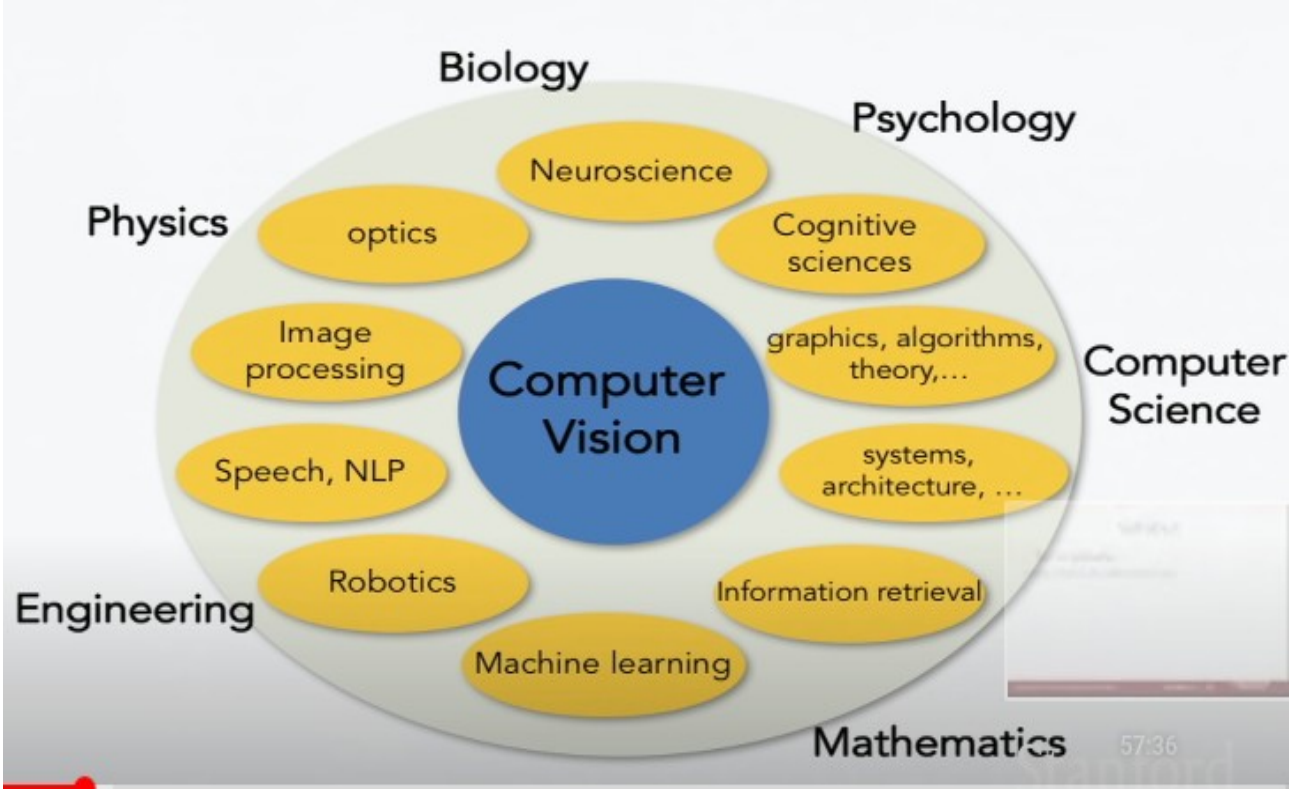
Computer Vision

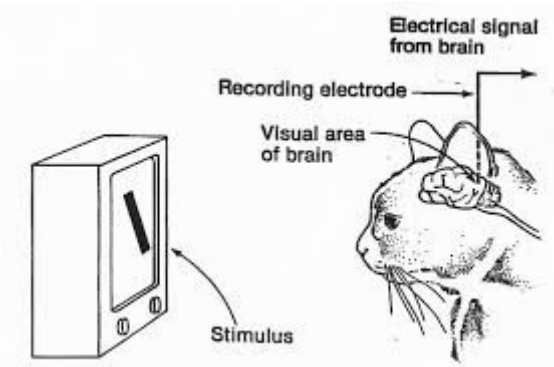


Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex(a part of the brain which processes the image), and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in Chapter 11 (Vanishing/gradient problem , Regularization , Optimization , Fine Tuning)for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing. Most of the algorithms / Architectures we build in CNN , will help us as idea for building voice recognition and Natural Language systems However, we will focus on visual applications for now.

In this chapter we will explore where CNNs came from, what their building blocks look like, and how to implement them using TensorFlow and Keras. Then we will discuss some of the best CNN architectures, as well as other visual tasks, including Image Classification , Object Detection(Classifying multiple objects in an image and placing bounding boxes around them for

identifying the position of the objects , recognizing where that object takes place in the Image) , and semantic segmentation(classifying each pixel according to the class of the object it belongs to , not object detection , this helps to only focus on the important part of the image for example Brain Tumor from MRI image or mass from a chest x ray).

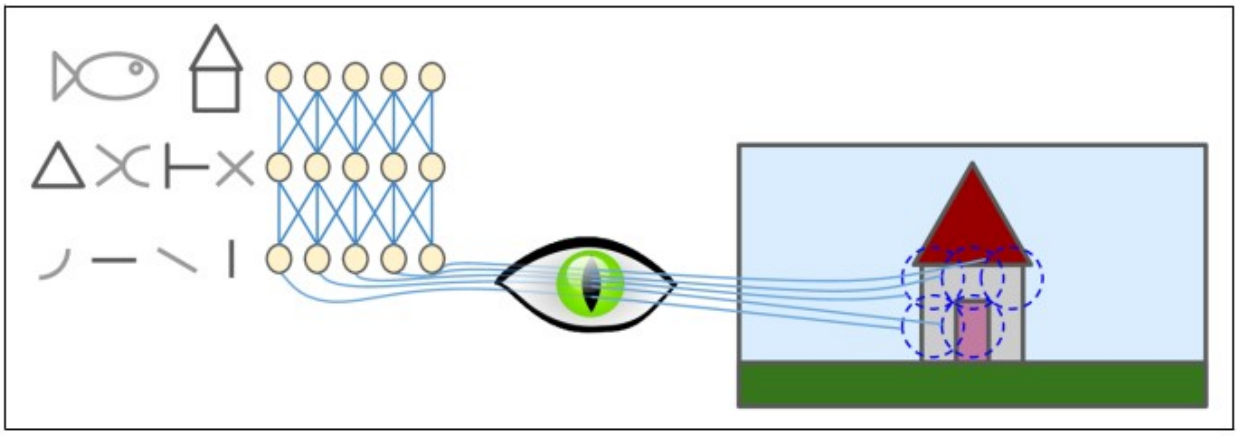
The Architecture of the Visual Cortex



David H.Hubel and Torsten Wisel experiment on cats :- Convolutional neural networks (CNNs) were inspired by early findings in the study of biological vision. One of the most influential work in both human vision and animal vision , as well inspired computer vision is the work done by Hubel and wisel in the 50's and 60's using electrophysiology. What they were asking the question is "what was the visual processing mechanism like mammal's " , they choose to study cat's brain which is more or less similar to human brain from the visual processing point of view ,what they did to stick some electrodes in the back of the cat brain which is where the primary visual cortex area is and then look at what stimuli makes the neurons in the back of the primary visual cortex of a cat brain respond excitedly. In particular , they showed that many neurons in the visual cortex have a small local receptive field , meaning they react only to visual stimuli located in a limited region of the visual field .The receptive fields of different neurons may overlap , and together they tile the whole visual field. So what they really learned is that the visual processing starts with simple structure of the visual world , oriented edges and as information moves along the visual processing path way the brain builds up the complexity of the visual information until it can recognize the complex visual world.

Moreover , the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line

orientations). They also noticed that some neurons have larger receptive fields and they react to more complex patterns that are combinations of the low level patterns.



Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules , neurons respond to more complex patterns in larger receptive fields.

These studies of the visual cortex inspired the neocognitron , introduced in 1980, which gradually evolved in to what we now call convolutional neural networks. An important mile stone was a 1998 paper by Yann Lecun that introduced the famous LeNet-5 architecture , widely used by banks to recognize hand written check numbers. The architecture has some building blocks such as fully connected layers and sigmoid activation functions but it also introduces two new building blocks : convolutional layers and pooling layers.

what is neocognitron ? The neocognitron is a hierarchical, multilayered artificial neural network proposed by Kunihiko Fukushima in 1979. It has been used for Japanese handwritten character recognition and other pattern recognition tasks, and served as the inspiration for convolutional neural networks.

