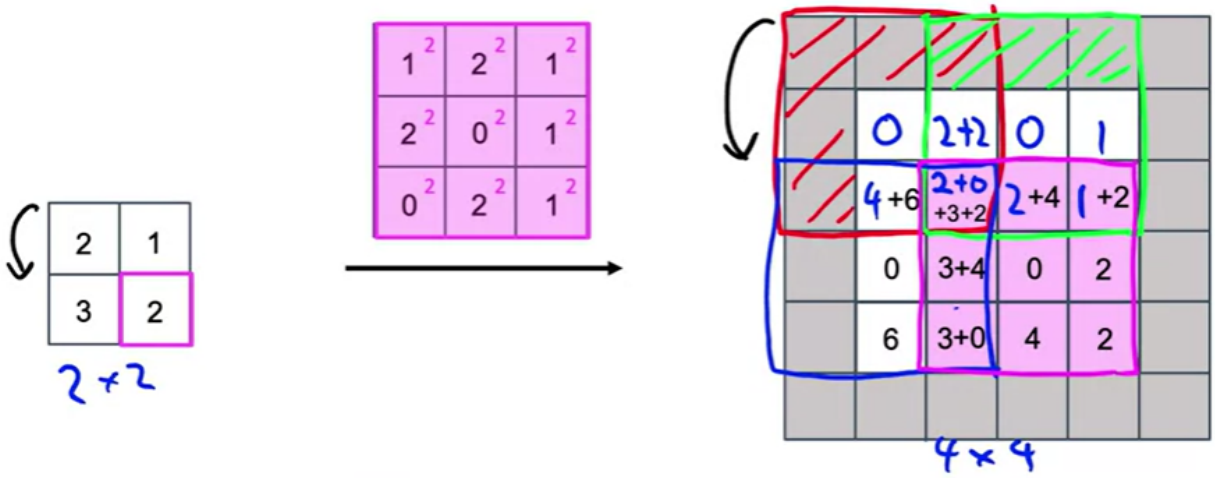
Here we label various pixel regions with different numbers. For e.g. car as 1 , road as 2, sky as 3. This can help to identify for eg. drivable road for self driving cars

Algorithm :



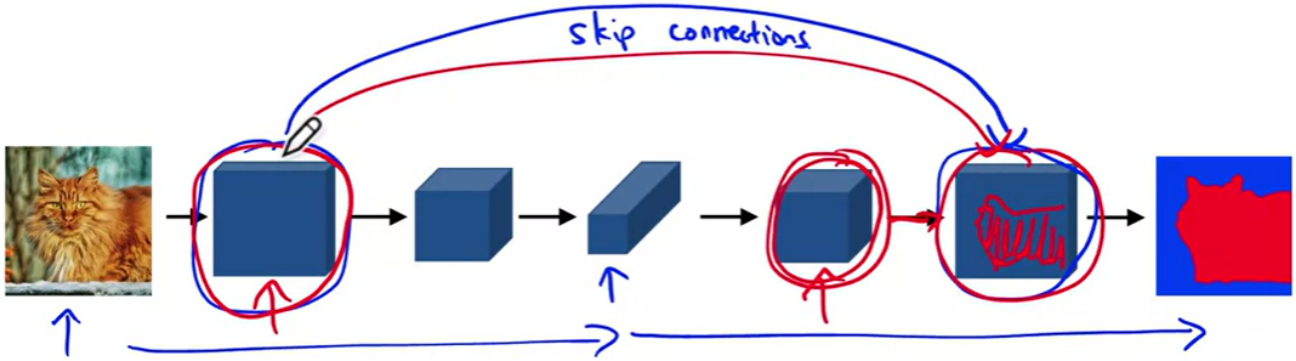
To perform the blowup in above image, we perform transpose convolution

