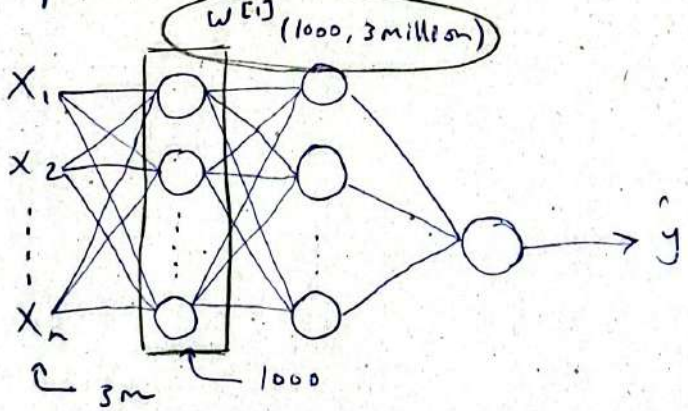
Course 4 → CNNs

Foundations of convolutional nueral net works :-

* When we train a network on 64 or 32x32 images this goes fine because if there is a 100 hidden units in the first hidden layer that makes the weights for this layer (100, 32*32*3) = (100, 3072) which are 30 thousand parameters which are actually not that bad compared to a high resolution image; say 1000*1000; assuming that we're gonna have 1000 hidden units then the shape of W will be (1000, 1000*1000*3) = (1000, 3million) resulting in 3billion parameters which is very expesive in matters of computations and time.



→ To solve this problem we'll use the convolution operation

Edge detection

* in the first course when we were trying to understand what's going on under the hood of the nueral network we said that in the first layers; the nueral network is sort of detecting edges (Lines); maybe vertical or horizontal Lines in the image but how does the nural network detect those edges?

Vertical edge detection

→ Let's assume that we have a 6x6 grayscale image

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

Convolution

*

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

=

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

→ We've gonna take this 6x6 image and Convolve it with a 3x3 filter (kernel), the result is a 4x4 matrix; here's how we can compute it:-

* We'll take a 3x3 window from our image and element wise multiply it with our filter as follows:-

3	0	1
1	5	8
2	7	2

*

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

$= 3*1 + 1*1 + 2*1 + 0*0 + 5*0 + 7*0 + 1*-1 + 8*-1 + 2*-1 = -5$

and so → -5 is our first element in the 4x4 matrix.

→ We shift the 3x3 window one step to the right and continue.

→ After we reach the end to the right ~~we start from the beginning~~ we shift one step down and start from the left again.

→ We repeat the process until we reach our final window

1	7	8
6	2	8
2	3	9

*

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

$= 1 + 6 + 2 + 0 + 0 + 0 - 8 - 8 - 9$

$= -16$

* So a 6x6 matrix convolved with a 3x3 filter gives us a 4x4 matrix.

* so how is this convolutions detecting images?

↳ if we take another example a 6×6 image ~~sketched only 2~~ and convolve it by a 3×3 filter we can see why

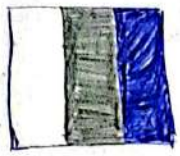
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

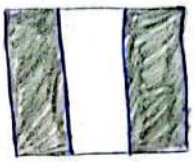
*

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

=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0





so as drawn in these images; the convolution process done by convolving the 3×3 filter with the image that had two colored regions and one edge between them now has the edge highlighted so it's basically like image processing.

* now; to why the region in the middle in the 4×4 image is thick? it's because the image that we used which is 6×6 is very small but when using larger images such as 1000 × 1000 images; the filter does a good job on them.

* one intuition to take away from edge detection:-

↳ a vertical edge is the output of convolving images with filters ~~to do brighter region~~.

Filter

↳ a lot of ideas in the rise of deep learning have been introduced for the values used in the filters; if you're using the filter for vertical edge detection then our previous filter might be suitable but if we're trying to detect something else such as horizontal edges then using another filter might be a better idea

This is a filter that focuses on horizontal edges rather than vertical ←

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

↳ other ideas were introduced regarding the values in the filter itself such as:

1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter

* We can also consider the values in the filter as a learnable parameter so instead of hardcoding the values in the filter we can consider them to be learnable weights

→ Instead of using values that were predetermined by computer vision researchers; maybe we can set these values to learnable weights

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

→ Back prob allows the filter to detect more than just vertical or horizontal edges (i.e learnable parameters case).

Padding •

→ When we're convolving images with the filters we lose some information from the Border pixels since the filter sees them less, especially the Corner pixels as the filter only sees them once.

*one way to deal with this loss of information is to use Padding; padding is the process of adding some pixels all around the pixels of the Borders of the image.

0	0	0	0
0	3	-2	
0	1	0	
0	2	5	
0			

→ thanks to the padded area now the 3×3 filter processes the pixel with the value of 3 four times instead of just once.

*Remember that the goal of Convolutions is to detect images in the picture not shrink the information from the images so it doesn't matter if the output of the Convolution is the same size of the image.

Types of padding

→ The two main types of padding are valid padding and same padding; in the valid Convolution we don't pad

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

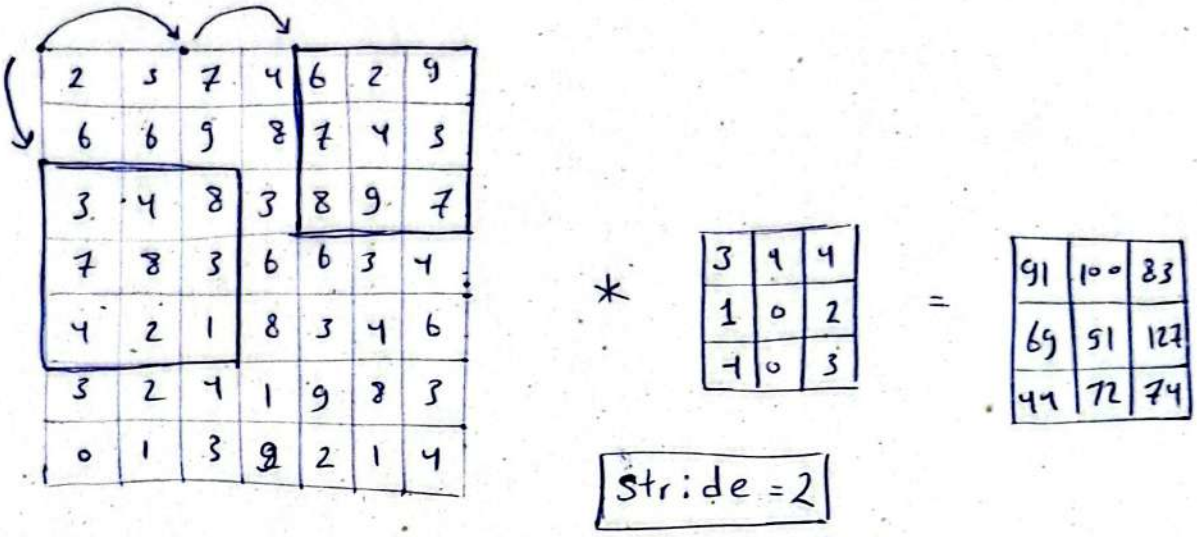
$6 \times 6 \quad * \quad 3 \times 3 \longrightarrow 4 \times 4$

Same: pad so that the output is the same size as the input

Same: (for $p=1$) $\rightarrow n+2p-f+1 \times n+2p-f+1$, $n+2p-f+1 = n$

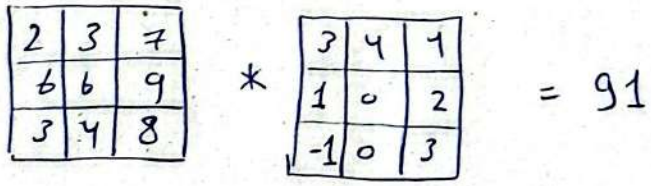
p is the padding size; f is almost always odd. $\hookrightarrow p = \frac{f-1}{2}$

Strided Convolutions

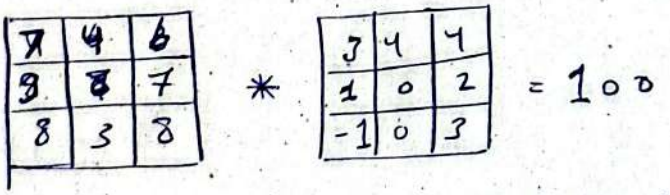


Stride = 2

→ Striding convolutions is the process of skipping rows and columns; specifically s number of rows and columns when we're convolving the input with the filter such as:

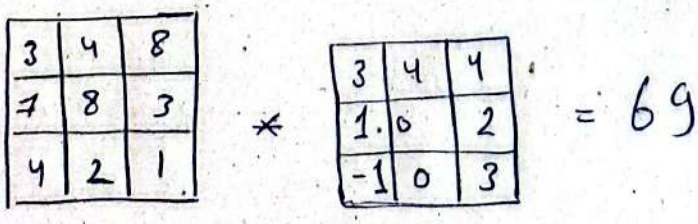


now skip 2 columns →



and so on until we hit the border

↳ now we stride 2 rows down :-



and so on until we're done.

→ in the previous example we convolved a 7×7 image with a 3×3 filter with stride = 2 and got a 3×3 output.

* The formula for this is:-

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

↳ padding → p

↳ stride → s

↳ in our example we used $p=0$ and $s=2$

so the output shape should be $\frac{7+0-3}{2}+1 = \frac{4}{2}+1 = 3$

which is the correct shape that we got.

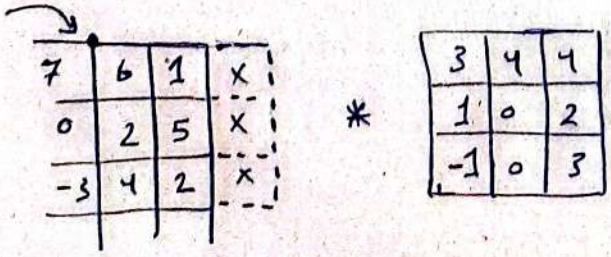
* if $\frac{n+2p-f}{s}+1$ is not an integer we can round it down to the nearest integer.

* if input image is 8×8 and we use same padding having a filter 3×3 and stride = 2 we get:-

$$p = \frac{f-1}{2} = \frac{3-1}{2} = 1 \quad , \quad s = 2$$

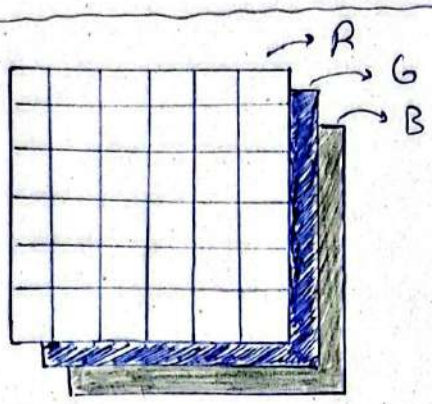
$$\frac{n+2p-f}{s}+1 = \frac{8+2-3}{2}+1 = \frac{7}{2}+1 = \lfloor 4.5 \rfloor = 4$$

→ The way rounding down is implemented in the stride is that if the final stride doesn't apply fully to the filter (isn't the same size), you remove the whole computation completly



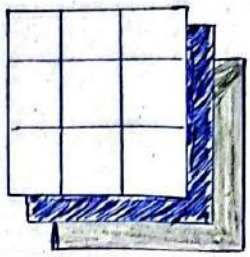
→ removed computation because the shape of the 2 matricies isn't the same.

Convolutions on RGB images:-



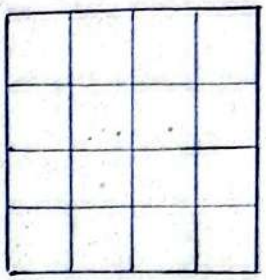
6 x 6 x 3

↳ 3-channel RGB image



3 x 3 x 3

↳ 3x3x3 filter



4 x 4 x 1

6 x 6 x 3

height width channels

3 x 3 x 3

height width channel

* The number of channels in the image needs to match the number of channels in the filter.

→ the way that we calculate the convolution of the image with the filter is that we apply the 3x3x3 to the first 3x3x3 cube in the image this results in the multiplication of 27 numbers, we add the results and this is our first output.

↳ we continue to do so on as we did befor even with padding and strides.

* having a 3x3x3 filter allows us to detect edges in the way that we want so if we want to detect edges in the red channels we can set the filter to be:

R

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

G

0	0	0
0	0	0
0	0	0

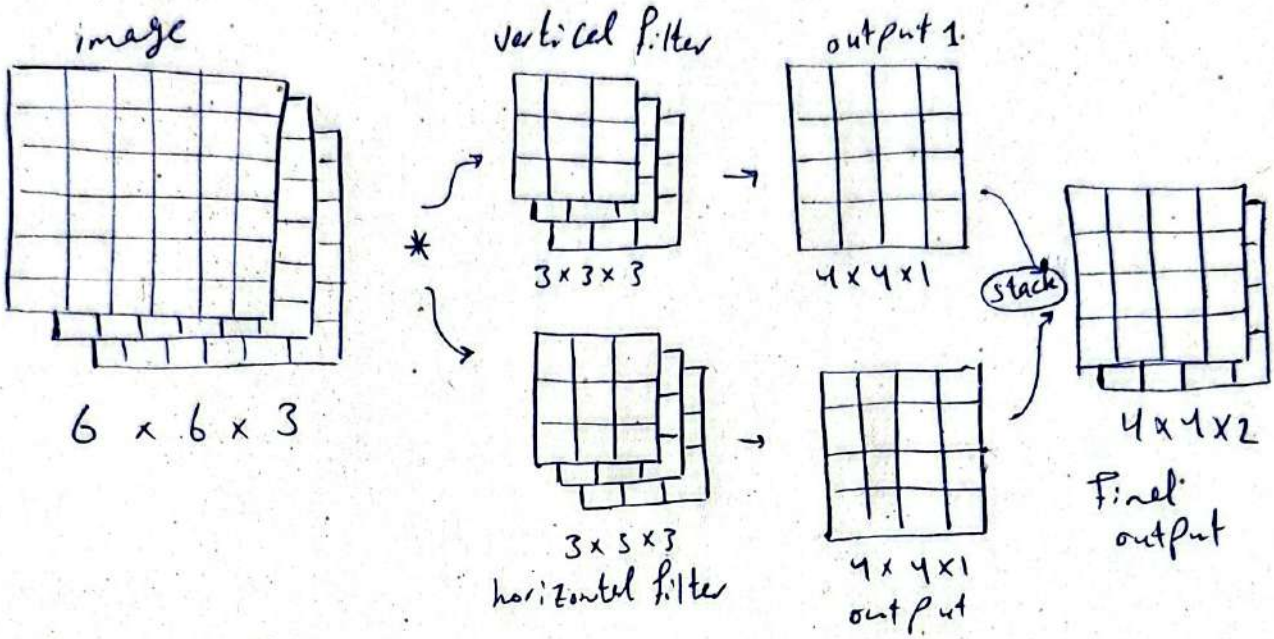
0	0	0
0	0	0
0	0	0

→ just like we wanted now the filter detects edges in the red channel only.

↳ those are the three channels

*using multiple filters:-

↳ now we've seen how to use filters on 3-chanel images so what if we want to use multiple filters; for example a filter for vertical edges and another filter for horizontal images?