Building Block Of Convolutional Neural Network

The convolutional layers are the major building blocks in convolutional neural networks. In the context of a convolutional neural network , a convolution is a linear operation that involves the multiplication of a set of weights with the input , much like a traditional neural network. Given that the technique was designed for two dimensional input, the multiplication is performed between an array of input data and a two-dimensional array of weights , called a filter or a kernal.

The filter is smaller than the input data and the type of multiplication applied between a filter - sized patch of the input and the filter is a dot product. A dot product is the element wise multiplication between the filter - sized patch of the input and the filter is a dot product , which is then summed always resulting in a single value. Because it results in a single value , the operation is often referred to as the scalar product(shape = rank 0). What is the difference between Dot product and element wise operator ? With the dot product you multiply the corresponding components and add those products together , The elements corresponding to same row and and column are multiplied together and the products are added such that , the result is scalar , unlike matrix multiplication the result of dot product is not another vector or matrix , it's a scalar. With the Hadamard product (element-wise product) you multiply the corresponding components, but do not aggregate by summation, leaving a new vector with the same dimension as the original operand vectors.

The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem , such as image classification. The result is highly specific features that can be detected anywhere on the input images.

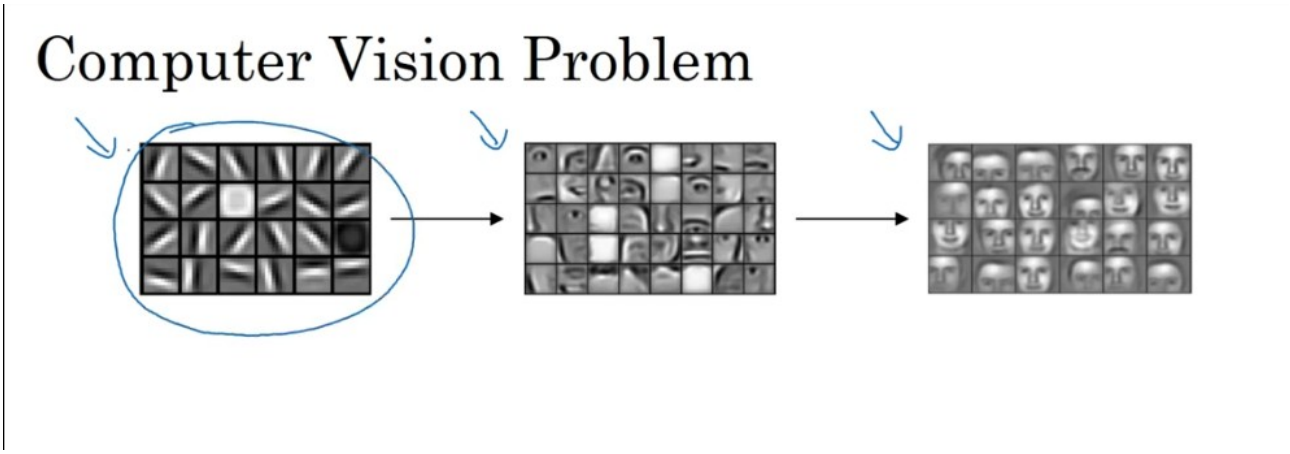
Why not simply use a deep neural network with fully connected layers for image recognition tasks?

Unfortunately , although this works fine for smaller images (e.g MNIST) , it breaks down for larger images because of the huge number of parameters it requires . For example:- a 100*100*3 - pixel image has 30,000 pixels and if the first layer has just 1000 neurons , this means a total of 30million connections , and that's just the first layer. CNNs solve this problem using partially connected layers and weight sharing.

The other reason is all the multilayer neural networks we've looked at so far had layers composed of a long line of neurons and we feeding them to the neural networks. In a CNN each layer is represented in 2D , which makes it easier to match neurons with their corresponding inputs. so Convolution neural networks are architectures suite for images.

Edge Detection

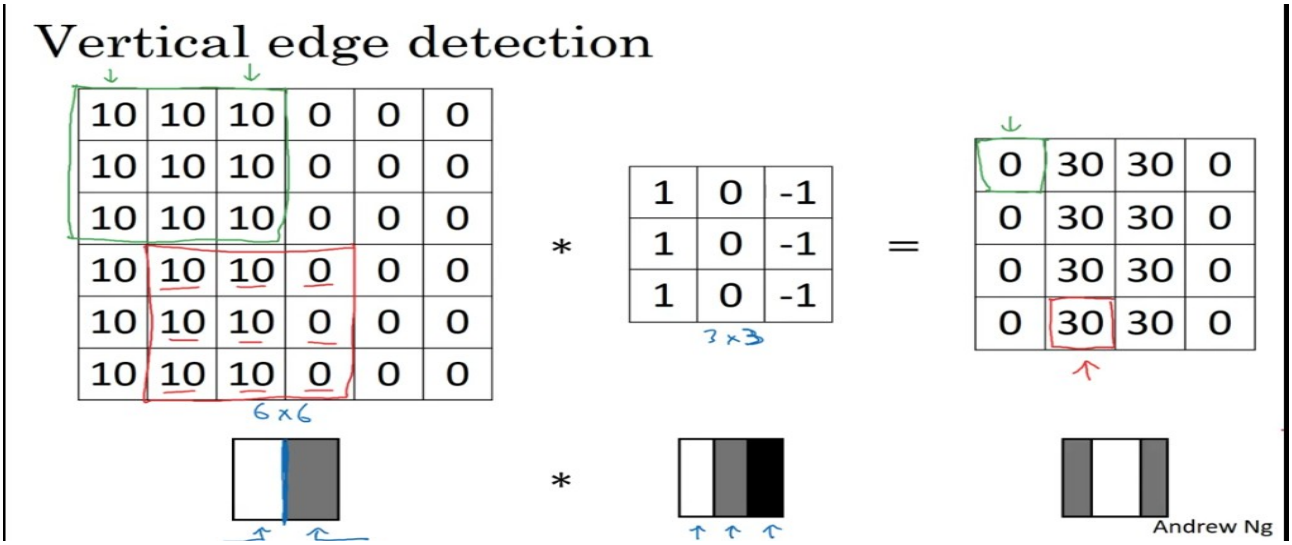
In neural networks the earlier layers might detect edges and somewhat later layers might detect parts of objects and then even later layers may be detect parts of complete objects like peoples faces.



Edge detection is an image processing technique for finding the boundaries of objects within images. It works by detecting discontinuities in brightness.

Edge detection is a technique of image processing used to identify points in a digital image with discontinuities , simply to say , sharp changes in the image brightness varies sharply are called the edges (or boundaries) of the image.

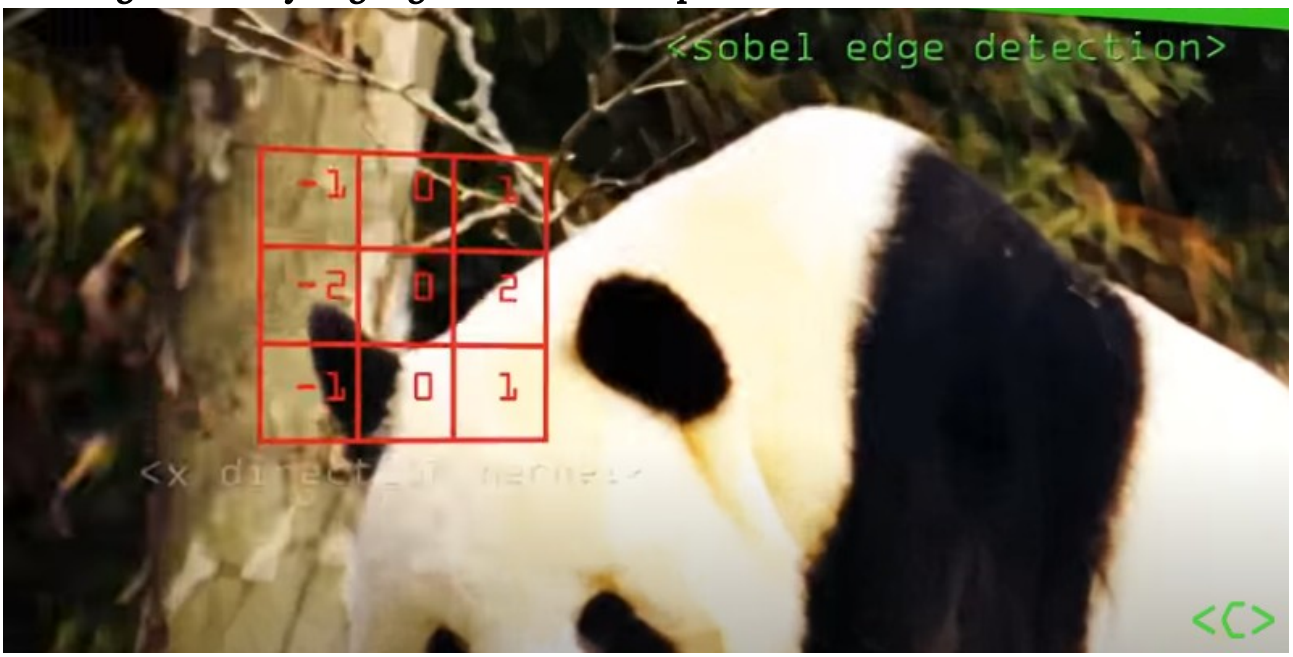
For example:- below is hand crafted **3*3** element filter for detecting vertical lines:



Applying this filter to an image will result in a feature map that only contains vertical lines. It is a vertical line detector.