Transpose Convolution

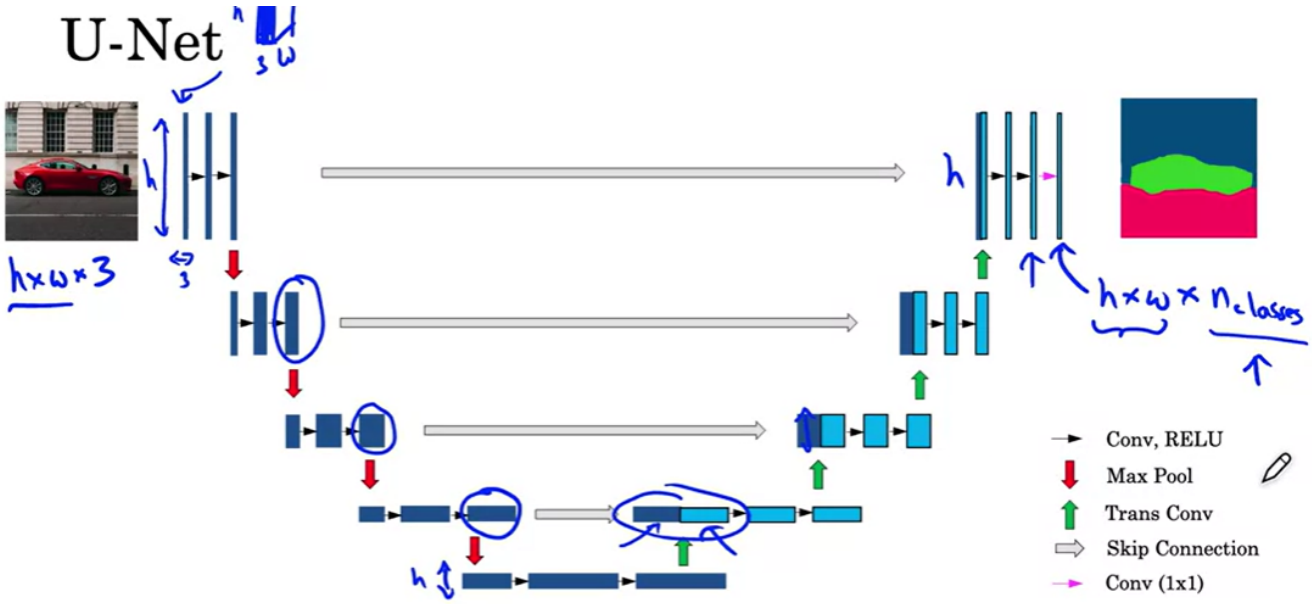


filter $f \times f = 3 \times 3$ padding $p = 1$ stride $s = 2$

Deep Learning for Semantic Segmentation



U-Net



Face Recognition

Face verification vs. face recognition

→ Verification 1:1 99%

- Input image, name/ID
- Output whether the input image is that of the claimed person 99.9

→ Recognition 1:K

- Has a database of K persons K=100 ←
- Get an input image
- Output ID if the image is any of the K persons (or "not recognized")

One-Shot Learning

In face recognition we have a very less training set. thus we calculate degree of difference between feeded image and test image, if degree is greater than tau then we say its not the same person

Learning a "similarity" function

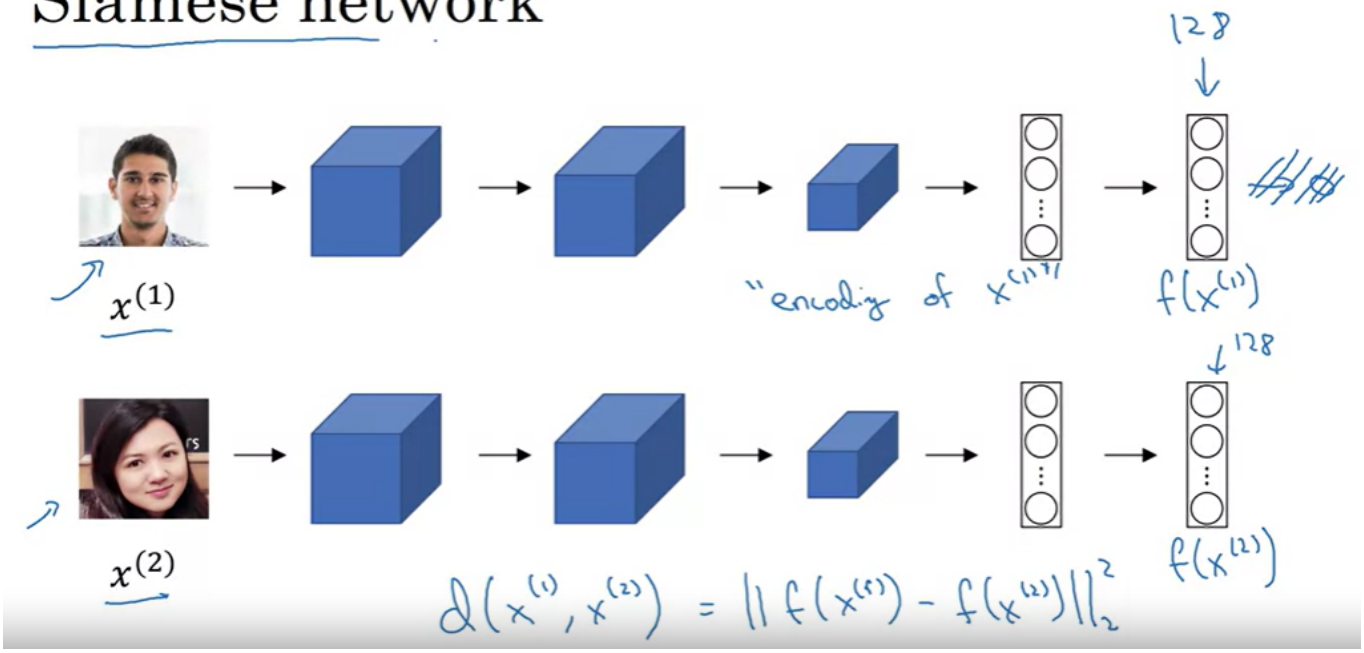
→ d(img1,img2) = degree of difference between images