→ The process is just as seen in the drawing, first we convolve the image with the first filter and then convolve it with the second filter. and in the end we stack the two 4×4×1 outputs to be 4×4×2.

summary:-

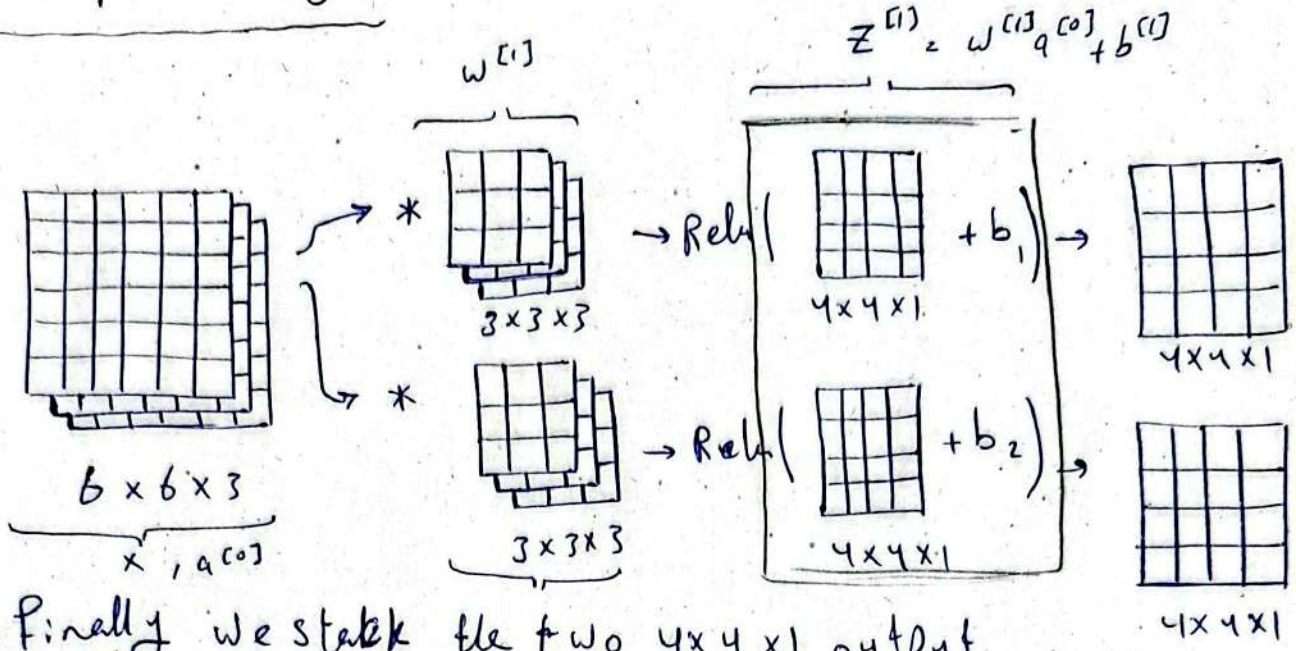
$$n \times n \times \boxed{n_c} \quad * \quad f \times f \times \boxed{n_c} \rightarrow n-f+1 \times n-f+1 \times \underline{n_c'}$$

↳ number of filters used

↳ assuming $s=1$, $p=0$

* in the past example we used two filters to detect 2 features, we can add as much filters as we want to detect as features as we want.

Example of a layer:-



↳ Finally we stack the two 4x4x1 output to make the 4x4x2 output of the whole layer.

* So just like neural networks; we take some input multiply it with weights (filters) and add some bias and pass it through some non-linear function such as Relu and after stacking the two filter outputs now we have what we used to call $a^{[1]}$

$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$
($a^{[0]}$ = x)

$$a^{[1]} = g^{[1]}(z^{[1]})$$
↳ probably Relu.

* In the previous example the output is $\boxed{4 \times 4 \times 2}$ because we only had 2 filters but if we had 10 filters then the output would have been $\boxed{4 \times 4 \times 10}$ so the channels of the output depend on the number of filters we use.

* no matter how big the input image is; the number of parameters in a CNN layer depends only on the size of the filter and the number of filters.

* If Layer L is a Convolutional layer :-

→ $f^{[l]}$ = filter size
→ $p^{[l]}$ = padding
→ $s^{[l]}$ = stride
→ $n_c^{[l]}$ = number of filters

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

output : $n_H^{[L]} \times n_w^{[L]} \times n_c^{[L]}$

* $n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$

* $n_w^{[l]} = \left\lfloor \frac{n_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$

Each filter :-

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

Activations :-

$a^{[l]} \rightarrow n_H^{[l]}, n_w^{[l]}, n_c^{[l]}$

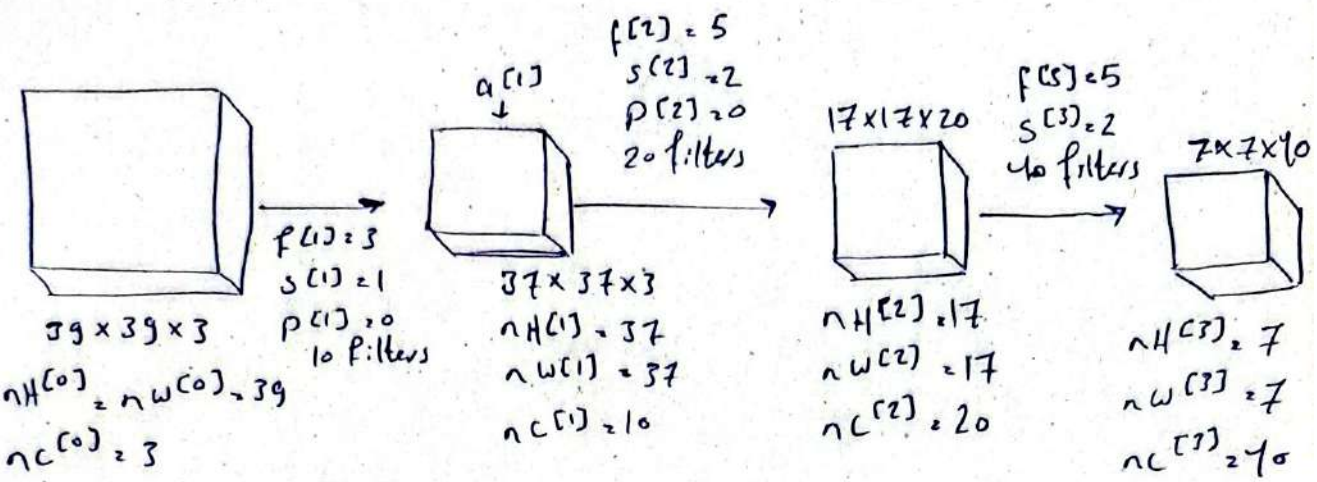
$A^{[l]} = m \times n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

weights :-

$w \rightarrow f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
↳ number of filters in layer l.

$b \rightarrow n_c^{[l]}$
↳ number of filters in layer l.

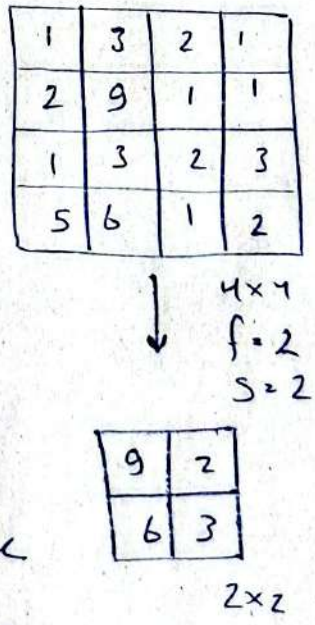
Example ConvNets:



→ now what's commonly done in the next step is to flatten the final 7x7x40 output that we have to 1960 elements and after that we can have fully connected layers or dropout or whatever we want to do befor feeding the output to the final (segmoide or soft max) function for classification.

Pooling Layers:

→ IN this Max pooling we started with a filter size 2 and stride = 2 but instead of convolving the matrix with the filter Like we did with convolutional layers here we just take the max of the Filtered Area (the window); ~~some~~ the hyper parameters of the pooling layer are, f (filter size) and stride.

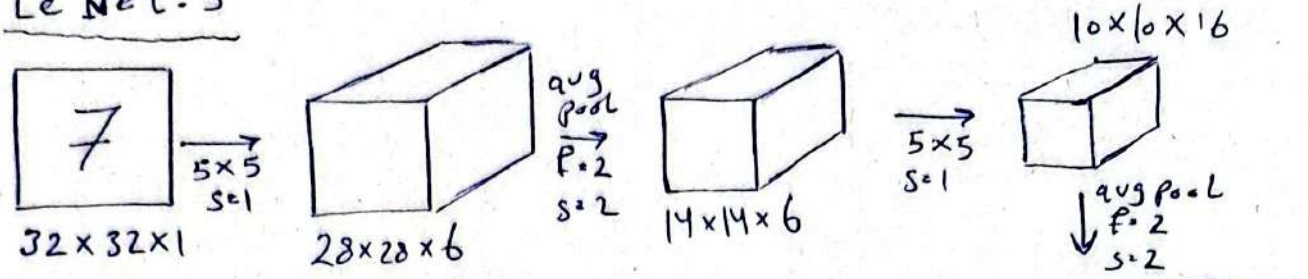


→ no parameters to Learn

$$\left\lfloor \frac{nH - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{nw - f}{s} + 1 \right\rfloor \times nC$$

Week 2:- Deep Convolutional models

Le Net-5



→ the LeNet-5 was a nueral network used to classify hand written digits, it's made of 2 Convolutional layers followed by ~~max~~ pooling layers; back then, it was more common to use Averagepooling instead of Maxpooling.

→ the Lenet-5 is considered a small nueralnet since it has ≃ 60k parameters only.

Conv-1 → 6 filters (5×5) = (25 + bias) × 6 = 156 params

Conv-2 → 16 filters (5×5) = (25 + bias) × 16 = 416 2416

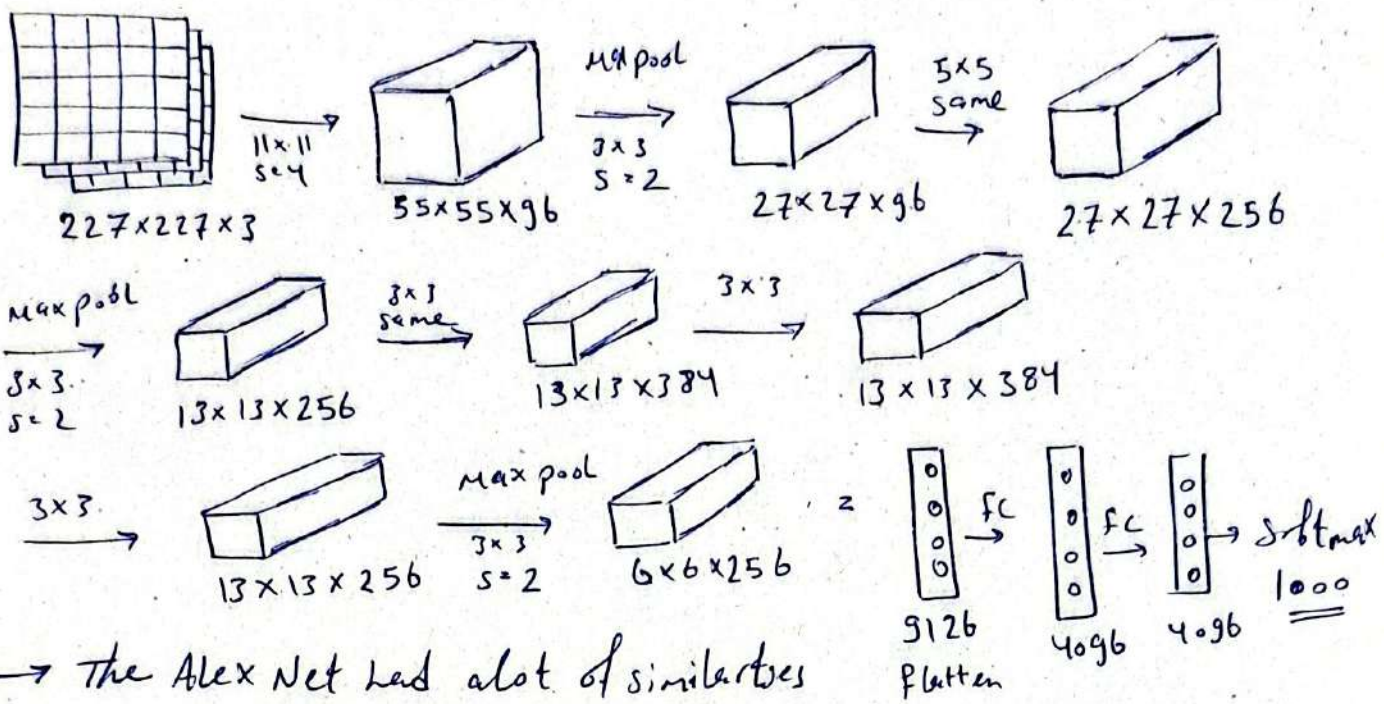
FC-1 → 400 inputs, 120 hidden units → 48,000 + 120 = 48120

FC-2 → 120 inputs, 84 hidden units → 10080 + 84 = 10164

softmax → 84 inputs, 10 units → 840 + 10 = 850

→ total parameters for LeNet-5 = 51706 ≃ 60k

AlexNet

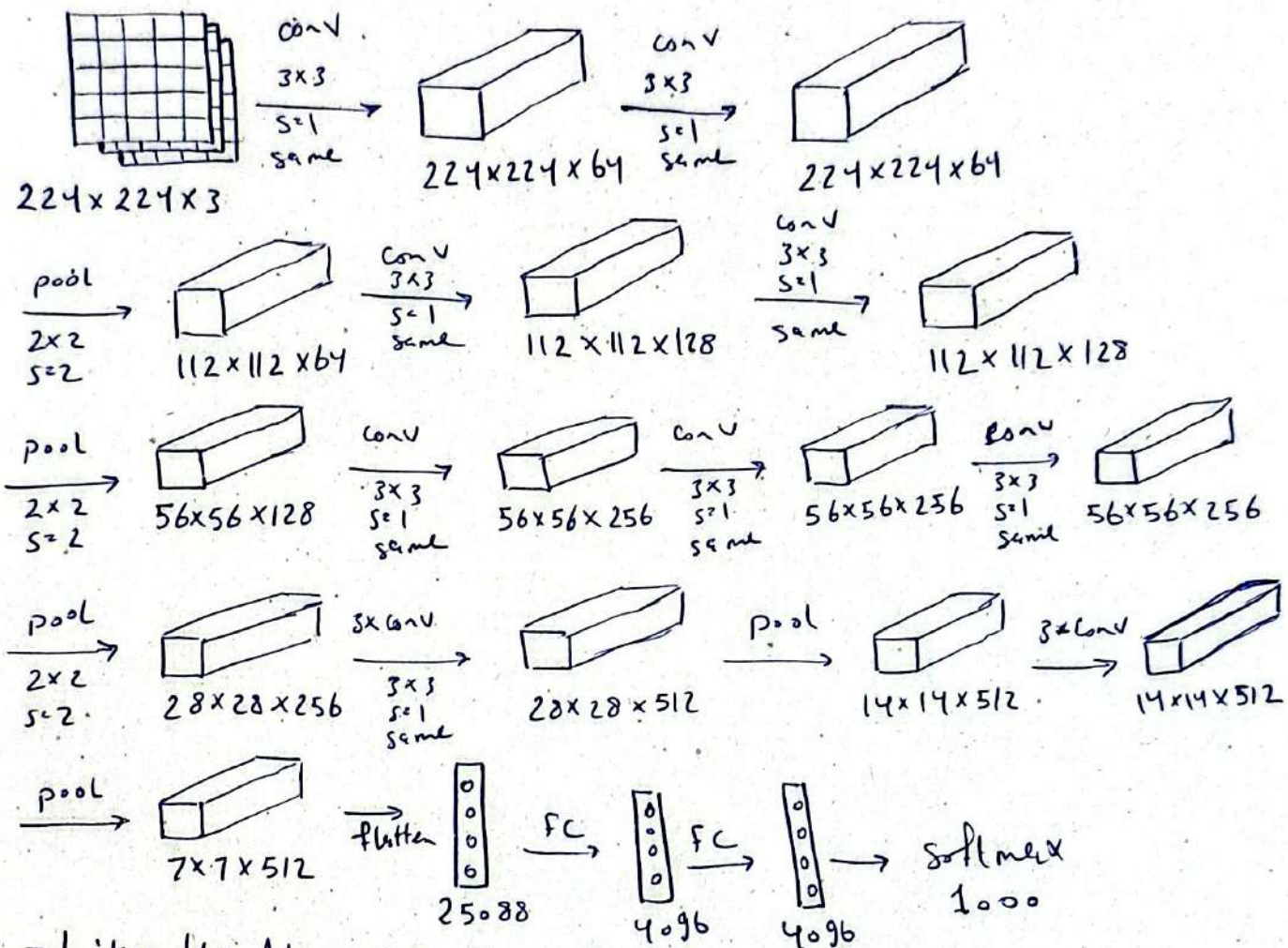


→ The AlexNet had alot of similarties to the LeNet-5 nueral net work but it was much bigger, the LeNet-5 had around 60k parameters, but the AlexNet has around 60 Million parameters whic is almost 1000 times bigger than the LeNet-5.