A high value of edge indicates there a steep change and a low value indicates there is a shallow change.

You can see this weight values in the filter , any pixels values in the center vertical line will be positively activated and any on either side will be negatively activated. Dragging this filter systematically across pixel values in an image can only highlight vertical line pixels.



A horizontal line detector could also be , created and also applied to the image , for example:

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

Combining the results from both filters e.g combining both feature maps , will result in all of the lines in an image being highlighted. A suite of tens or even hundreds of other small filters can be designed to detect other features in the image.

The innovation of using the convolution operation in a neural network is that the values of the filter are weights to be learned during the training of the network.

The network will learn what types of features to extract from the input . Specifically , training under stochastic gradient descent , the network is forced to learn to extract features from the image that minimize the loss for the specific task the network is being trained to solve , e.g extract features that are the most useful for classifying images as dogs or cats.

In this context , you can see that this is a powerful idea.

Multiple Filters
Convolutional neural networks do not learn a single filter , they in fact , learn multiple features in parallel for a given input.

For example:- it is common for a convolutional layer to learn 32 to 512 filters in parallel for a given input.

This gives the model 32 , or even 512 different ways to extracting features from an input or many different ways of both “learning to see” and after

training many different ways of “seeing” the input data , this diversity allows specialization , e.g not just lines seen in your specific training data.

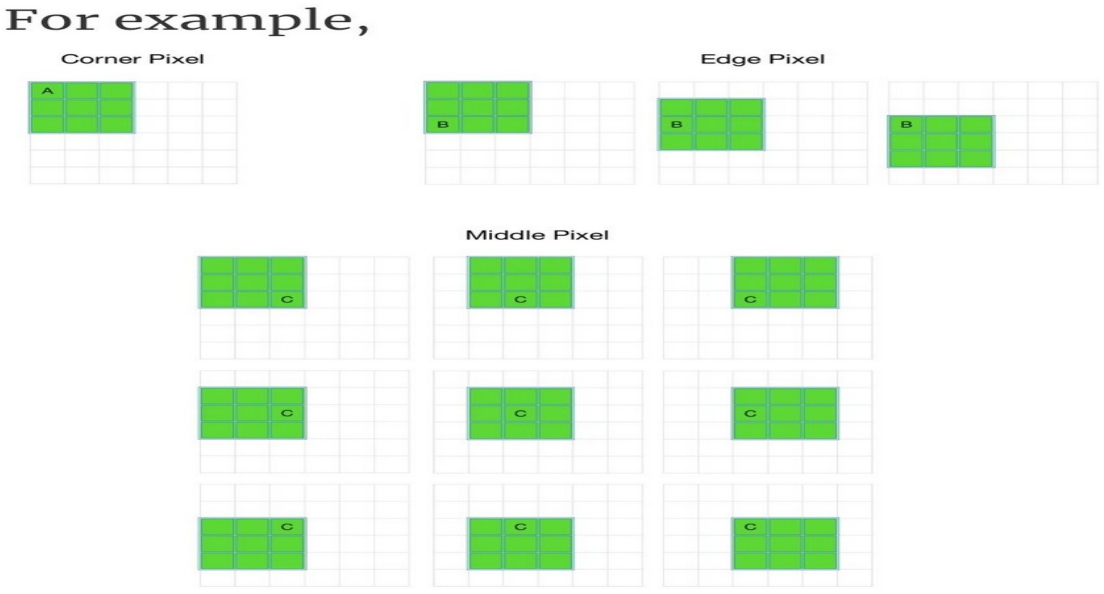
What is Padding

There are two problems in here :-
1) Some pixels may overlap(mostly pixels in the middle will be multiplied so many times , but pixels in the corners may take only once. these leads to huge information loss).

2) The feature - map dimension is smaller than the input image , when we have hundreds of convolutional layers , we will extract features using filters in each layer , so the final output image will be very much smaller than the input.

Problem with simple convolution layers for a gray scale (n*n) image and (f*f) filter/kernal , the dimensions of the image resulting from a convolution is (n-f+1) * (n-f+1) .

For-example:- for an (8*8) image and (3*3) filter , the output resulting after convolution operation would be of size (6*6) thus , the image shrinks every time a convolution operation is performed. This places an upper limit to the number of times such an operation could performed before the image reduces to nothing thereby precluding us from building deeper networks.



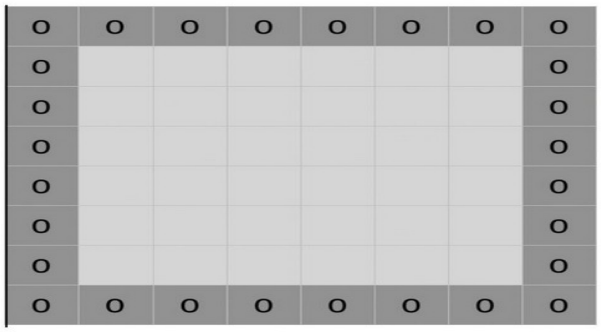
Also, the pixels on the corners and the edges are used much less than those in the middle.

Clearly , pixel A is touched in just one convolution operation and pixel B is touched in 3 convolution operations , while pixel C is touched in 9 convolution operations. In general , pixels in the middle are used more than the pixels on corners and edges. Consequently , the information on the borders of images are not preserved as well as the information in the middle.

To overcome these problems we use padding..

Padding Input Images

Padding is simply a process of adding layers of zeros to our input images so as to avoid the problems mentioned above.



Zero-padding added to image

This prevents shrinking as , if p= number of layers of zeros added to the border of the image , then our (n*n) image becomes (n+2p) * (n+2p) image after padding so , applying convolution operation (with (f*f) filter) outputs (n+2p-f+1) * (n+2p-f+1) images. For example:- adding one layer of padding to an (8*8) image and using a (3*3) filter we would get an (8*8) output after performing convolution operation . This increases the contribution of the pixels at the border of the original image by bringing them into the middle of the padded image. Thus , information on the borders is preserved as well as the information in the middle of the image.

Types of padding

Valid-Padding:- It implies no padding at all. The input image is left in its valid/unaltered shape. So, [(n) * (n) image] * [(f*f) filter] -> [(n-f+1)*(n-f+1) image]

where * represents a convolution operation.

Same Padding :- in these case , we add 'p' padding layers such that the output image has the same dimensions as the input image. So,
[(n+2p)*(n+2p) image] * [(f*f) filter] -> [(n*n) image] which gives p = (f-1)/2 because (n+2p -f +1 = n)

so , if we use a (the 3*3) filter the 1 layer of zeros must be added to the borders for same padding. Similarly , if (5*5) filter is used 2 layers of zeros must be appended to the border of the image.

For the most part the filter must be odd , otherwise if it is even then some part of the padding with more pixels and some portion of the padding with less amount of pixels.

How to choose which of the padding that we need to select ? One of the advantage of the padding is to make the input image size compatible with the output image size, for this we need to have p padding for a certain f filter.
n + 2p - f + 1 = n (This means the input image size must be equal to the output image size)

$$n + 2p - f + 1 = n$$
$$2p - f + 1 = 0$$
$$p = f-1/2$$

Example , let's say we have a 3*3 filter then p = 3 - 1/2 = 1// so we need one padding for having an output size as the same as the input size.

Let's say we have a 5*5 filter then p = 5 - 1/2 = 2// so we need two padding for having an output size as the same as the input size.