If d(img1,img2) $\leq \tau$ "same"

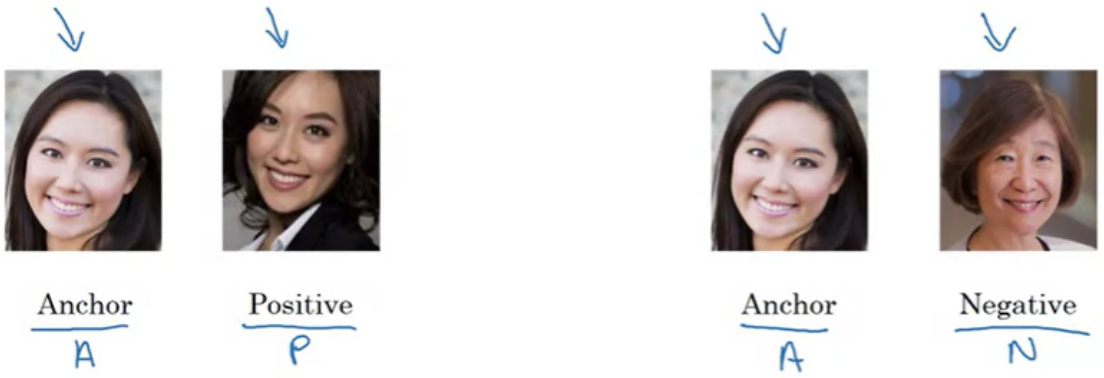
$> \tau$ "different"

} Verification.

Siamese network



Triplet Loss



Want: $\underbrace{\|f(A)-f(P)\|^2}_{d(A,P)} + \alpha \leq \underbrace{\|f(A)-f(N)\|^2}_{d(A,N)}$

Loss function

Given 3 images A, P, N:

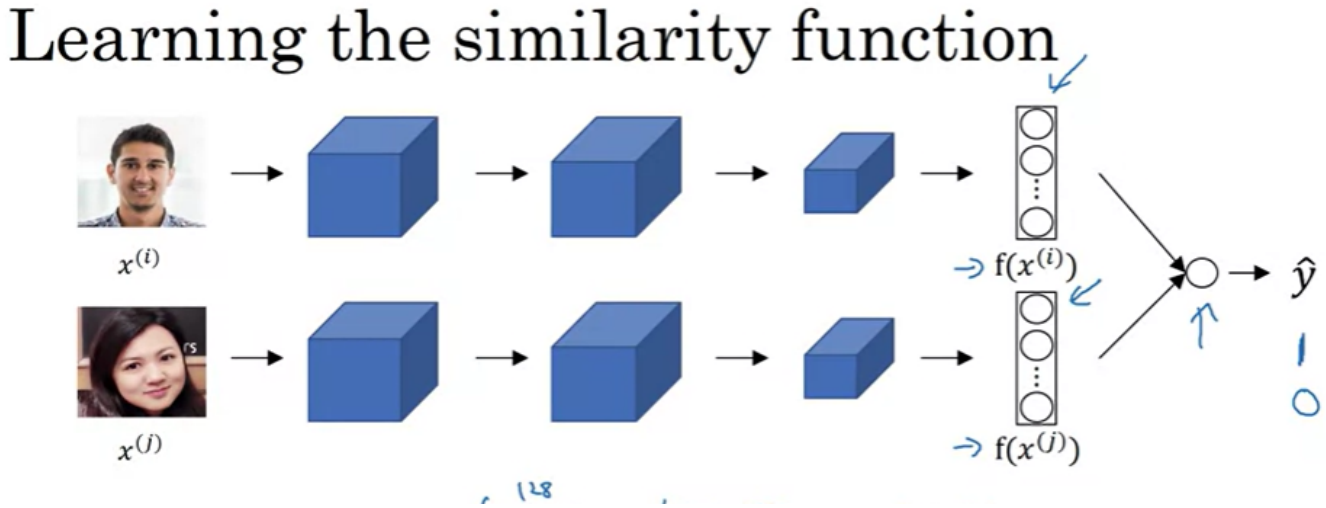
$$\mathcal{L}(A,P,N) = \max\left(\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha, 0\right)$$

$$J = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$$

A, P

Training set: 10k pictures of 1k persons

Choose hard A,P,N triplets such that it shard to train on them , otherwise if those triplets are chosen randomly then the condition will be easily satisfied and gradient descent wont do much of a difference



$$y_{hat} = \sigma\left(\sum_{k=1}^{128} w_k |f(x^{(i)})_k - f(x^{(j)})_k| + b\right)$$

Neural Style Transfer

Cost Function

Neural style transfer cost function

Content C

Style S

Generated image G

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

Find the generated image G

1. Initiate G randomly

   G: $100 \times 100 \times 3$ (RGB)

2. Use gradient descent to minimize $J(G)$

$$G := G - \frac{\partial}{\partial G} J(G)$$

Content cost function

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

- Say you use hidden layer l to compute content cost.
- Use pre-trained ConvNet. (E.g., VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

$$J_{content}(C, G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

Style matrix

Let $a^{[l]}_{i,j,k}$ = activation at (i, j, k). $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{ijk} \, a^{[l](S)}_{ijk'}$$

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](G)}_{ijk} \, a^{[l](G)}_{ijk}$$

"Gram matrix"

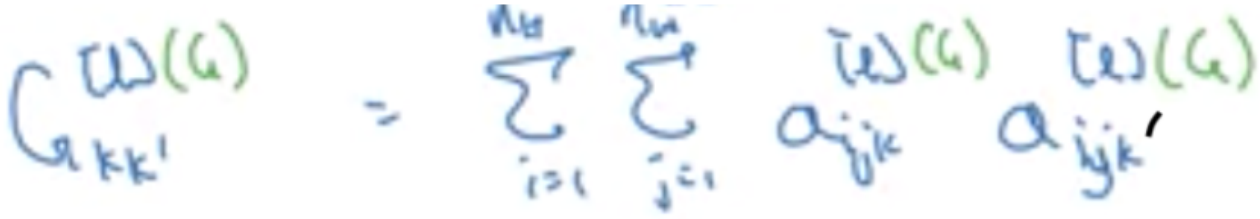
$$J^{[l]}_{style}(S, G) = \left\| G^{[l](S)} - G^{[l](G)} \right\|_F^2 = \frac{1}{2n_H n_W} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'} \right)^2$$

Style cost function

$$J^{[l]}_{style}(S, G) = \frac{1}{\left(2n_H^{[l]} n_W^{[l]} n_C^{[l]}\right)^2} \sum_k \sum_{k'} (G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'})$$

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S, G)$$

Please note that in the next video at around 8:50 when Andrew wrote down the second formula of matrix, the second factor of the multiplication should be on index k' (k prime), not k (otherwise it does not calculate any kind of covariance). This is shown in black below.



So the formula should be:

$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a^{[l](G)}_{i,j,k} a^{[l](G)}_{i,j,k'}.$

Also at 11:08 the style cost function formula should be the squared difference:

$$(G_S - G_G)^2$$

instead of just the difference:

$$(G_S - G_G).$$

The style cost function should be:

$$J^{[l]}_{style}(S,G) = \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}\right)^2$$