→ Another aspect why this was better than the LeNet-5 is that it uses ReLU (rectified Linear unit) activation function instead of the sigmoid function like in the LeNet-5 but back in 1998 when the paper for the LeNet-5 was written, ReLU was not used commonly yet.

VGG-16

→ In this nueral net the researchers decided to go with a much simpler network in their approach compared to something like the AlexNet; they used ConV layers with 3×3 Filters, stride = 1 and used same padding, in the Max pooling layers they used 2×2 Filters with stride = 2 which is very simple.



→ Like the AlexNet, this also is a pretty big nueral net with around 138 million parameters, it uses simpler Convolutions but is way deeper than the AlexNet.

→ it's called the VGG-16 because it has 16 layers.

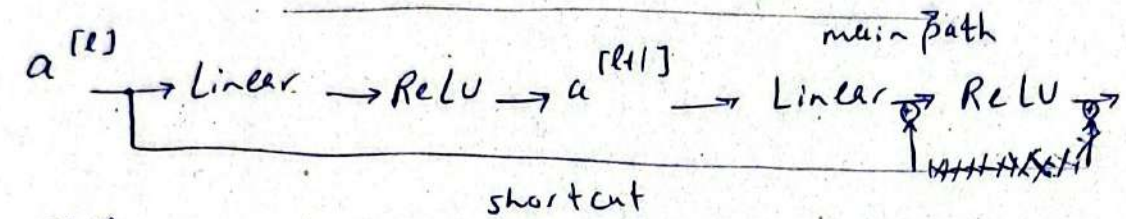
→ The main down side of this nueral net is that it is a pretty big nueral net even by modern standards.

ResNets:

→ very deep nueral net works are harder to train because of problems like vanishing and exploding gradients so in this case study we'll see how skip connections are implemented.

* skip connections allows us to take the activations from one layer and feed it to a layer much deeper in the nueralnet.

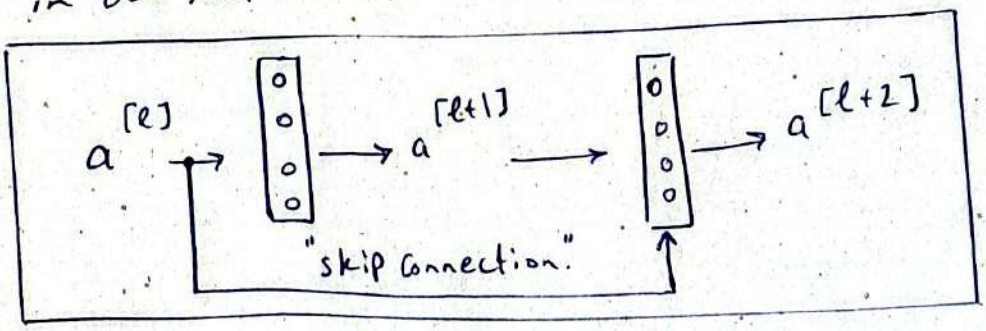
→ ResNets are built on something called a Residual block;



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

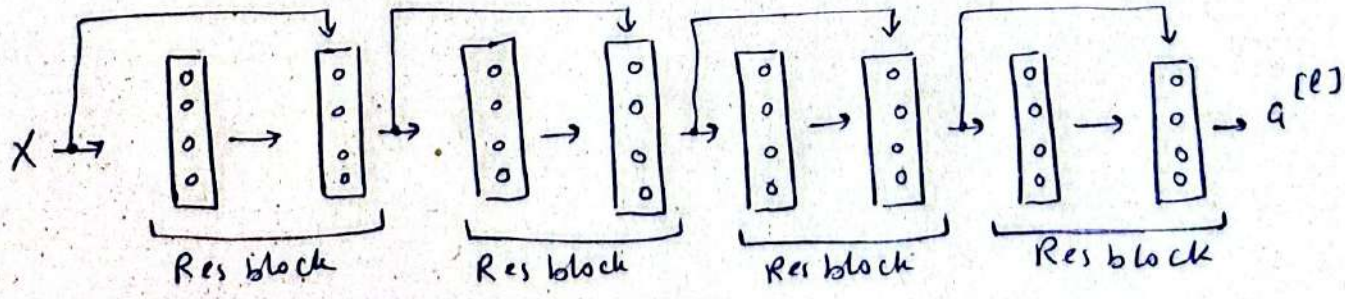
$$z^{[l+2]} = w^{[l+2]} a^{[l+1]} + b^{[l+2]} \rightarrow a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

* so what happened here is that we took $a^{[l]}$ and we directly fed it or added it to another activation much deeper in the network.

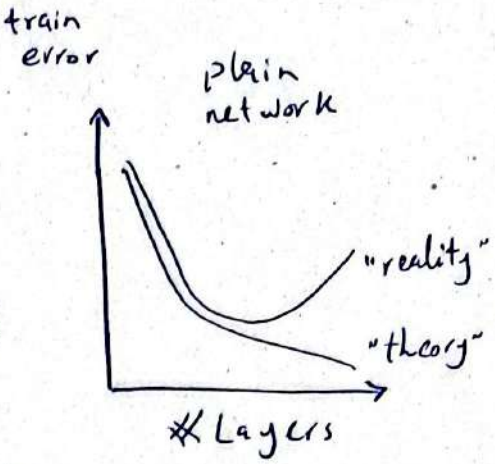


Residual block

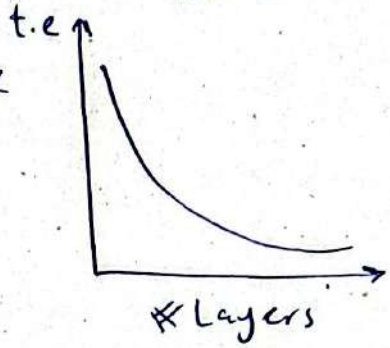
ResNet



* so in theory; training a neural network or like what the authors of the Res Nets paper called it; "plain network" (meaning it does not have any residual blocks); is supposed to have lower training error as we increase the number of layers but in reality; the training error actually increases.



→ while in plain networks the training error increases as you increase the number of layers; this does not happen when we're using Res Nets because Res Nets handle the problem of exploding and vanishing gradients much better than the plain neural networks.



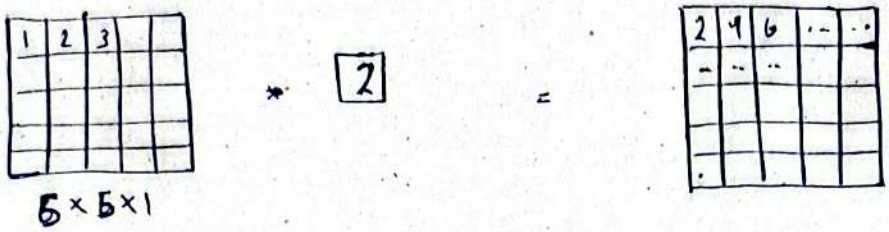
Why Res Nets work so well?

---

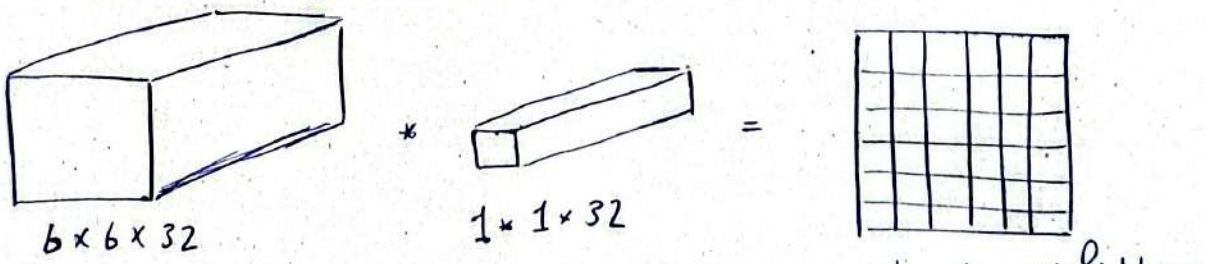
*complicated; it just works.

What does a 1×1 Convolution do?

→ if you think about it interms of a having a picture with only one channel then the 1×1 Convolution is only multiplying the pixel value with the weigh of the filter



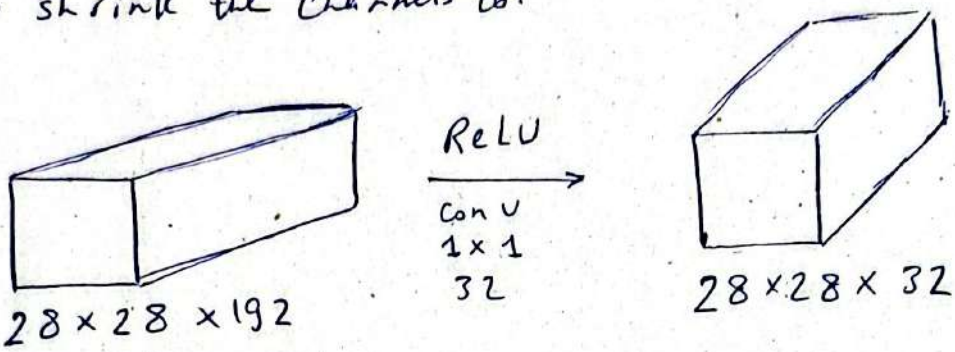
* In reality the 1×1 Convolutions are not used with images that have only 1 Channel but rather with matricies that are have larger number of channels



→ So actually in this case the value that we're computing is being computed across the whole channels of the 6×6×32 volume not just multiplication of two numbers and in a single step of computing the output volume ~~You can imagine it as a fully connected layer between 32 and 32 hidden units~~ You can imagine it as applying Relu to some values multipled by the weights

→ This is sometimes called "Net work in Net work".

* a useful use of the 1×1 Convolutions is that if you have for instance an input volume that has grown too big in number of channels and you want to decrease the number of channels then you can use 1×1 Convolutions with the number of filters that you want to shrink the channels to.



→ with each 1×1 filter having 192 channels as the input volume.

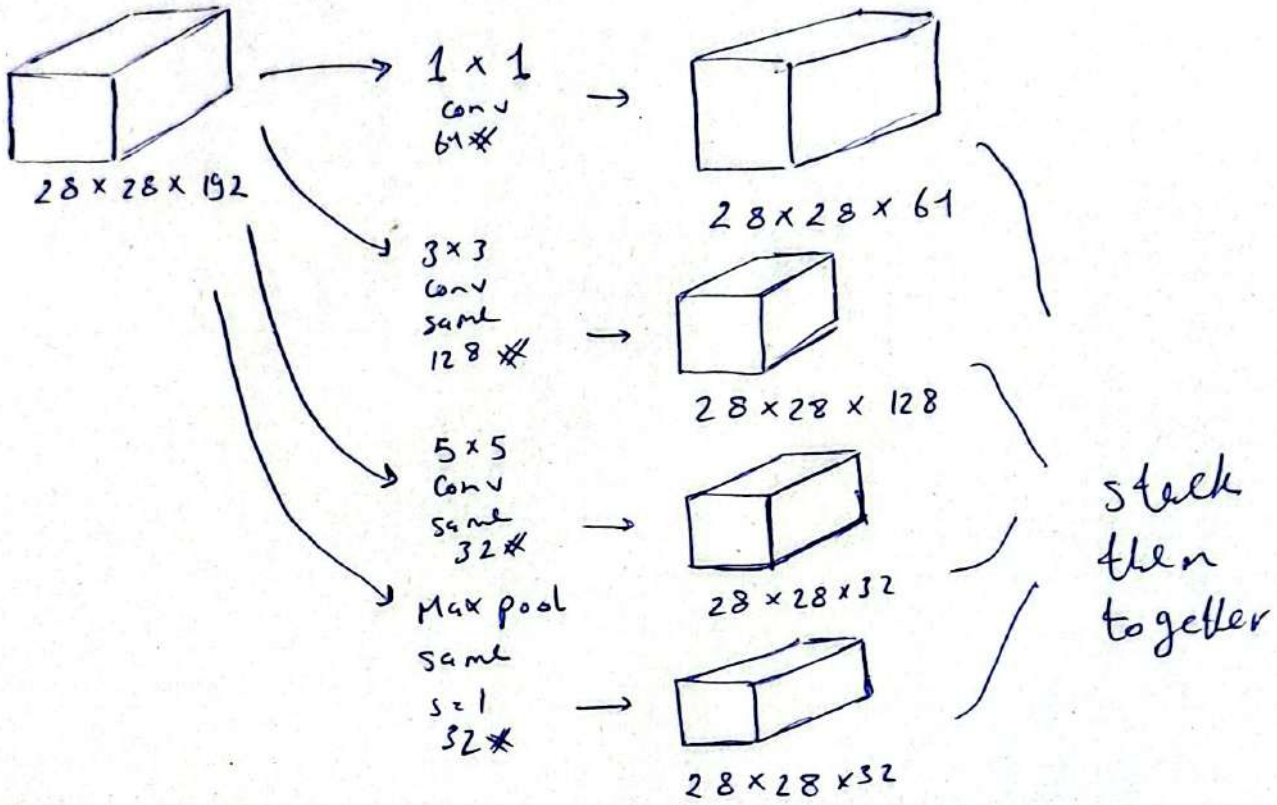
→ So the idea of having 1×1 Convolutions allows you to shrink the number of channels allowing you to save on computations without shrinking the actual height or width of the input volume.

→ even if we don't use the 1×1 Convolutions to shrink the number of channels it still plays a role as it applies Relu or non linearity to it's inputs so it doesn't have to be used only to shrink channels.

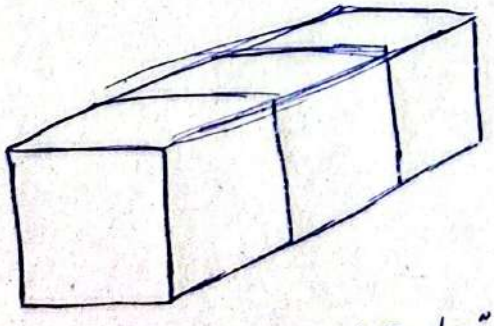
Inception network

→ the idea behind the inception neural net is that it allows you to do more than one convolution; even max pooling in the same computational process. so how does that exactly work?