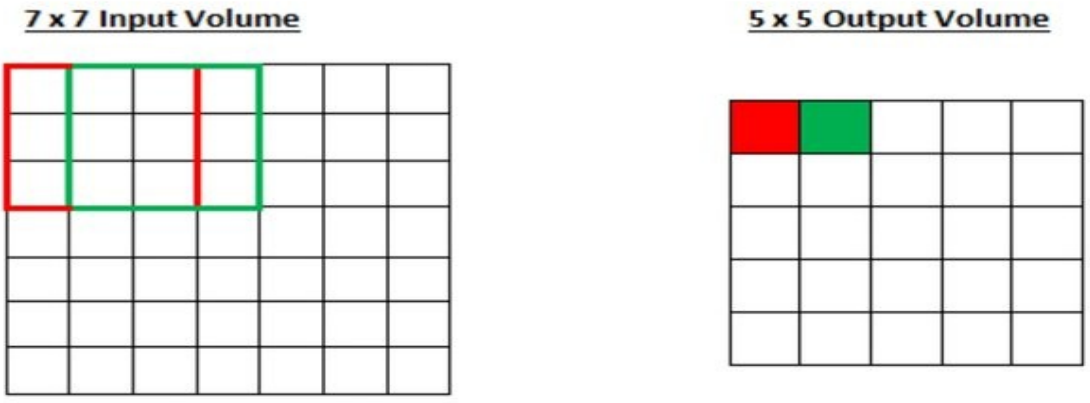
Why is the shape of a filter applied to a CNN layer always odd?

Now in order to padding size (p) to be a whole number we need f to be odd. This is why it is a convention to use the odd-shaped filter matrix. other wise if it's even then some part of the padding will have more pixels and some portion of the padding will have less amount of pixels.

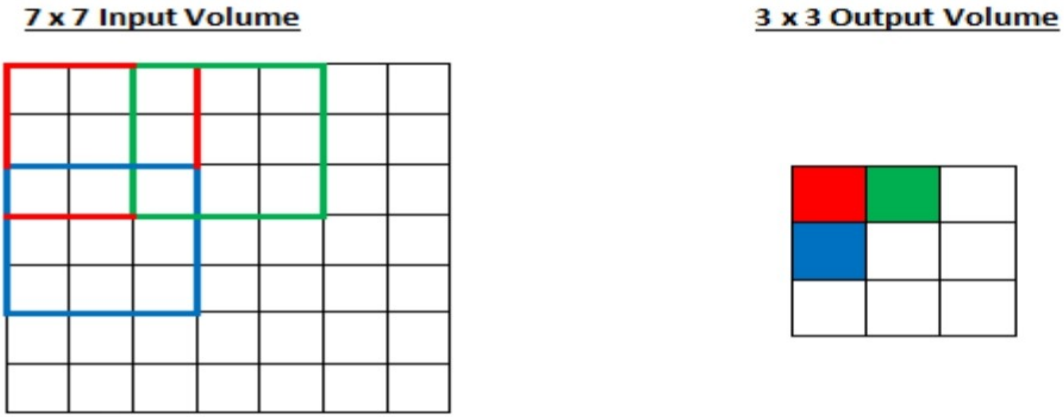
Strided Convolutions

Stride controls how the filter convolves around the input volume , in the example we had , the filter convolves around the input volume by shifting one

unit at a time. The amount by which the filter shifts is the stride. In that case , the stride is normally set in a way so that the output volume is an integer not a fraction. So if it's going to be a fraction number we will going to take the floor , which is the floor of n + 2p - f/s + 1 Let's look at an example , Let's imagine a 7*7 input volume , a 3*3 filter and a stride of 1. This is the case that we're accustomed to



same old , same old , right ? see if you can try to guess what will happen to the output volume as the stride increases to 2.



So, as you can see, the receptive field is shifting by 2 units now and the output volume shrinks as well. Notice that if we tried to set our stride to 3, then we 'd have issues with spacing and making sure the receptive fields fit on the input volume. Normally , programmers will increase the stride if they want receptive fields to overlap less.

The filter is moved across the image left to right , top to bottom , with one-pixel column change on the horizontal movements , then a one pixel row change on the vertical movements.

The amount of movement between applications of the filter to the input image is referred to as the stride , and it is almost always symmetrical in height and width dimensions.

The default stride or strides in two dimensions is (1 , 1) for the height and the width movement performed when needed. And this default works well in most cases.

The stride can be changed , which has an effect both on how the filter is applied to the image and in turn , the size of the resulting feature map.

For example:- the stride can be changed to (2 , 2) . This has the effect of moving the filter two pixels right for each horizontal movement of the filter and two pixels down for each vertical movement of the filter when creating the feature map.

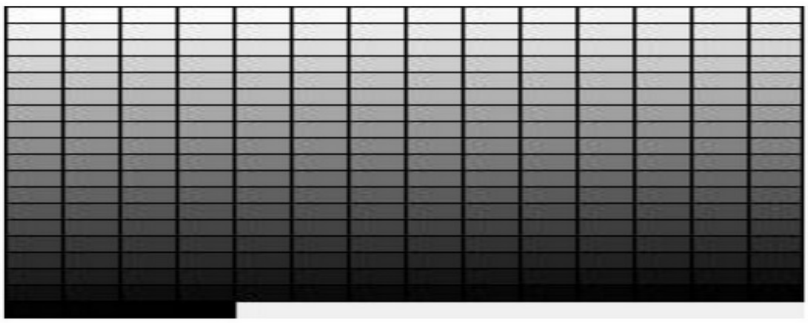
We default to sliding one element at a time. However , sometimes , either for computational efficiency or because we wish to down sample , we move our window more than one element at a time , skipping the intermediate locations.

The stride can reduce the resolution of the output. Applying the handcrafted filter to the input image and printing the resulting feature map , we can see that , indeed the filter still detected the vertical line and can represent this finding with less information. With or without the padding if the receptive field is not fit the input volume we will discard it. But there will be a loss in information how can we handle it ?

Convolution in RGB images
Objective :-What does it mean by Gray Scale and RGB in convolution
- How to compute the convolution for RGB images
- The padding and the stride for RGB convolutional images.

GrayScale and RGB

We”ll start simple..with a gray-scale image. A grayscale(Black and white) picture just needs intensity information - how bright is a particular pixel. The higher the value , the greater the intensity. Current displays support 256 distinct shades of gray. Each one just a little bit lighter than the previous one.



The grayscale palette

So for a gray-scale image, all you need is one single byte for each pixel. One byte (or 8-bits) can store a value from 0 to 255, and thus you'd cover all possible shades of gray. So in the memory, a gray-scale image is represented by a two dimensional array of bytes. The size of the array being equal to the height and width of the image. Technically, this array is a "channel". So, a gray-scale image has only one channel. And this channel represents the intensity of whites. An example gray-scale image:

A grayscale image

Coloured images When color is added, things get trickier. More information needs to be stored. Its no more just about what shade. Its about what shade of which "color". To be able to distinguish these different shades, you need 3 bytes for each pixel (3 bytes, or 24-bits, can store a value up to 255255255... which is equal to 16,581,375).

A colour image

Now think about this: You have 16 million numbers to assign to different shades of colors. If you just randomly assigned colors to each number, things would get weird. (say, 1=brightest red, 2=brightest green, 45780=dullest yellow, etc). So people figured out different "color spaces" to systematically assign numbers to the HUGE number of colors. The RGB color space We'll start off simple, with the most common color space: RGB. The 3 bytes of data for each pixel is split into 3 different parts: one byte for the amount of red, another for the amount of green and the third for the amount of blue. Red, green and blue being primary colors can be mixed and different proportions to form any color You have 256 different shades of red, green and blue (1 byte can store a value from 0 to 255). So you mix these colors in different proportions, and you get your desired color. This color space is quite intuitive. You've probably used it all the time without realizing what all was going on behind the scenes. And since you've "dedicated" one byte of each pixel to red, the second byte to green and the last byte to blue... it makes sense to club these bytes together. That is, all the "dedicated red" bytes together... the "dedicated green" bytes together at another place... and the blue ones at another location. And, behold, you get the red channel, the green channel and the blue channel!!

Black - (0 , 0 , 0)
white - (255 , 255 , 255)
Red - (255 , 0 , 0)
Green - (0 , 255 , 255)
Blue - (0 , 0 , 255)

Convolution Over Volumes

we called it volumes because instead of representing images in two dimensional we are now using three dimensions.

In the previous parts, we were considering images on a grayscale (black and white images). So what if the input images have color? It turns out that now we have to consider 3D convolutions instead of 2D convolutions.

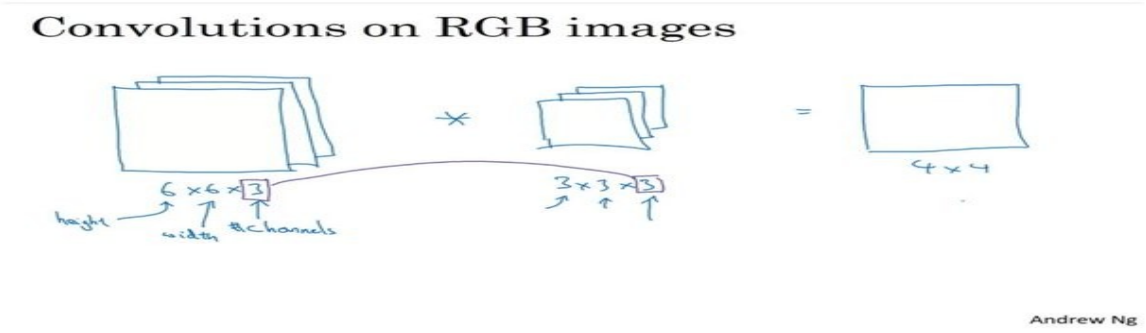
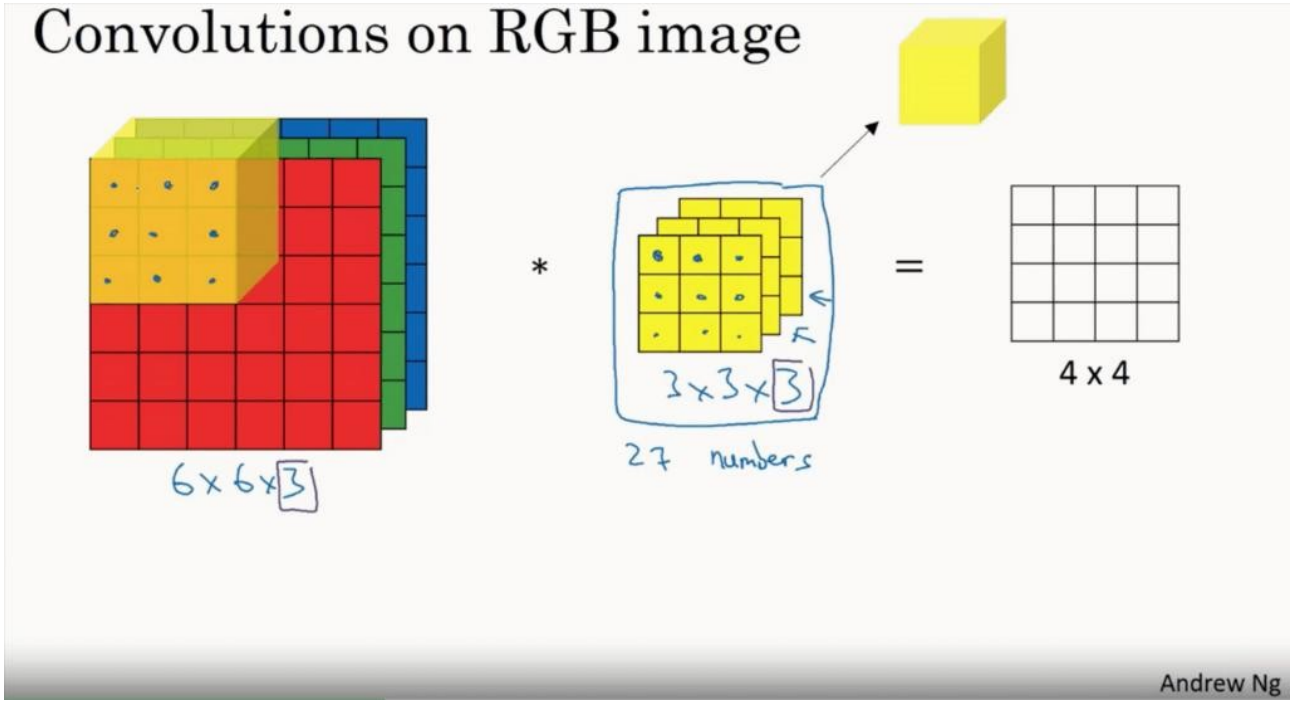


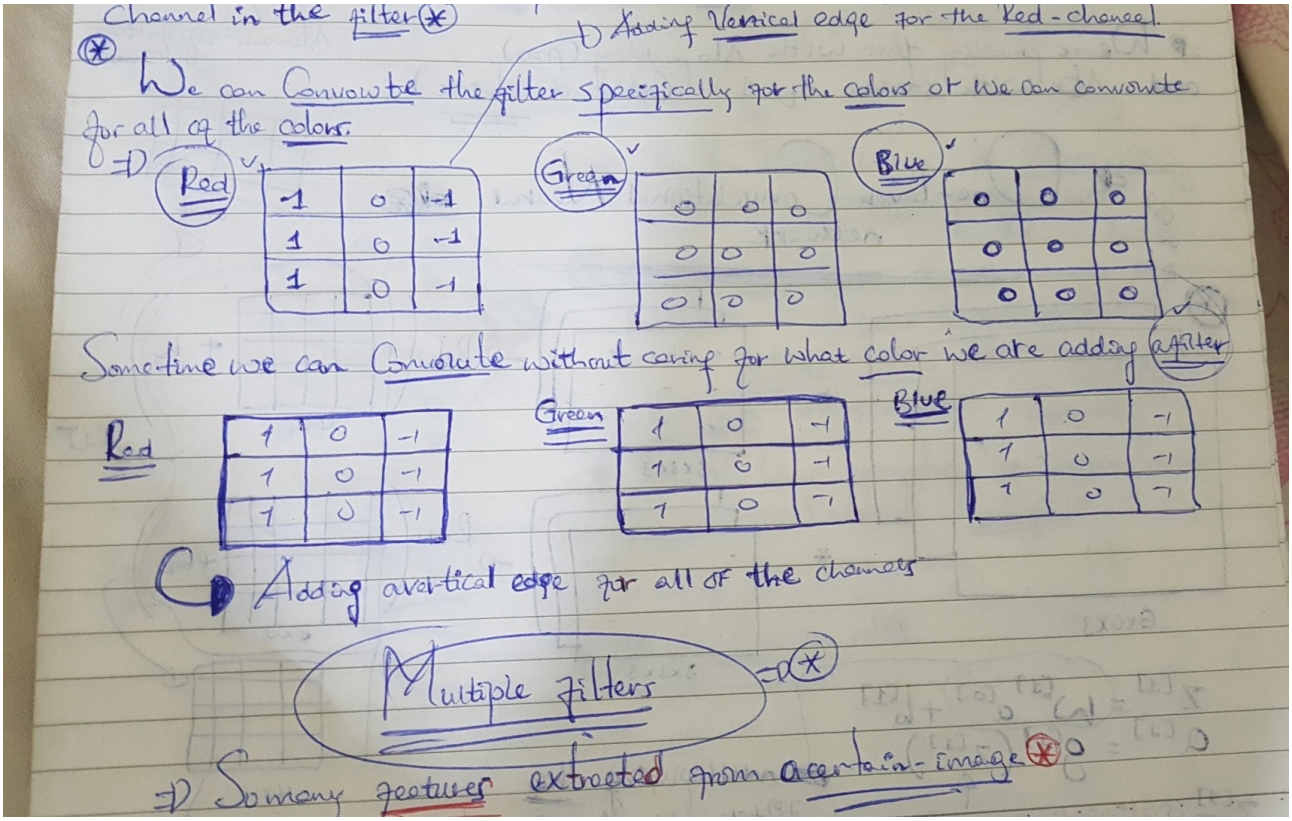
FIGURE 1: Convolving a 6 by 6 by 3 volume with a 3D filter (a 3 by 3 by 3 filter).

An RGB image is represented as a 6 by 6 by 3 volume, where the 3 here responds to the 3 color channels (RGB). In order to detect edges or some other feature in this image, you could convolve the 6 by 6 by 3 volume with a 3D filter (a 3 by 3 by 3 filter) as shown on figure 1, not with a 3 by 3 filter, as we have seen in the previous parts. So the filter itself will also have 3 layers corresponding to the red, green, and blue channels From figure 1, notice that the first 6 is the height of the image, the second 6 is the width, and the 3 is the number of channels. And your filter also similarly has a height, a width, and the number of channels. The number of channels in your image must match the number of channels in your filter. Also, notice that the output image is a 4 by 4 image (or 4 by 4 by 1 image) instead of a 4 by 4 by 3 image.