- let's say you have a 28 × 28 × 192 input volume



- and then we have an output volume of the stacked volumes from each process and we can do that because we paid attention to the dimensions of each output volume



"stacked outputs"

final dimension

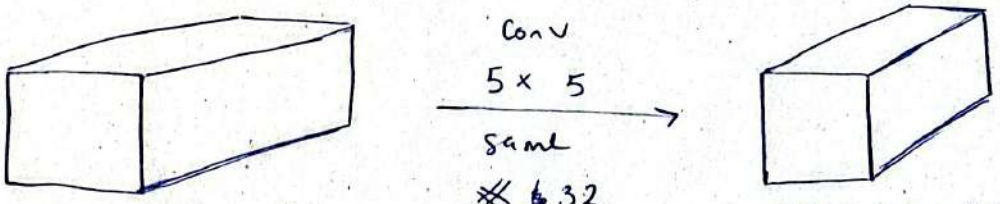
28 × 28 × 256

so instead of having to pick a single computation for each layer like a certain kernel size with a certain number of filters for a single Conv layer; you can do them all and have multiple convolutions an even Max pooling like we said in a single "Inception module" and the network will learn whatever parameters it wants to use for whatever combinations of these filter sizes that we have.

The problem with Inception layers :-

↳ of course as it's predicted the inception layer or module has a problem which is the computational cost.

- for just the 5x5 out block that we wanted to stack in the previous illustration the computational cost is :-



↳ 32 filters with each being 5x5x192 = 4800 values

these values will be multiplied by 28 x 28 x 32 = 25 088

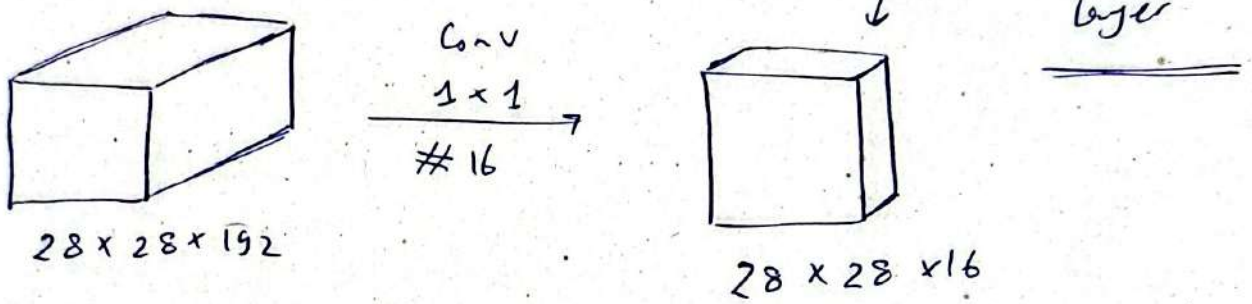
- so the full number of computations for this step will be 25088 * 5x5x32 ≈ 120 Million

number of computations for each output block

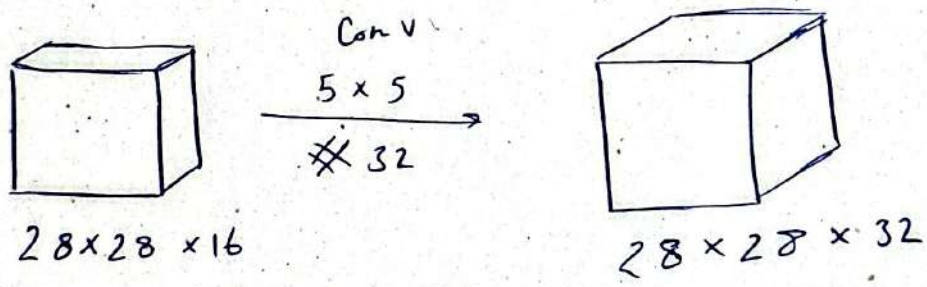
number of computations for each value in each filter

↳ while 120 million computations on a modern computer is possible; it is still very computationally expensive.

- Here comes the role of 1×1 convolutions as it can decrease the number of computations needed for this step by nearly a factor of 10; making them ≈ 12 million computations only instead of 120 million.
- First we use 1×1 convolutions to reduce the ~~then~~ volume of the input block



- Then we can use the 28×28×16 output volume and run the 5×5 convolution on it.



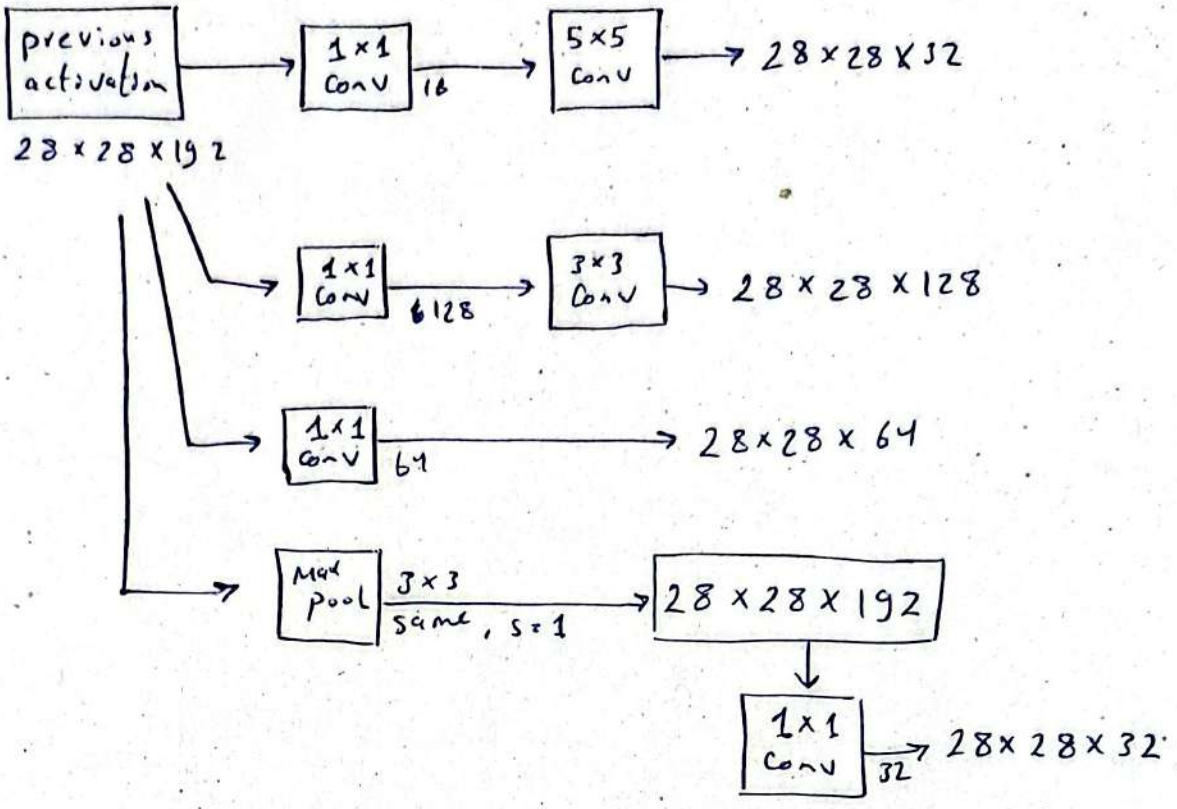
- Computational cost for the past step:-

→ Conv1 → 1×1×192 × 28×28×16 = 2.4 Million

→ Conv2 → 28×28×~~16~~32 × 5×5×16 = 10.0 Million

total ≈ 12 Million

Inception network



and then we concatinate all the output volumes resulting in a $28 \times 28 \times 256$ output volume.

↳ This was the inception module; what the inception network does is put alot of inception modules together.

- one famous inception network is the goolenet nural network developed by google research team.

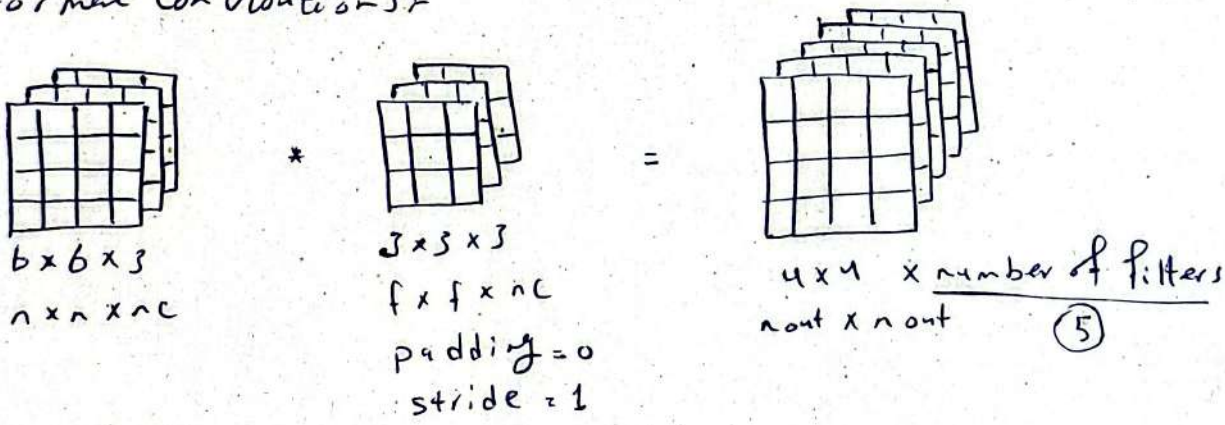
Mobile Net

↳ Mobile net is another foundational nueral network architecture used in Low Compute environments such as mobile phones.

↳ The previous nueral networks that we Learned about are actually Computationally expensive and if we want a nueral net to work on a Low Compute environment such as mobile phones then we need another nueral network architecture that can perform better.

∴ Normal Vs depth wise seprable Convlutions :-

↳ Normal Convlutions :-



(5)

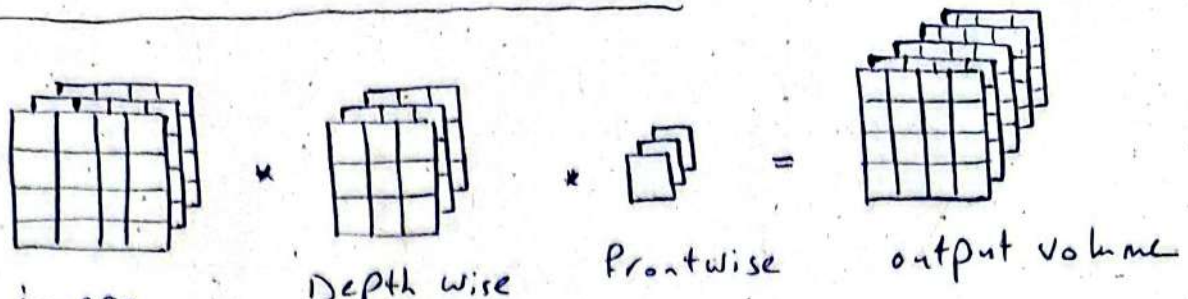
→ Computational Cost :-

Cost = # filter params x # filter positions x # of filters

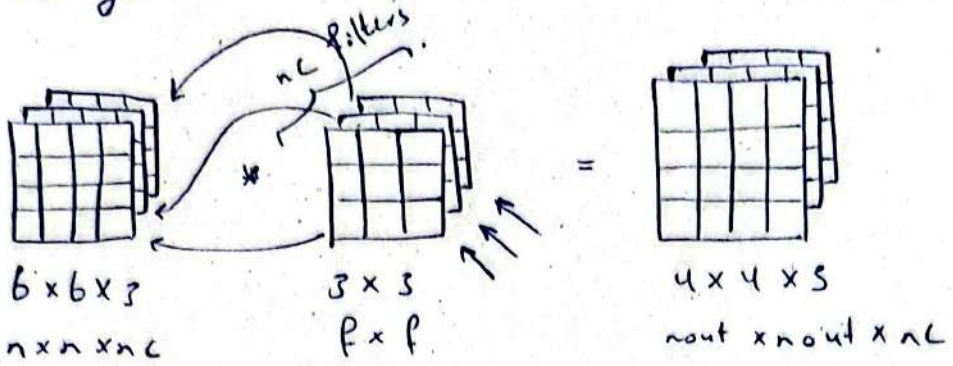
= 3 x 3 x 3 * 4 x 4 * 5

= 2160

Depth wise seprable convolutions :-



1)



The way that we apply these convolutions is that we take each channel of the filter and apply it to the corresponding channel in the input volume; not like we did in the normal convolutions where we applied the whole filter by the whole slice of the image.

↳ By convolving each channel of the filter with each channel of the image we get 3 three 2-dimensional matricies then we stack these together to get our output volume.

↳ in the previous process, we got each value of the output matrix by multiplying only 9 values from the filter by 9 values from the corresponding image channel, much less than we used to do in normal convolutions.

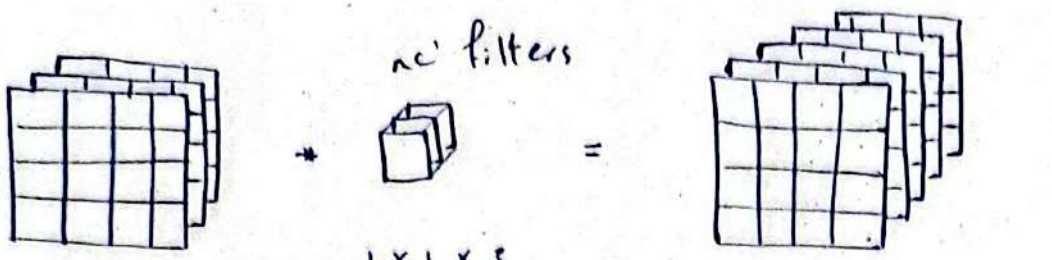
Cost = # filter params * # filter positions * # of filters

= 3 x 3 * 4 x 4 * 3

= 432

which is much less than normal convolutions

↳ We're not done yet as this was the depth wise part of the convolution only; now we still have to do the front wise convolution.



→ Now with this step we would do as we did with normal convolutions, we have a 4x4x3 input volume and to get the same output as the 4x4x5 normal convolution that we did before we have to convolve our input with 5 filters that are 1x1x3.

↳ Computational cost for this step:-

Cost = 1 x 1 x 3 (filter params) * 4 x 4 (filter positions) * 5 (number of filters)

= 240 multiplications

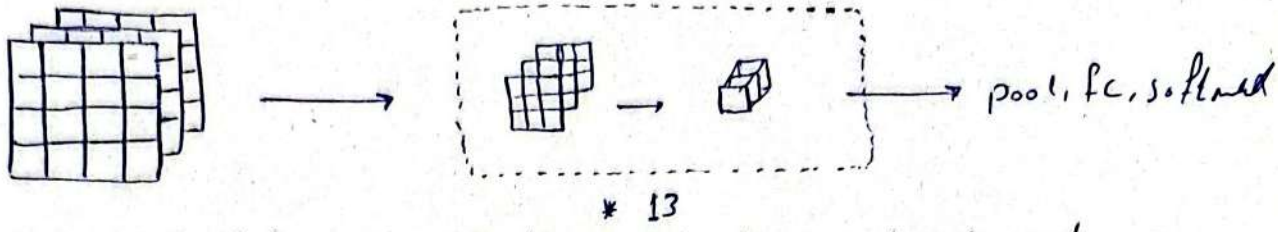
↳ now the total number of computations needed by this depth wise seprable convolution to get to the same output volume as the normal convolution is

= 432 + 240 = 672 multiplications.