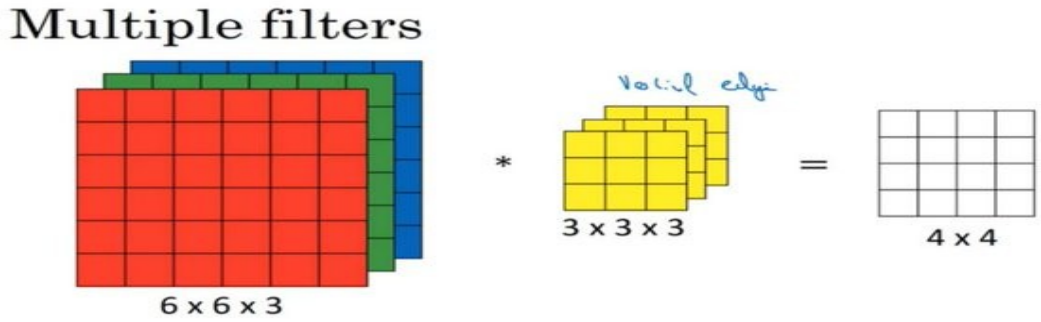
Here's a 6 by 6 by 3 image, and a 3 by 3 by 3 filter once again shown on figure 2. Notice that the 3 by 3 by 3 filter has 27 numbers, or 27 parameters, that's three cubes. So how do you convolve this RGB image the 3D filter? What you do is take each of the 27 numbers of the filter and multiply them with the corresponding numbers from the red, green, and blue channels of the image, i.e take the first 9 numbers from red channel, then the 3 beneath it to the green channel, then the three beneath it to the blue channel, and multiply it with the corresponding 27 numbers that gets covered by this yellow cube show on the left. Then add up all those numbers and this gives you this first number in the output, and then to compute the next output you take this cube and slide it over by one, and again, due to 27 multiplications, add up the 27 numbers, that gives you this next output, do it for the next number over, for the next position over, that gives the third output and so on.

We can convolute the filter specifically for the colors or we can convolute for all of the colors.



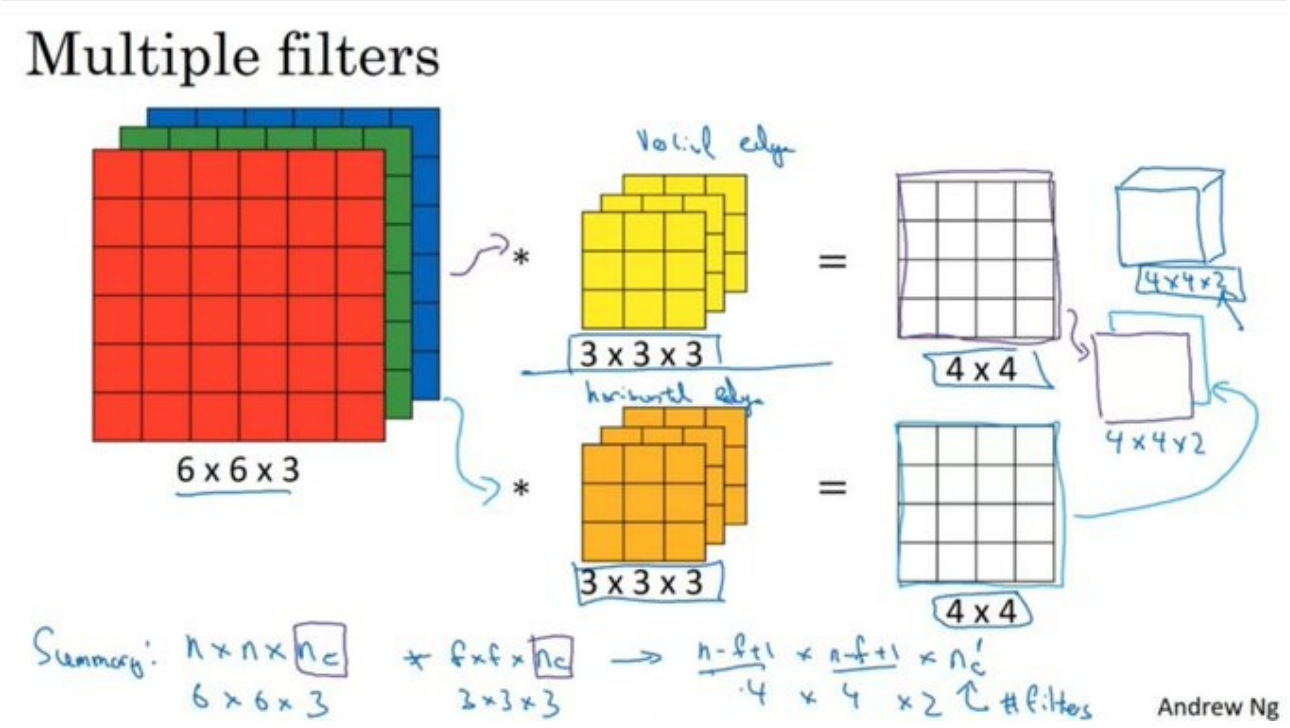
Multiple Filters For RGB Images

Now that you know how to convolve on volumes, what if we didn't just want to detect vertical edges? What if we wanted to detect vertical edges and horizontal edges, maybe 45 degree edges, 70 degree edges as well, but in other words, what if you want to use multiple filters at the same time?



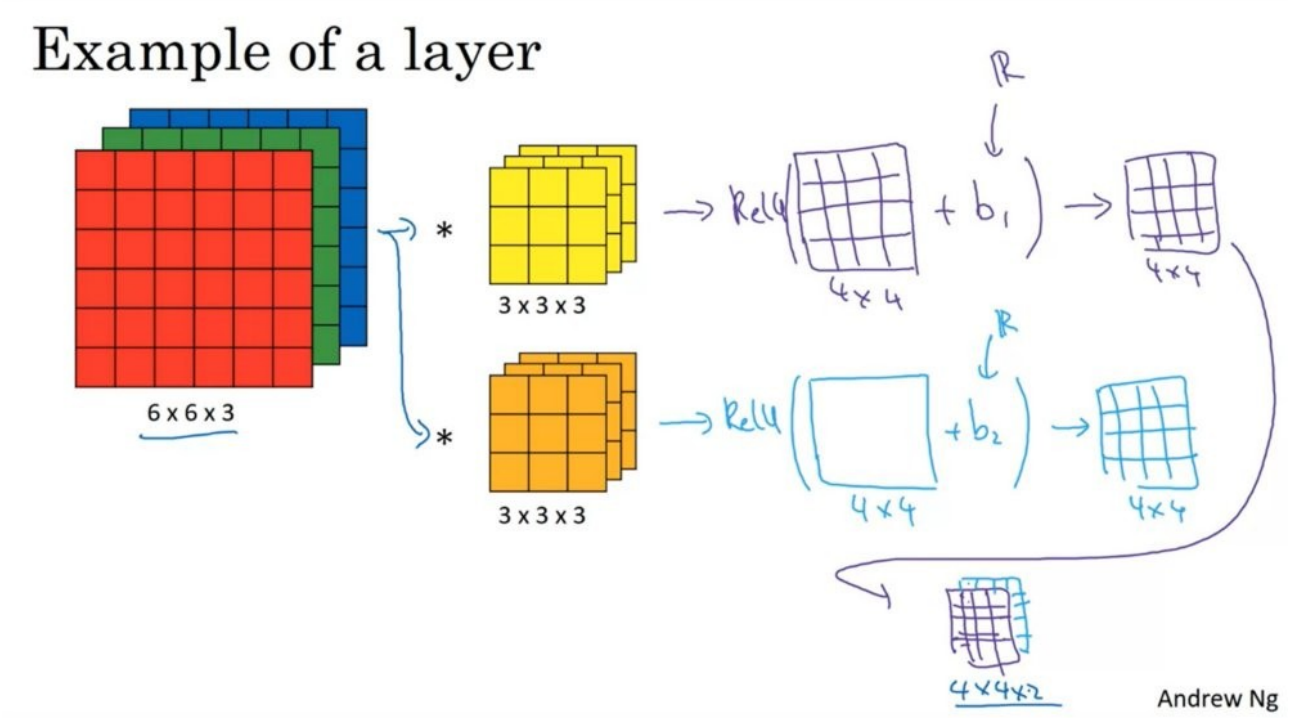
Andrew Ng

So, here's the picture we had from the previous in section 1.1 on figure 3. Let's say, that the yellowish 3 by 3 by 3 filter is maybe a vertical edge detector (or maybe it's intended to detect some other feature).