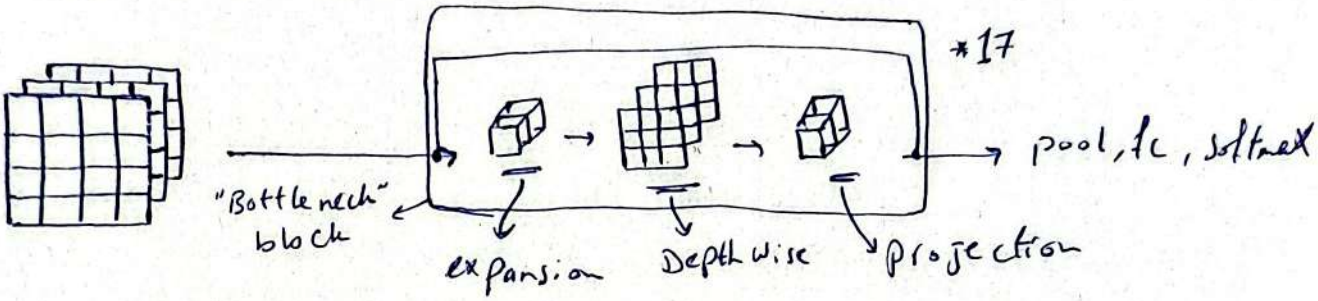
→ which is much less than the 2160 that was needed by the normal convolution.

→ the general formula for relative computation used to normal convolutions = $\frac{1}{nc} + \frac{1}{f^2} = \frac{1}{5} + \frac{1}{3^2} = 0.51$

MobileNet Architectures



→ In MobileNet version 1 they fed the input volume to 13 blocks of Depth wise seprable convolutions and then applied some pooling followed by some fully connected layers and finally a softmax function for classification.

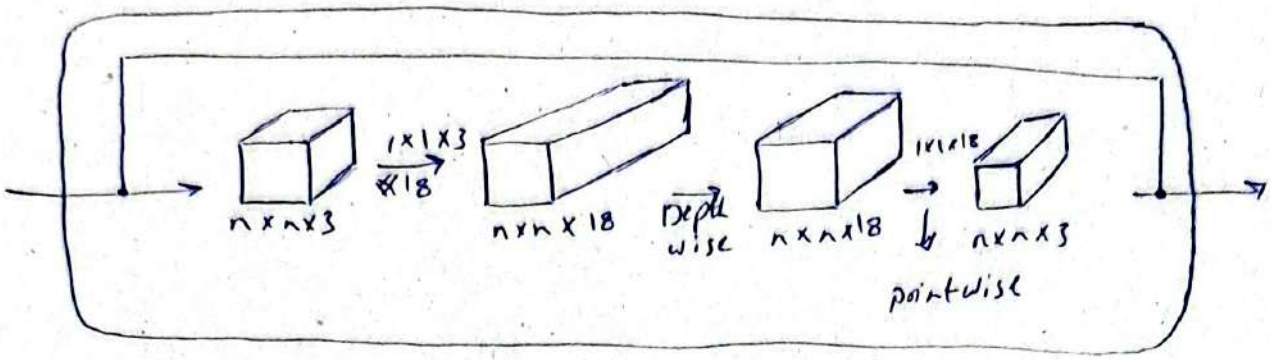


→ In MobileNet version 2 the applied the Consept of skip Connections aka (Residual Nets) to the input volume and Combined it with the Depth wise seprable convlutions; they first fed the input volume to an expansion layer then to the normal Depth wise layer followed by the front wise layer (here called projection layer).

→ Using the Residual Connection allows the gradients to probagate more efficiently.

→ Just like MobileNet v1; MobileNet v2 fed the input volume not to 1 but to 17 Consecutive Residual blocks and then did some pooling followed by fully connected layers then softmax.

MobileNet V2 bottle neck



→ so what happened in this Residual block or bottle neck block that we made is:-

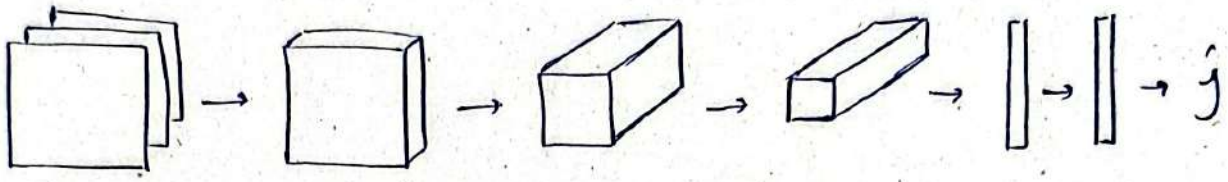
- we fed the input volume to convolution layer that uses 1x1x3 filters, specifically 18 of them and by using the 1x1x3 we maintained the height and width but with 18 filters we increased the number of channels to 18 hence why it's called an expansion layer.
- we then took the nxnx18 output volume and applied a Depth wise convolution to it which will maintain the same height and width. and also the same number of channels because it convolves each filter with the coressponding channel.
- we then fed the nxnx18 output volume that we got from the depthwise convlution and fed it to a point wise (front wise) convlutional layer, which is a typical convlution with filters of size 1x1xnC, specifically with 3 filters resulting in an output volume of nxnx3.

→ The expansion layer expands the information from the input volume which allows the model to learn better which is good over all for the performance.

→ But since we're doing all this in low compute environments we have to project the output volume down again to save up on computations.

EfficientNets

→ MobileNet v1 and v2 gave us a way to implement a nueral network that is more computationally efficient but there is a way to scale or computations either up if we have more computational resources or down if we're constrained by less resources; this is efficientNet.



→ Let's assume that we have a "Baseline nueral network" in which we have a image of resolution r and depth of the network d; and the layers have a certain width w.

∴ The three thing that you can do to scale things up or down are

↳ use a higher or lower resolution (r).

↳ Make the network more or less deep (d).

↳ Make the layers more or less wide (w).

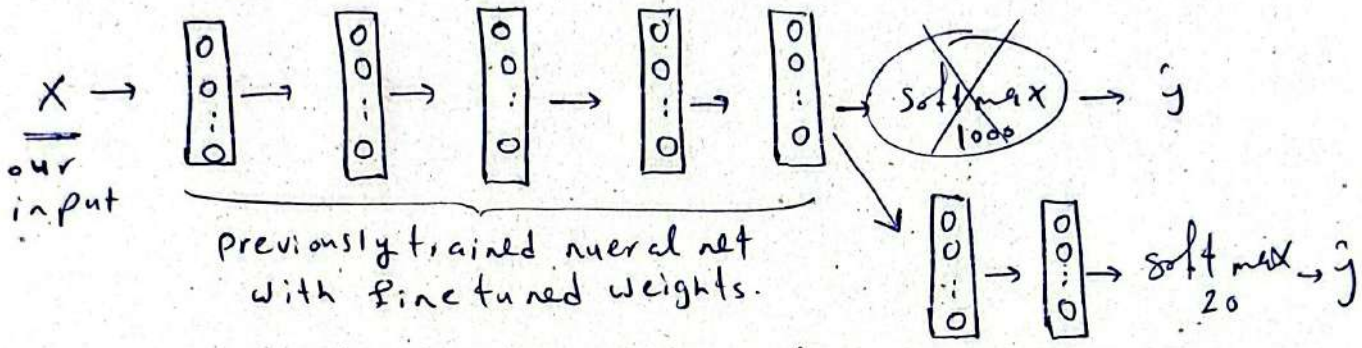
→ So the three things that we can control are resolution, depth and width of the network.

• Now the question is, if we want to scale up r, d and w what's the rate at which we scale them up?

↳ a good way to consider this is to look up some of the open source implementations of efficient Net, this will help us decide the rate at which we chose to scale up to.

Transfer Learning

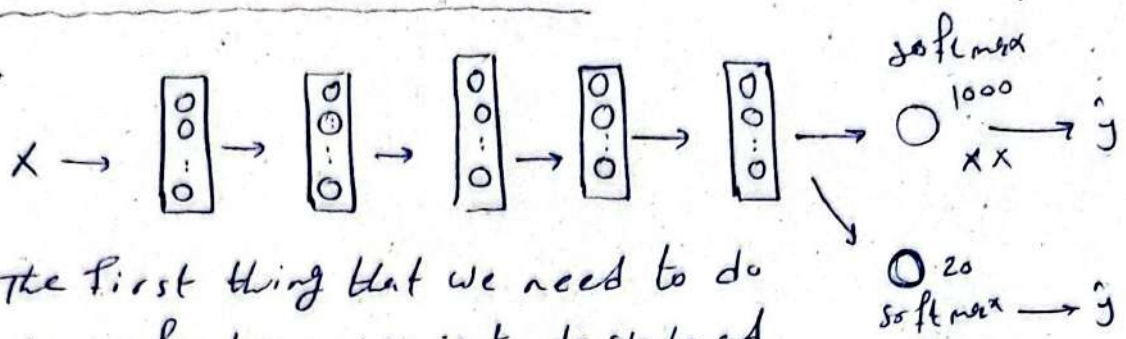
↳ Transfer learning is the process of transfering knowledge or information from a model that was previously trained and fine tuned to our model that we're trying to train, we do this by using the weights of these trained models as our intialization weights and of course this is much much better than randomly intializing the weights.



→ So it's just like getting rid of the output layer of a previously trained model and adding a couple of new layers so the model can adapt to the new data which are our data set and then we can have a new softmax with number of hidden units = our data set classes

→ Transfer learning also helps when we have a small data set so we don't have alot of training examples to train the model on, in this case we can just use the knowledge gained by a previously trained model and adapt this knowledge to our new data set by fine tuning the weight a bit on our data.

approaches to transfer learning



→ the first thing that we need to do in transfer Learning is to download a pretrained nueral network like image net, this pretrained nueral net will have it's own architecture and pretuned weights and based on the dataset in hand we'll do one of the following options.

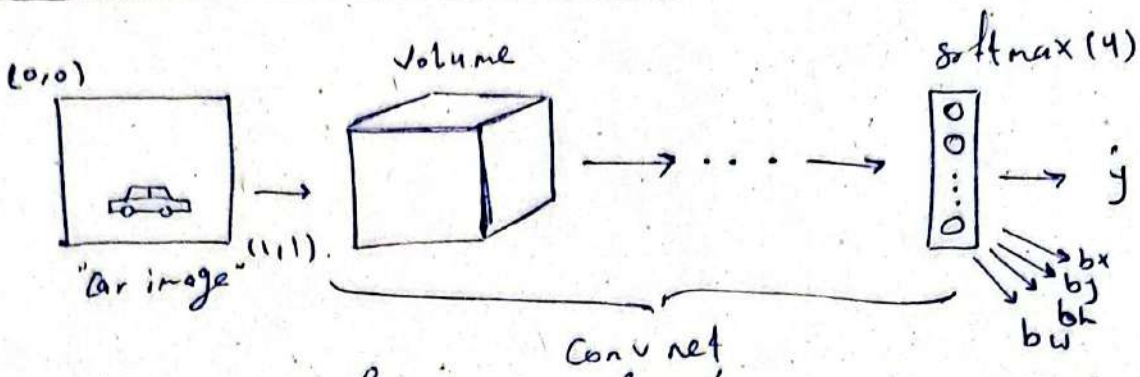
1 - if we have a very small dataset we can just feed our training examples (the whole dataset) to the pre trained nueral net without changing a thing; actually we'll freeze the weights in the Layers and just feed the data to the nueral net and classify the images using the nueral net's weights that we downloaded.

2 - if we have a bigger dataset; we can freez some of the earlier Layers and just train some of the Layers at the end of the nural net; so we'll freeze the weights in some layers but retrain and update the weights in some of the layers at the end, or we can just remove some of the last layers and create new ones with our own new hidden units.

3 - if we have alot of data we can just use the nueral net's architecture and the pretrained weights and use them for intialization and retrain the whole network on our data with the weights that we already have as intial values and use the same or almost same architecture

Week 3:

classification with localization:



→ we've seen before the classification problems where we had a picture on which we did some convolutions turning it to a vector of features in the end and we fed that vector of features to a softmax function and that softmax layer predicts the class in the image; this week we'll add to that a Bounding box around the object that we're trying to classify so now we're given ~~and~~ an image and our goal is not only to classify the image but also to localize it first in the image.

→ the classes that our softmax function has to predict may be:

1- pedestrian
2- car
3- Motorcycle
4- background (picture with no target objects)

- So for our "Bounding box" we'll need our model to output some parameters associated with the location of the object in the image; these parameters are:

1- bx → mid point of the Bounding box
2- by →
3- bh → height of the box
4- bw → width of the box

→ with the new parameters, the output y will change to include b_x, b_y, b_h and b_w + class labels.

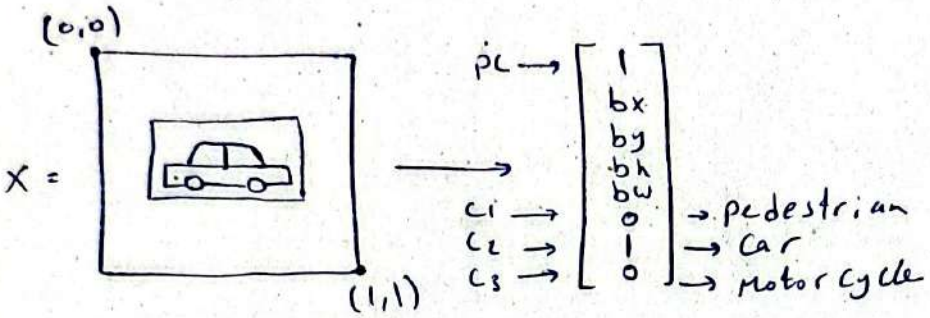
$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

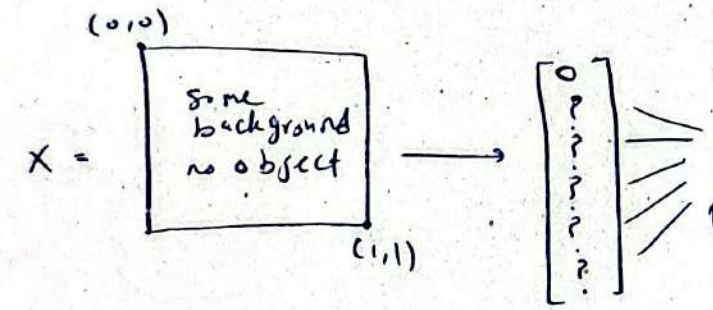
p_c → is there an object?

b_x, b_y, b_h, b_w → Bounding Box

c_1, c_2, c_3 → if p_c is true then which object is it?

* with the problem that we're dealing with; we assume that there is only one object in the image.





- Loss Function:-

→ if we're using squared error:

if $p_c = 1$

$$\mathcal{L}(\hat{y}, y) = \{(\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y^2)^2 \cdots \cdots (\hat{y}_8 - \hat{y}_8)^2\}$$

↳ there are 8 components of the output vector y

if $p_c = 0$

$$\mathcal{L}(\hat{y}, y) = \{(\hat{y}_1 - \hat{y}_1)\}$$

→ we said we'll use squared squared error just for the simplicity of the example but in reality we'll probably use Log likelyhood loss for c_1, c_2, c_3; squared error for the rest.

Land mark detections.

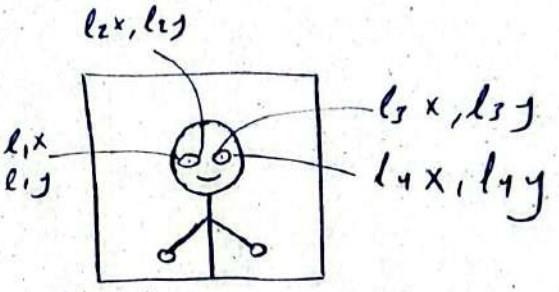
→ a Land mark is a (x, y) location in the image at which you want the nueral network to point.

↳ May be in face recognition problem this Land mark could be the corner of someone's eye.

* In general cases you can have a nueral network just output these Land mark in an image; for example; for the face detection problem you can have the nueral network output l_1x, l_1y, l_2x, l_2y

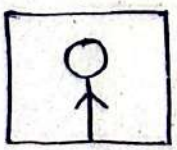
l_1x, l_1y → left corner of the first eye

and so on.



→ Generalizing this idea; you might want the nueral network to output a set of numbers that point to some Land marks in the image; like a punch of points for the eye; another punch for the mouth ... etc.

* of course for a nueral network to be trained to do such things you have to have Labled data of images with Labled Land marks.

 → Conv net → [PL → is there a face?, l1x, l1y, ..., l129x, l129y] ⇒ positions for 64 Land marks in the image.

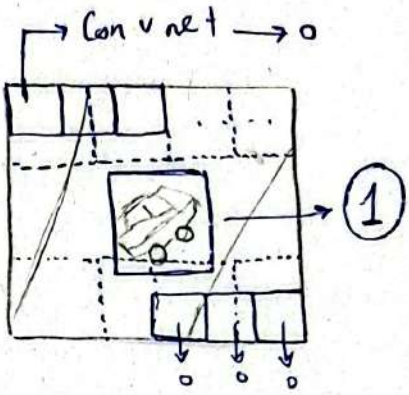
→ these Land marks can be used to classify the emotions or the facial expressions in the image such as smiling or frowning and so on.

→ Some apps use these Land marks to add filters to images like instegram and snapchat.

object detection:

Sliding window detection

→ In the sliding window detection you first get some closly cropped car images  no surroundings; just a small cropped picture of the car and train a classification ConvNet on these images having the network output 1 for car and 0 for no car.



→ Then you would input in to this ConvNet small squares from the image you want to detect an image in and get your output; you slide the window to the right and continue the process.

→ Having done the previous process once; we now take larger reigons or larger squares and feed them to the ConvNet.

→ Then you might do it even a third time using larger reigons and also feed them to the ConvNet.

→ Hopefully by doing this you'll finally be able to predict if there's a car in the image or not.

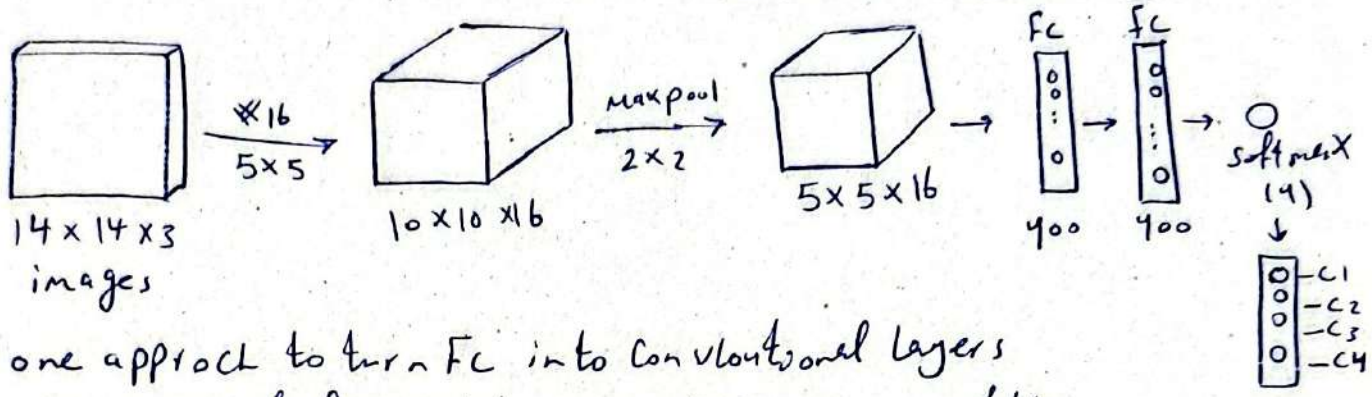
→ of course the disadvantage of sliding window detection is the computational cost because you're taking all these square reigons and feeding them to the ConvNet and each of these processes require some computations.

→ if we use big stride (skipping some pixels between each window) this of course would decrease the number of windows but also it would probably hurt performance.

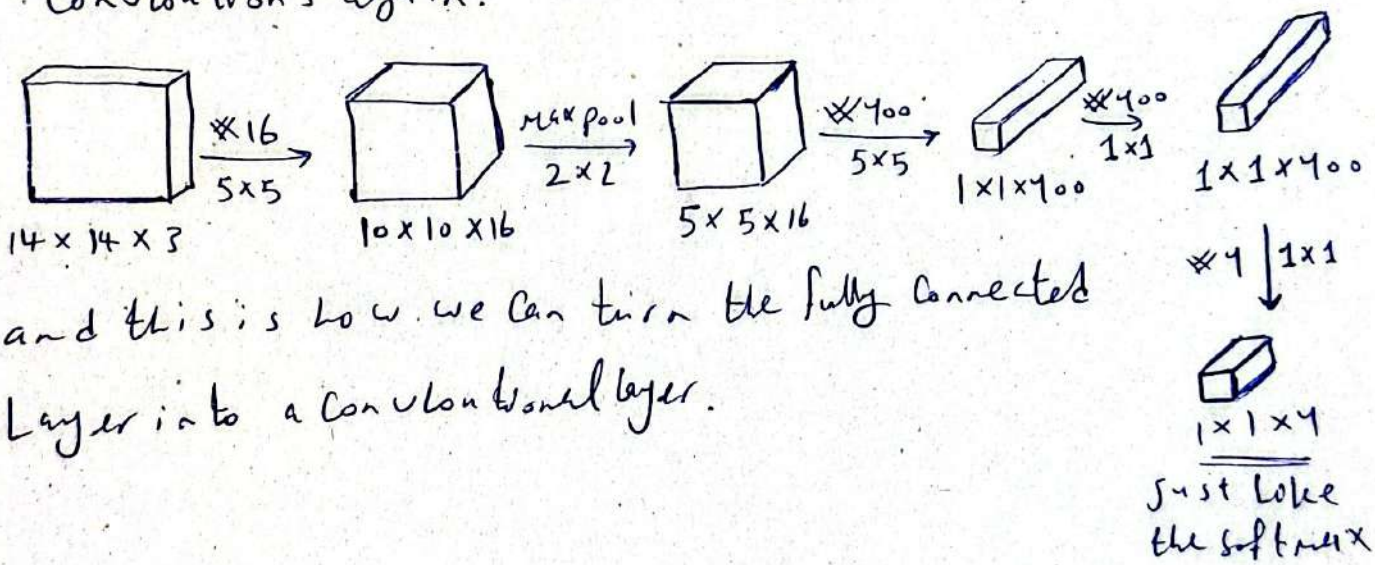
Convolutional implementation of sliding windows:

→ Since sliding window algorithm is computationally expensive when dealing with it and feeding each image to the Conv net; an implementation of Convolutional sliding window was made to reduce computations.

Turning Fully Connected Layers into Convolutional Layers:



→ one approch to turn Fc into Convolutional layers is instead of flattening the output volume of the last Convolutional layer, we'll just turn it into 1x1 x nC output volume and for the next Fc we'll do 1x1 x nC Convolutions again.



and this is how we can turn the fully connected layer into a Convolutional layer.

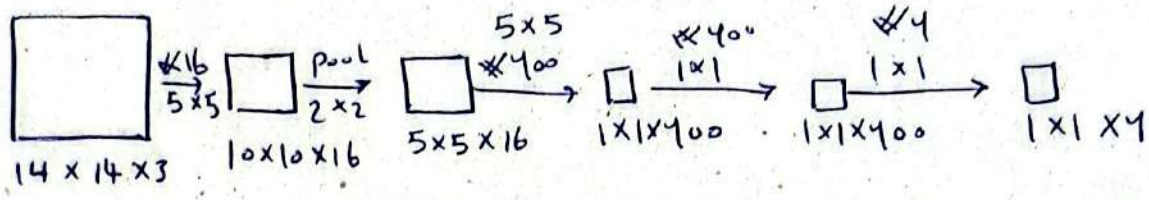
Just like the softmax

• What if the image that we're inputting to the Convnet is bigger than the image size the Convnet was trained on?

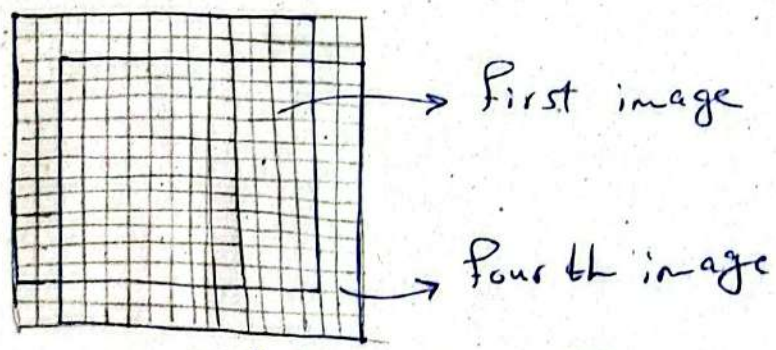
↳ We use sliding window.

* But since the normall sliding window is like we said befor computationally expensive we'll have to do it using Convolutions.

→ Let's say we have 16x16x3 images and the Convnet was trained on 14x14x3 images:-

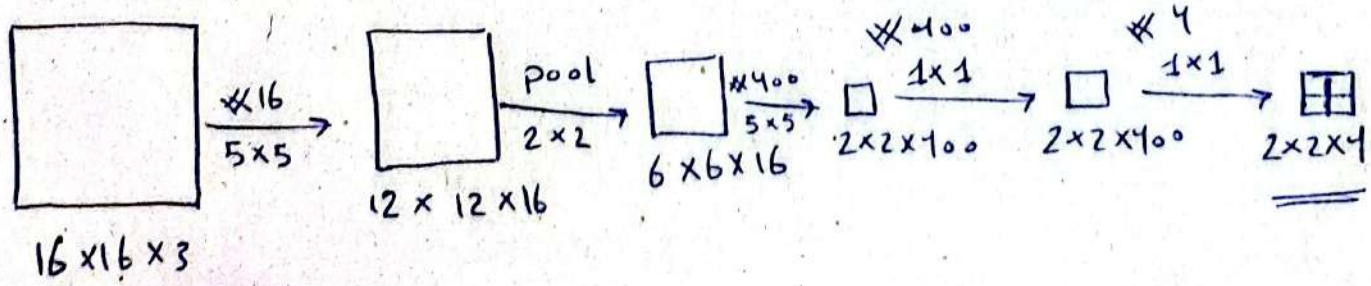


→ Now since our image is 16x16x3 we can't just feed it to the Convnet; we'll have to use 14x14 windows and a stride of 1; this will result in having 4 windows



so instead of having to feed multiple smaller crops of the same image to the Convnet, you just feed the Convnet the whole image and have multiple output vectors one for each crop

→ so instead of feeding 4 diffrent images to the Convnet we'll just feed it the 16x16x3 image to the Convolutional Layers and have our output be 2x2x4 with each vector corresponding to one of the four images



Bounding Box predictions :-

Yolo algorithm :-

→ Yolo stands for "you only look once" and it's an algorithm that utlizes convolutional sliding windows to ~~detect~~ predict Bounding boxes around objects in images.

* Instead of having to take multiple crops of images and feeding them to the CNN; now you can only look at the input image once as a whole and the algorithm will utilize convolutional to sliding windows on them which will include partial crops to be convluted with filters.

→ First we layout an $s \times s$ grid on the image, like the illustration and then do some convolutions on the image and make sure that the output shape is a ~~vector~~ $s \times s \times nc$ volume; each of the 9 vectors now has some values that correspond to one grid cell.

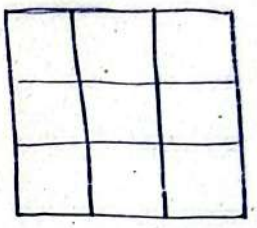
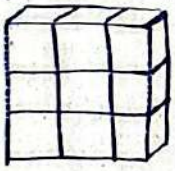
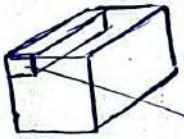


image grid



100x100x3 → Conv net → Maxpool → ..... →



3x3x8

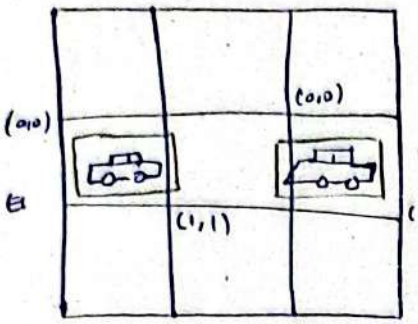
→ our output shape consists of 9 vectors each of them have 8 chanels; these chanels corresponds to some values

$$y_1 = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \begin{matrix} \rightarrow \text{is there an object in the grid cell} \\ \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} \text{Bounding Box} \\ \rightarrow \text{class 1?} \\ \rightarrow \text{class 2?} \\ \rightarrow \text{class 3?} \end{matrix}$$

* each grid cell have coordinates from (0,0) to (1,1) so b_x and b_y are $0 < b_x, b_y < 1$

→ the bounding box can be larger that the grid cell so b_h, b_w can be greater than 1.

specifying the Bounding Box



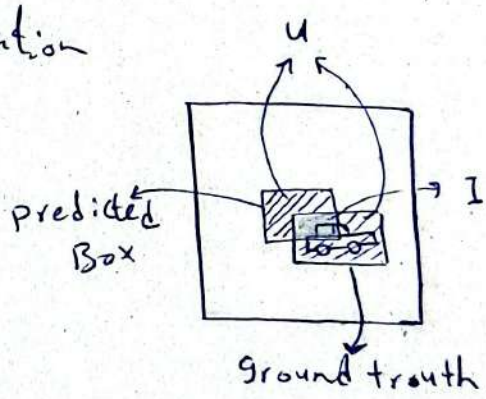
→ Coordinates for bx and by are relative to the upper left and lower right corners of the cell grid as they are the (0,0) and (1,1) coordinates for the object.

→ So bx and by can never be 1 as they point to the center of each object unless the center of some object is at some corner $0 < bx, by < 1$ unless in a corner.

→ bh and bw are calculated relative to the height and width of the grid cell but since the box can be larger than the grid cell; the can be greater than 1.

Intersection over union:-

→ Intersection over union is a evaluation method for objectLocalization; it computes the size of the intersection and devides it by the size of the union → IoU = $\frac{\text{size of } \blacksquare}{\text{size of } \blacksquare}$



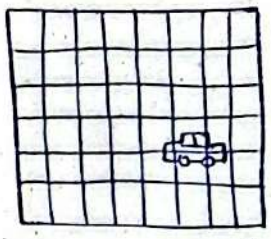
→ By convention; alot of computer vision algorithms would judge a Bounding Box Prediction if IoU ≥ 0.5.

→ If the predicted and the ground truth Boxes overlaped perfectly; the Intersection over union would equal 1.