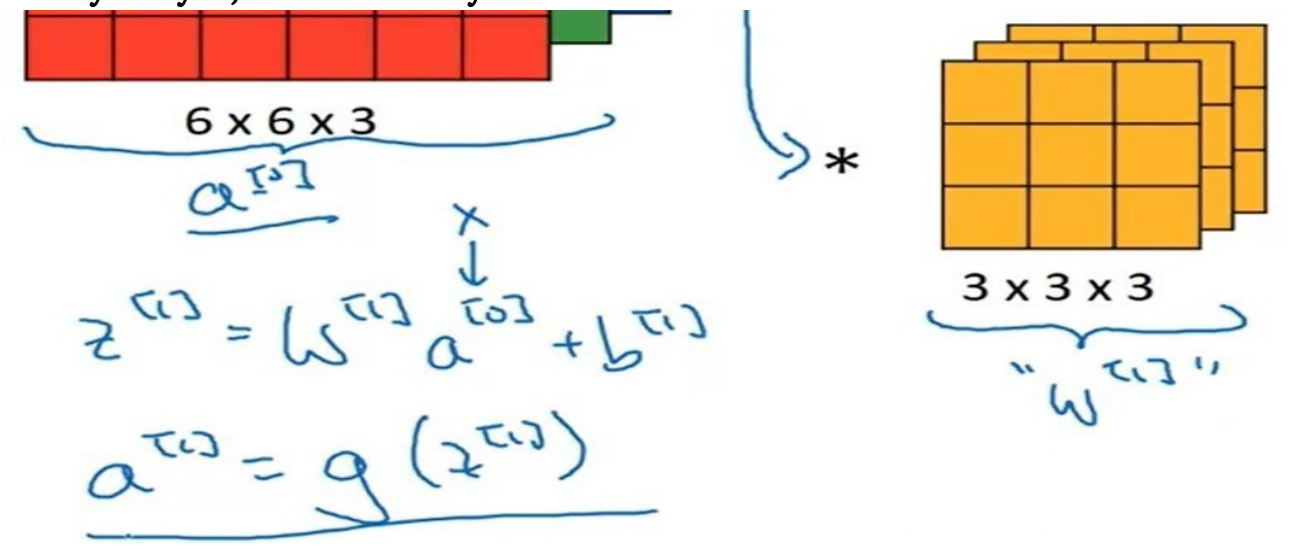


Now, let's say that there's a second 3 by 3 by 3 filter denoted by an orange color, is a horizontal edge detector as shown of figure 4. So, convolving the image with the the yellow filter gives you a 4 by 4 output and convolving with the orange filter gives you a different 4 by 4 output. Now, what we can do is then take these two 4 by 4 outputs and stack them to get a 4 by 4 by 2 output. Notice that if the image was first convolved with the yellow filter , then we take the image & yellow filter's output to be the first one (at the front) and you can then take the image & orange second filter's output and stack it at the back to end up with the 4 by 4 by 2 output image shown on the right in figure 4. Notice that the 2 comes from the fact that we used two different filters.

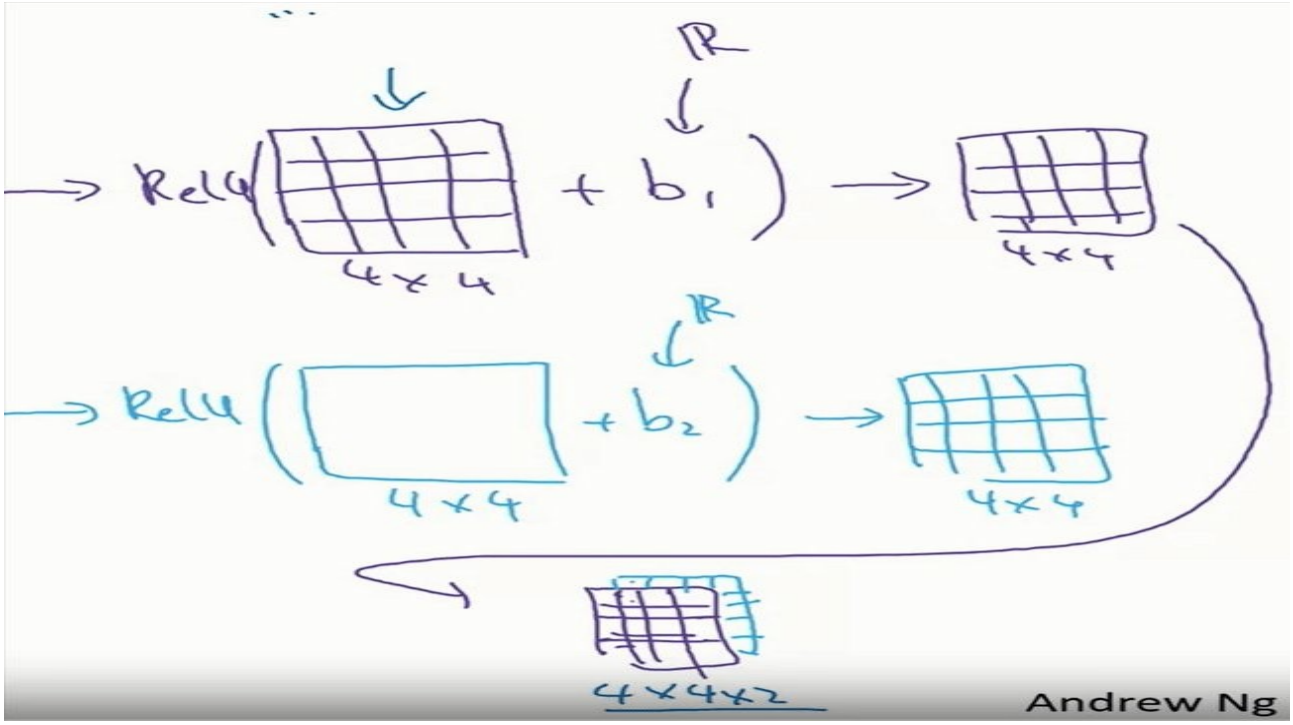
One Layer ConvNet Neural Network



The final thing to turn this into a convolutional neural net layer, is that for each of these we're going to add it bias, so this is going to be a real number. And where python broadcasting, you kind of have to add the same number so every one of these 16 elements. And then apply a non-linearity function , which for this illustration let's say a ReLU (Rectifier Linear Unit) non-linearity, and this gives you a 4 by 4 output, all right? After applying the bias and the non-linearity. We also apply the ReLU non-linearity to the the orange filter (after adding the bias) and get a different 4 by 4 output. Then same as we did before, if we take this and stack it up as follows, so we ends up with a 4 by 4 by 2 outputs. Then this computation where you come from a 6 by 6 by 3 to a 4 by 4 by 2, this is one layer of a convolutional neural network.

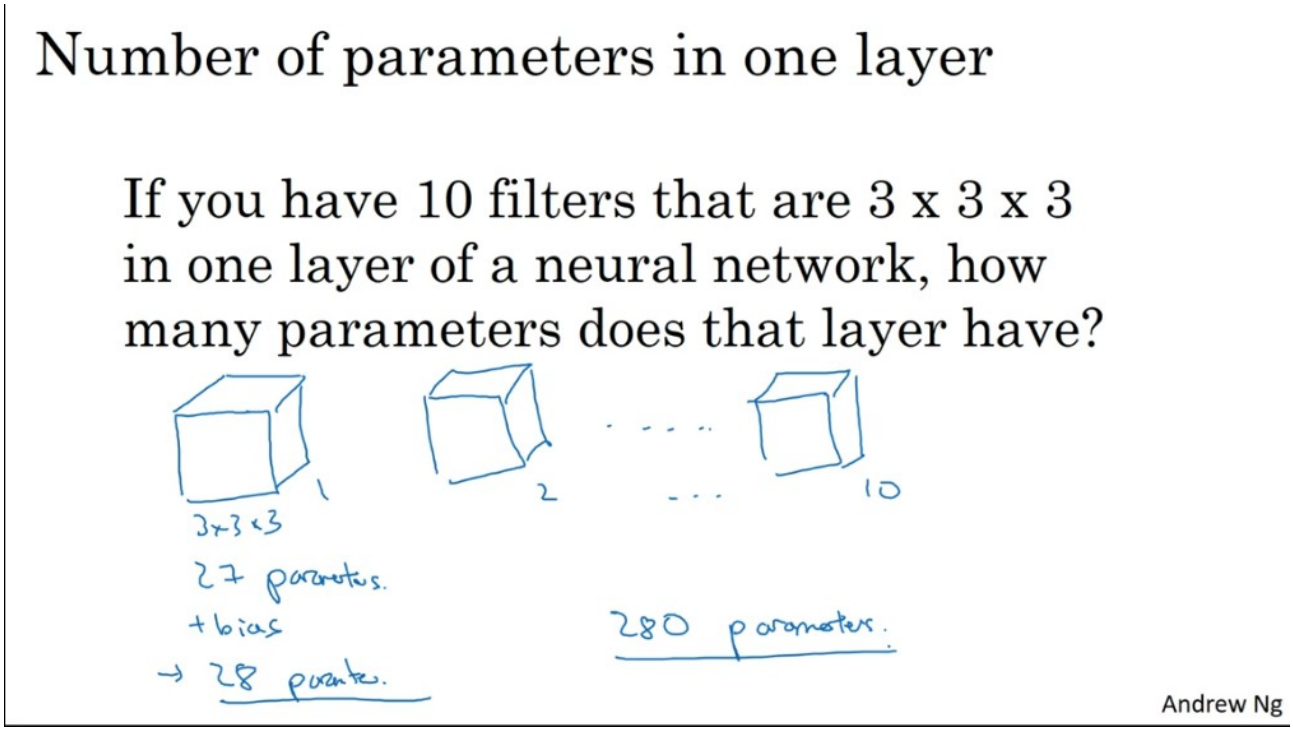


So to map this back to one layer of forward propagation in the standard neural network, in a non-convolutional neural network. Remember that one step of forward propagation was $z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$. And you apply the non-linearity to get a[1], so that's $g(z^{[1]})$ just as underlined in figure 6. By analogy, The input image here is $a^{[0]}$, this is $x^{[3]}$. The filters here, play a role similar to $w^{[1]}$. So you're really computing a linear function to get the 4 by 4 output matrix which is $a^{[1]} = g(z^{[1]})$.