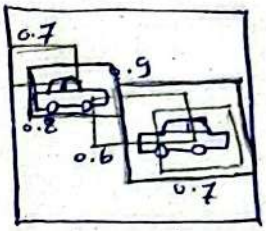
→ IoU is more generally a measure of the overlap between 2 Bounding Boxes.

Non Max suppression :-

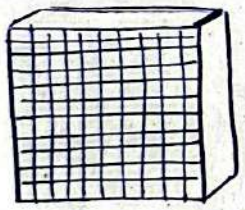
→ when we lay out the 5x5 grid on the image each of the grid cells has an indicator weather the cell has an object or not and of course there can be alot of cases where more than one grid cell has pc (the indicator of the object's existance) equal 1 indicating that the center of the object is in that grid cell; Non max suppression helps clean out these multiple predicted Bounding Boxes around an object to 1 Box.



→ Each of the Bounding Boxes has a parameter which tells us the confidence of weather an object exists in the Box. or not; non max suppression first looks at these values, it takes the highest confidence value and assumes that the object exists in this Box so it supresses all the other Boxes that have a big IoU with that Boxes and then looks at the next highest confidence and supresses the other Boxes that overlap with it and so on.

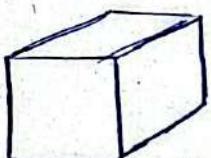


→ All the non supressed Boxes are our final predictions.



input image
19x19 grid

→ Conv net → Max pool - - - →



19 x 19 x (5)
output volume

in this example we don't care about the class of the object

→ Discard all boxes with pc ≤ 0.6

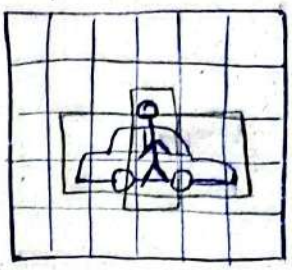
→ pick the Box with the highest pc and output it as a prediction

etc...etc ← → Discard any remaing Box with high IoU with the previous Box.

Anchor Boxes :-

→ one of the problems of object detection as we've seen it so far is that a grid cell can only detect one object but what if two object's centers are in the same grid cell?

→ what if we have 2 objects overlapping in the same grid?



image

* If we go about this problem with our current implementation of YoLo then we won't even be able to tell that there is 2 objects in the image because the vector associated with this grid cell has only 1 PC.

→ To solve this problem; Anchor Boxes were introduced as a solution; Anchor Boxes are predefined Boxes or shapes in the y vector; so instead of having just 1 pc and one of each bx, by, bh, bw, c1, c2, ... now we'll repeat these values again in the vector.

$$y = \begin{bmatrix} pc \\ bx \\ by \\ bh \\ bw \\ c1 \\ c2 \\ c3 \\ pc \\ bx \\ by \\ \vdots \end{bmatrix}$$

} Values associated with Anchor Box 1

} Values associated with Anchor Box 2

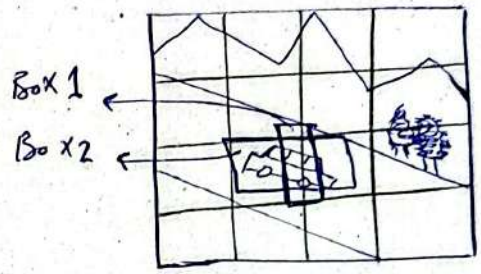
→ The Anchor Box that gets assigned to an object is the one that has a higher IoU (overlap) with the ground truth Bounding Box.

obj 1 → (grid cell, Anchor Box)
↳ it gets assigned.

Training with Yolo

→ The main idea of the Yolo algorithm is that we have an input image and some "Encoded" Lables, these encoded Lables match our desired output volume shape which is a number of $\hat{y}$ predicted vectors each corresponding to one grid cell of the $s \times s$ grid that we Laid out on top of the input image, the number of elements (channels) of that vector can vary based on the number of anchor boxes that we specify for each grid cell.

→ This is a 4x4 image on which we laid a 4x4 grid, we need to encode the lables for this image so we can use it for training.



*) Each grid cell will have some values associated with a vector y corresponding to a number of anchor boxes; 2 in this example

→ the values for each vector are as follows

→ pc → is there an object in the bounding Box

→ bx, by → Coordinates for the center of the object

→ bh, bw → height and width for the Box

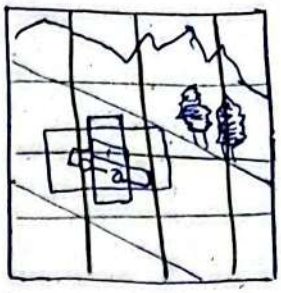
→ c_1, c_2, c_3 → which Class is the object

$$y = \begin{bmatrix} pc \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ pc \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} 1 \times 8 \\ \\ \\ \\ \\ \\ \\ \\ \text{Box 1} \\ \\ \\ \\ \\ \\ \\ \\ \text{Box 2} \end{matrix}$$

→ of Course we have to encode these lables ourselves for each image in order to be able to use it in the training phase.

→ How do we encode these lables?

* for an input image like the illustration; we have 1 object which is one of our target classes (car, pedistrian, motorcycle)



→ So since we have 1 object; this object will have only one center point associated with a certain grid cell; in this example it's grid cell 10.

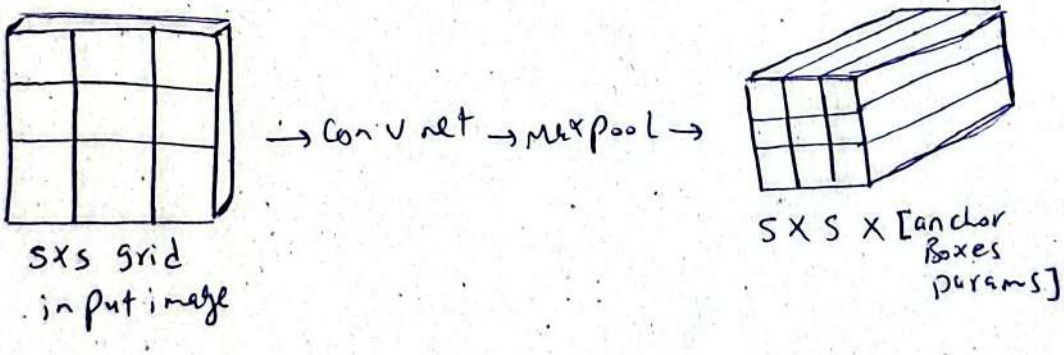
→ only grid cell 10 would have pc=1 and we will call it the ground truth, all other cells would have pc=0 so the rest of the encoded vector of lables are values that we don't care about.

→ Grid cell 10 now has pc=1, bx, by, bh, bw have assigned values for the first anchor box and also the second anchor Box resulting in a 16 chanel y vector

---

→ Now we can train our nueral net using the input image + the SxS output volume (lables)

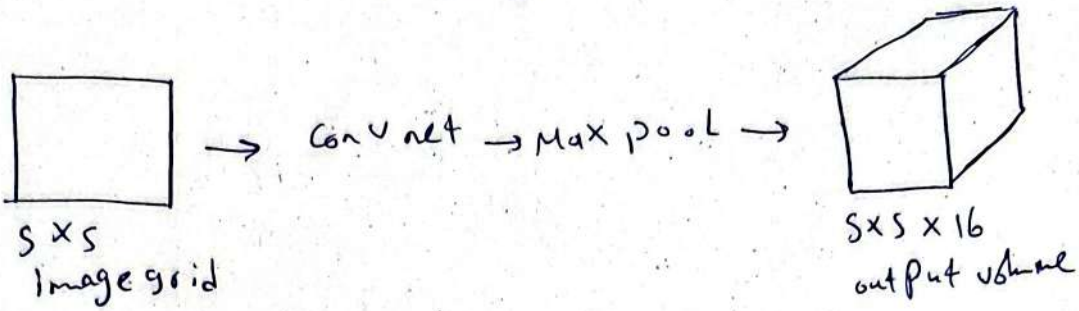
$y =$ [1, 0.5, 0.7, 0.6, 0.9, 0, 1, 0, 1, 0.2, 0.4, 0.6, 0.8, 0, 1, 0]



→ Each of the SxS output volume vectors corresponds to a grid cell.

Evaluation with Yolo:

→ After being done with the training phase; now the Yolo nueral net is ready to be tested.



→ We first take our input image and layout a SxS grid on it; feed it to the Conv net and get our output volume.

* Now we can evaluate our output volume using the IoU and non Max suppression first to drop one of the 2 anchor Boxes that we have, using a threshold for the intersection over union between the predicted output volume and the ground truth encoded lebele.

For grid cell 1o

$$y_{\text{ground truth}} = \begin{bmatrix} 1 \\ 0.5 \\ 0.4 \\ 0.6 \\ 0.9 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0.2 \\ 0.4 \\ 0.6 \\ 0.8 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad , \quad y_{\text{predicted}} = \begin{bmatrix} 1 \\ 0.7 \\ 0.8 \\ 0.2 \\ 0.6 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0.5 \\ 0.5 \\ 0.6 \\ 0.8 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

→ Now we can backprobagate using the loss function to improve the performance of the Nueral Network.

Semantic segmentation

→ semantic segmentation is the process of inputing an image and outputing a segmented image that corresponds to the classes in the input image.

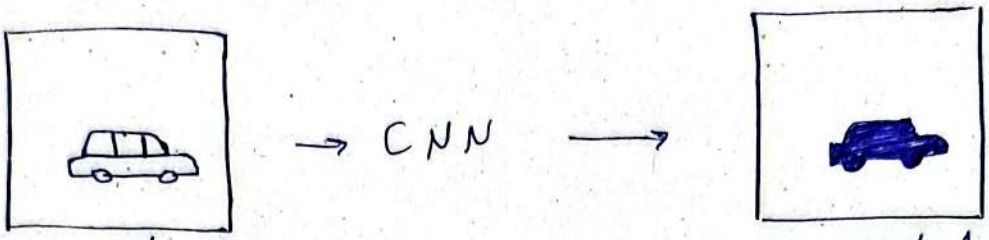


image of a car

segmented image.

→ So how does that work?

↳ to train a segmentation CNN you have to have the training data as labled segmented images and these labled images will have a pixel value for each element in the image so if there is a car, a motor cycle and background then the car might have a pixel value of 1, the motor cycle 2 and the back ground 0.

→ Here the car has pixel value 1, the trees pixel value 2, the road pixel value 3 and everything else which is the background has pixel value 0.

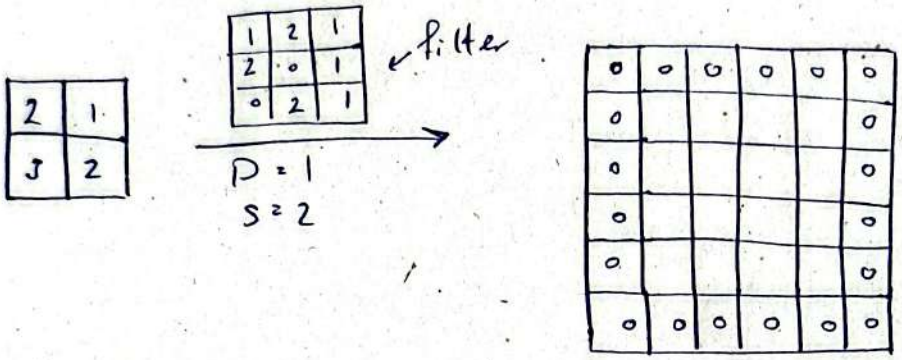
→ Again we find ourselves in front of a new problem which is that the output of the CNN has to be the same size as the input image, befor we learned about 1x1 Convolutions which allowed us to maintain the same image size but here it would be computationaly very expensive.

→ The solution to this problem is that we have to be able to do un sample the image to decrease the number of operations per layer which lowers the computational cost and up sample the image again to have an output which is the same size as the input.

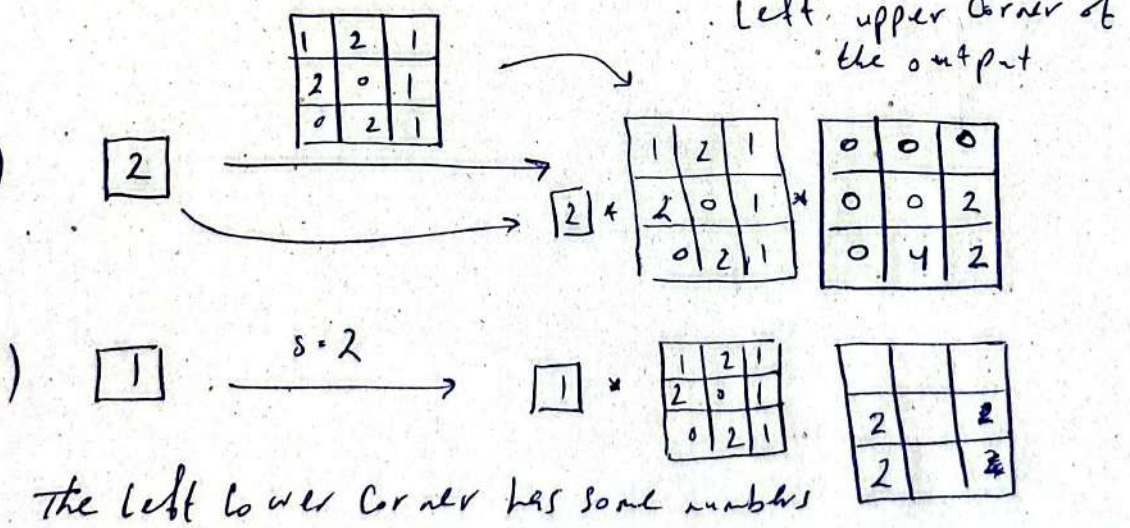
Transpose Convolutions :-

→ Transpose Convolutions is the solution to our problem since it reverses the convolution process.

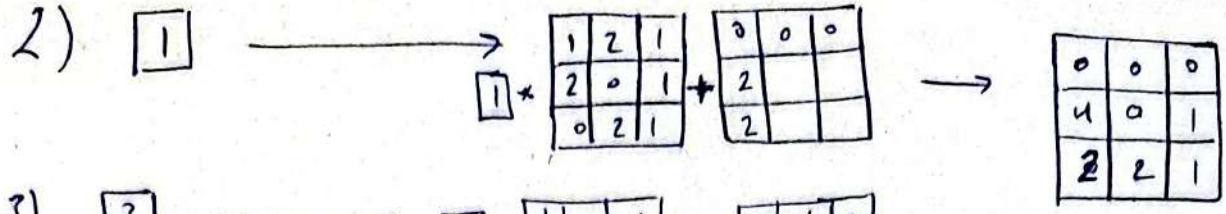
* How is transpose convolutions done?

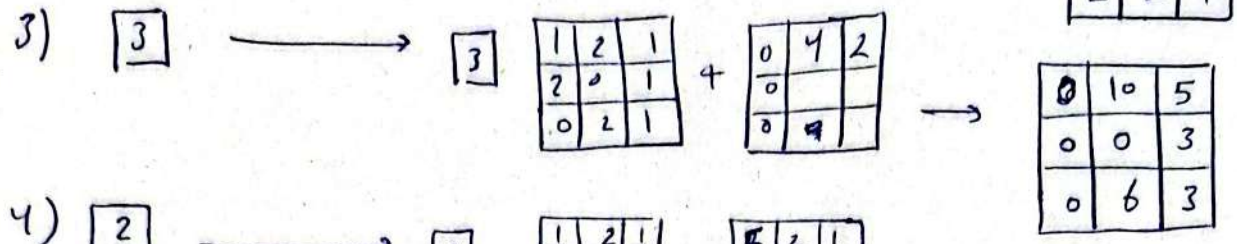


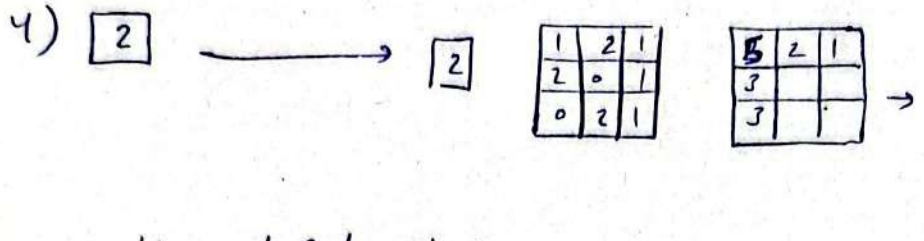
→ Instead of placing the filter on the input; we would place it on the output this time and multiplying it's values with the corresponding value in the input and go on and add the values of the multiplyed numbers to getter in the output



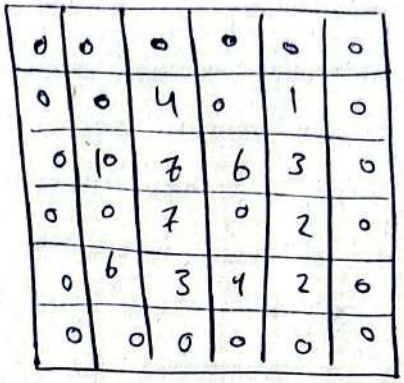
The left lower corner has some numbers which are values produced by the previous computation we'll add the new values to them.

2) 

3) 

4) 

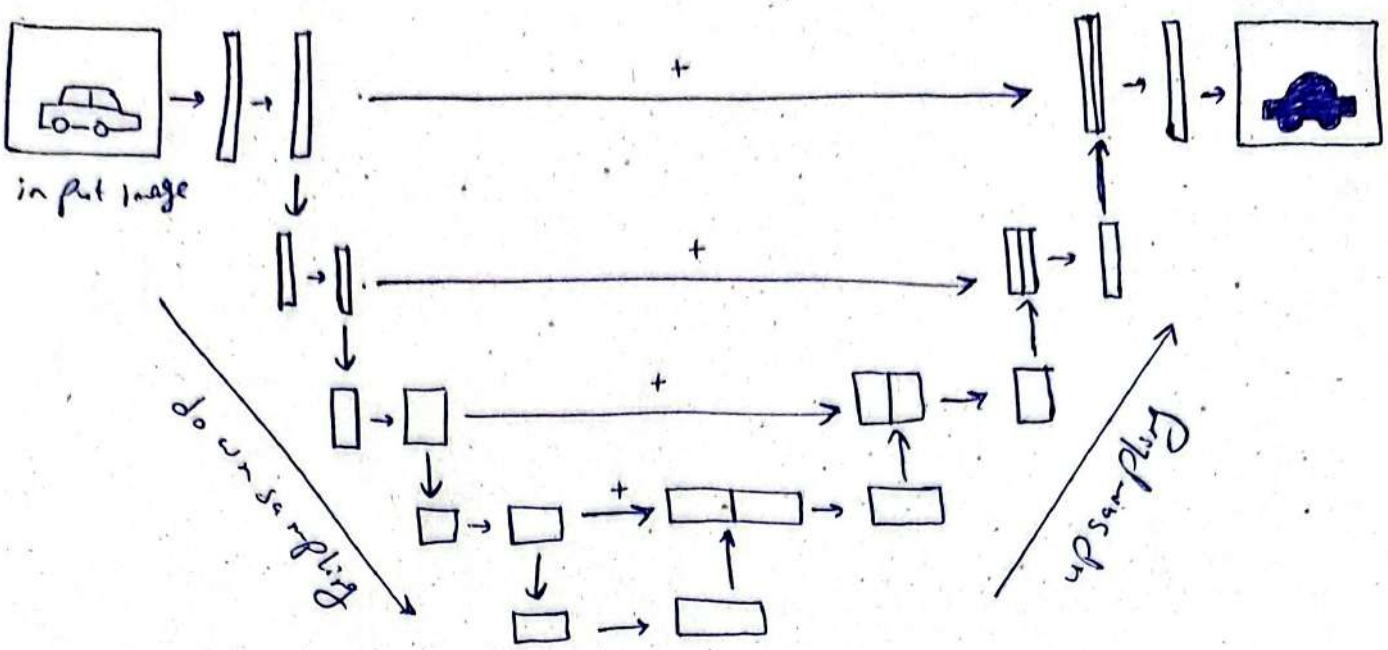
So the output will be →



There are multiple other ways to take a small input and turn it to a big output so other ways than the transpose convolutions can be used but transpose convolutions is just as effective, if not even more.

→ Next we'll use the idea of transpose convolutions in implementing the UNet which is a segmentation CNN based neural network.

U-Net



→ The U-Net is an image seg. neural network that utilizes transpose convolutions to upscale the output volume back to the input volume shape.

→ The U-Net also uses the concept of ResNets, adding each output volume in downsampling to the corresponding upsampled output volume with the same shape; this makes it more ~~repo~~ robust to problems like exploding and vanishing gradients, it also helps the model to learn the identity function of the inputs.

→ it's called a U-Net because it looks like the letter U.

→ When the Unet was first implemented it was intended to be an application that helps with segmenting medical images but it turned out to be useful for alot of Computer vision problems.

Week 4 :-

Face Recognition :-

→ Face Recognition is the problem of ~~Regd~~ recognizing faces and not just checking if a face matches an image; it's a technology that is potentially capable of matching a human face from a digital image or a video frame against a "database" of faces.

Face verification vs face recognition :-

FV →
↳ input image; name/ID
↳ output whether the input image is that of the claimed person.

FR →
↳ Has a database of k persons
↳ Gets an input image
↳ output ID of the image is any of the k persons.

---

* Face ~~Reg~~ recognition is much harder that face verification because if you have for example a face verification model that has 99% accuracy for matching an image to a single face; it would do much worse in maching a face to a database of images of a 100 people.

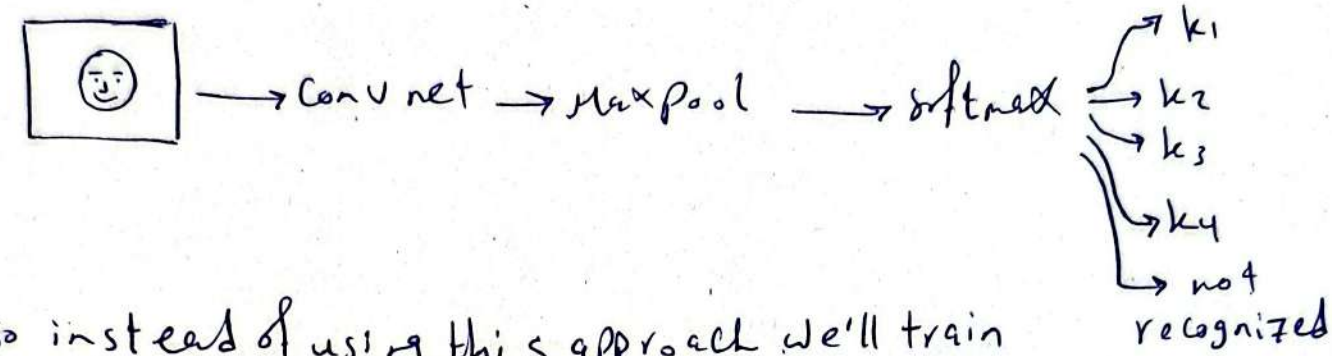
→ one of the challenges of solving the face recognition problem is solving the "one shot Learning" problem.

one shot Learning :-

↳ one shot Learning is being able to recognize a face with just one or only few images as training data.

→ Learning from one example to recognize a person again.

* passing an image of the face to a ConvNet and then to a softmax function to recognize if it's a face in the database or not is not a good solution because with just 1 image as a training example the model can't learn.



* so instead of using this approach we'll train a model to learn a similarity function :-

d (image1, image2) = degree of difference between images.

↓
difference

→ if the images are of the same person then you want d to output a small number (small difference) and if the images are of different people then you want d to be a large number.

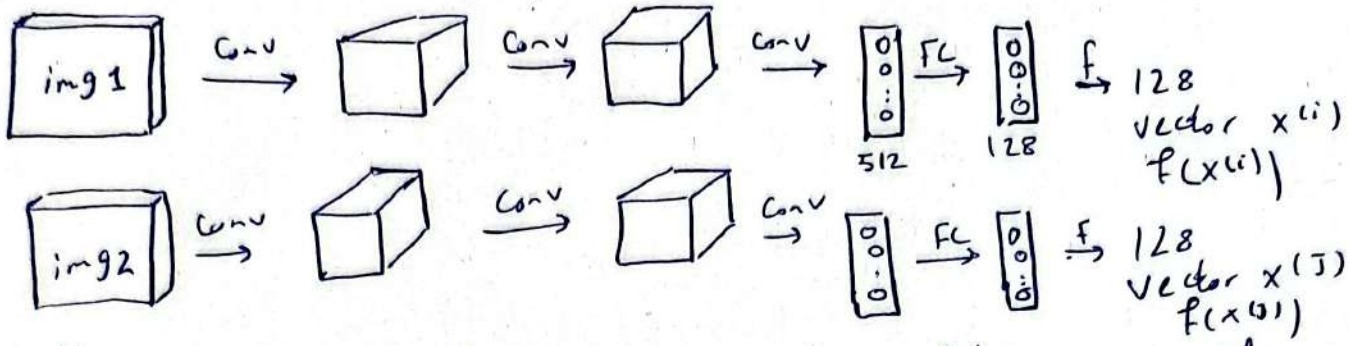
---

if $d \leq T$ → threshold → "same person"

if $d \geq T$ → "different people".

Siamese network :-

a siamese network take 2 vector representations of 2 images and calculates the difference between these representations if the difference is large then they are not of the same person and if small then they probably are of the same person.



~~These~~ 2 - 128 vectors are an encoding of the images; if you belive that this encoding is a good representation of the images then you can just calculate the difference between them using some function.

$$d(x^{(i)}, x^{(j)}) = \| f(x^{(i)}) - f(x^{(j)}) \|_2^2$$

small → "same person"

large → "different people".

Triplet Loss :-

→ In the termenology of the triplet loss function what you're going to do is look at 1 anchor image and a positive image which is an image of the person in the anchor image and again look at the same anchor image but with a negative image which is an image of a different person.

↳ you want to measure the distance between their vector representations so you can have

$$\| f(A) - f(P) \|_2^2 \leq \| f(A) - f(N) \|_2^2$$

So you want:-

$$\|f(A) - f(P)\|_2^2 - \|f(A) - f(N)\|_2^2 \leq 0$$

$d(A,P)$ — Anchor → positive

$d(A,N)$ — Anchor → negative

But since $0 - 0 = 0$ refering to the diffrences between the vector representations; the nueral network can learn to set everything to zero to satisfy the Loss function so instead of using $\|f(A) - f(P)\|_2^2 - \|f(A) - f(N)\|_2^2 \leq 0$ we will set it to $\|f(A) - f(P)\|_2^2 - \|f(A) - f(N)\|_2^2 \leq -\alpha$ where alpha is a hyper parameter that we can set; this small change prevents the network from setting all the parameters to zero and by convention the new loss is:-

$$\|f(A) - f(P)\|_2^2 - \|f(A) - f(N)\|_2^2 + \alpha \leq 0$$

→ alpha here is also called a margin which is driven from support vector machines.

Loss function :-

→ Given 3 images → A, P, N (A, P: same; A, N: different)

→ The triplet loss function is defined on triplets of Images.

$$L(A,P,N) = \left(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|_2^2 + \alpha\right)$$

we take the max between this and zero

$$\text{Loss} = \max\left(L(A,P,N), 0\right)$$

→ The effect of taking the max in the equation

$$Loss = \max(\mathcal{L}(A,P,N), 0)$$

is that as long as the $\mathcal{L}(A,P,N)$ is smaller than 0 then the loss is equal to zero which is what we want but if $\mathcal{L}(A,P,N)$ is larger than zero then the loss is equal to it's value.

$$\mathcal{L}(A,P,N) = \max(\|f(A) - f(P)\|_2^2 - \|f(A) - f(N)\|_2^2 + \alpha, 0)$$

↳ Now the overall cost function is:

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

* so if we have a 10k training set of 1k people then we can use the training data to create triplets of (A,P,N) and then train our algorithm using Gradient descent on the loss function that we defined.

---

choosing the triplets (A,P,N):

↳ one of the problems of this approach is if A,P,N were chosen randomly given that A and P are images of the same person then the condition that:

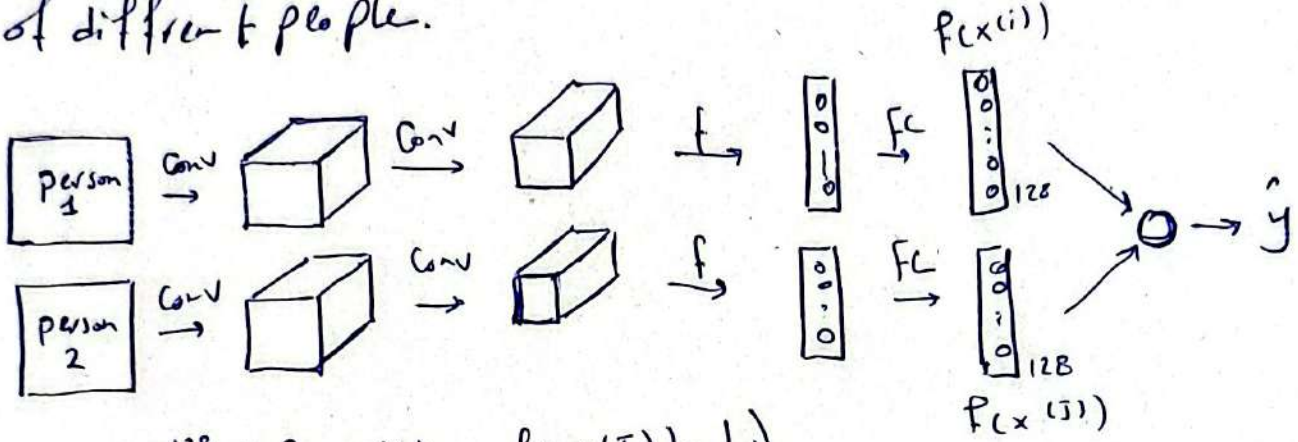
$$d(A,P) + d(A,N) + \alpha \leq 0 \rightarrow \text{is easily satisfied}$$

Because given A,P,N the difference between A and N is likely very large regarding the vector representation.

→ choose triplets where $d(A,P) \approx d(A,N)$ because having a small difference increases the efficiency of the algorithm.

Face verification and Binary classification:

* Another way to train a face recognition algorithm is having two neural networks embed images of same or of different people then feed these vector embeddings to a Binary classification function: if the output $\hat{y} = 1$ then the two images are of the same person if not then they are of different people.



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

(weights) (bias)

↳ so the sigmoid function classifies the difference between the different vector encodings of the images.

→ we first substract element wise values of the vectors and then feed it to the sigmoid function and get our output $\hat{y}$.

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

* other ways of measuring the difference could be

$$\frac{\left(f(x^{(i)})_k - f(x^{(j)})_k\right)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$$ → This is called the chi-squared similarity or formula χ^2