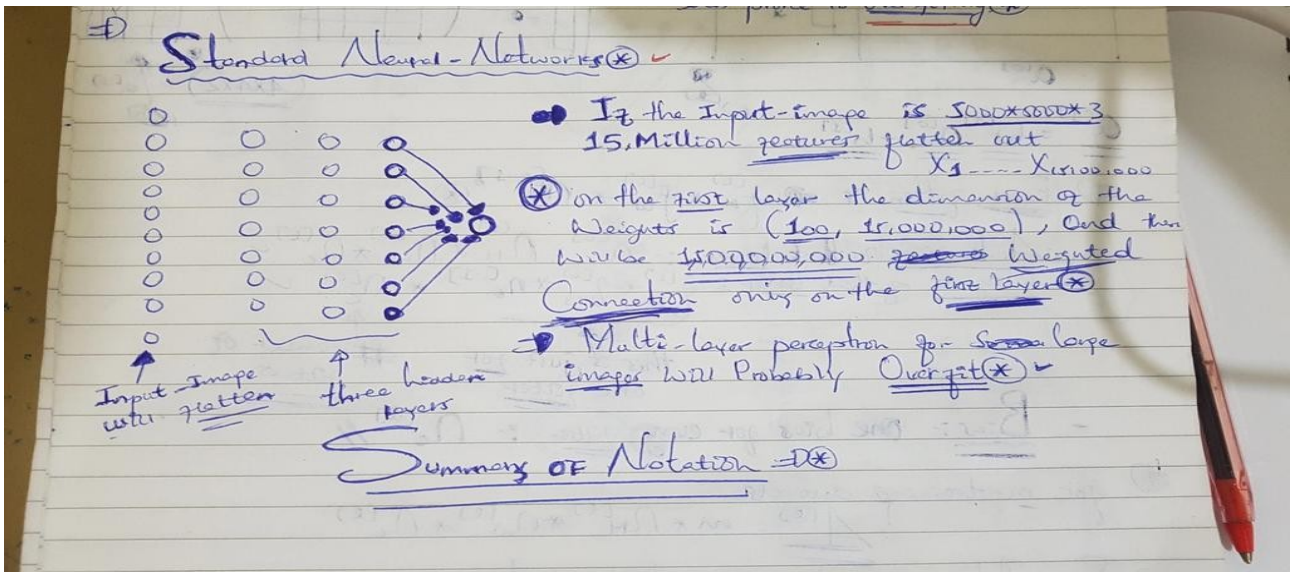


The 4 by 4 matrix output matrix, the output of the convolution operation, plays a role similar to $w^{[1]} a^{[0]}$. and the output image plays a role similar to $a^{[1]} = g(z^{[1]})$, where g is the non-linearity function such as the ReLU and $z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$. So we are really just going from $a^{[0]}$ to $a^{[1]}$, where $a^{[0]}$ is the 6 by 6 by 3 input image and $a^{[1]}$ is the 4 by 4 by 2 output image.

Number Of Parameters



No matter how big the input image is, the input image could be 5000*5000*3 but the number of parameters we have is still remains 280 weights and we can use 10 filters to detect so many features in the Image. These property makes Convnet less prone to overfitting. The two reasons of Convnet to have a decreased number of parameters is 1) parameter sharing 2) partially connected layers.



Summary Of Notations

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

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

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

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

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

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

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

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

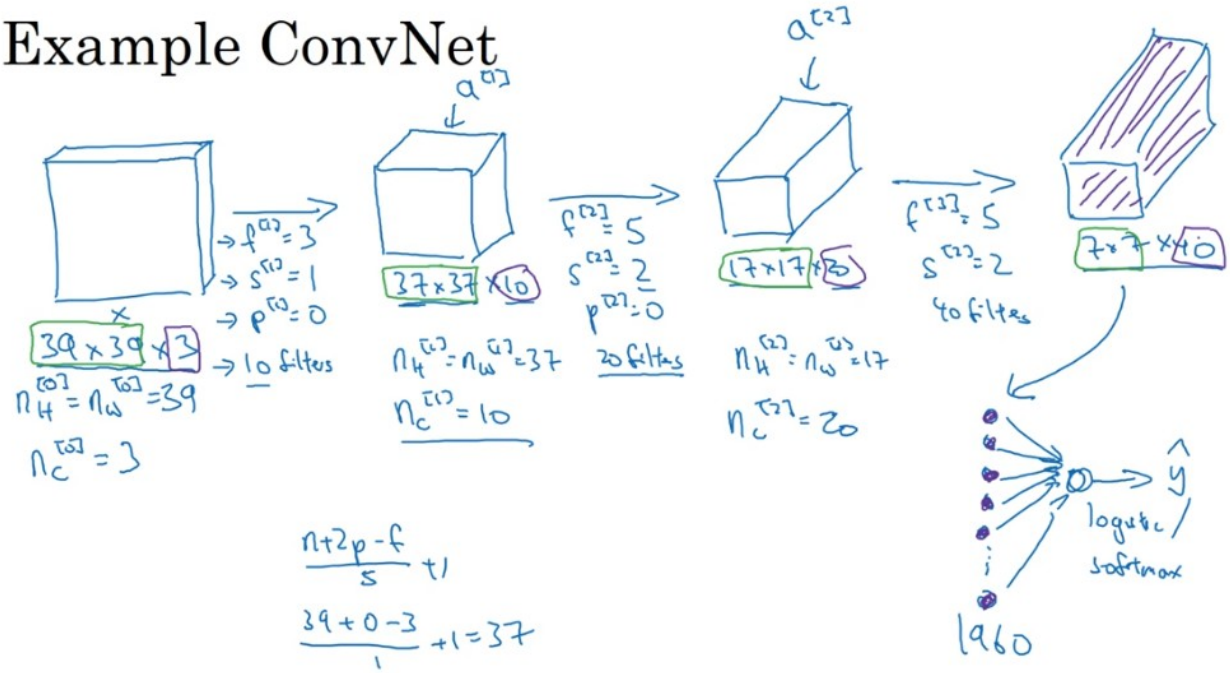
bias: $n_c^{[l]} - (1,1,1,n_c^{[l]})$ ← #filters in layer l.

$n_c \times n_H \times n_W$

Andrew Ng

Multi Layer Convolutional Neural Network

Example ConvNet



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

$$\frac{39+0-3}{1} + 1 = 37$$

Andrew Ng

Types of layers in Convolutional Neural Networks

1) ConvNet
2) Pooling Layers
3) Fully Connected Neural Networks

Pooling Layers

Convolutional layers in a convolutional neural network summarize the presence of features in an input image.

A problem with the output feature maps is that they are sensitive to the location of the features in the input. One approach to address this sensitivity is to down sample the feature maps. This has the effect of making the resulting down sampled feature maps more robust to changes in the position of the feature in the image, referred to by the technical phrase "local translation invariance."

Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.

Convolutional layers in a convolutional neural network systematically apply learned filters to input images in order to create feature maps that summarize the presence of those features in the input.

Convolutional layers prove very effective, and stacking convolutional layers in deep models allows layers close to the input to learn low-level features (e.g. lines) and layers deeper in the model to learn high-order or more abstract features, like shapes or specific objects.

A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a

different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image. Why are we caring this much for the small movements in the input image ? Because if the feature map records the precise positions of the features CNN's won't gain the flexibility to recognize an image so if the object in the image has different lighting or orientation, distortion , CNN will be give us wrong prediction.

A common approach to addressing this problem from signal processing is called down sampling. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task.

Down sampling can be achieved with convolutional layers by changing the stride of the convolution across the image. A more robust and common approach is to use a pooling layer.

A pooling layer is a new layer added after the convolutional layer. Specifically, after a nonlinearity (e.g. ReLU) has been applied to the feature maps output by a convolutional layer; for example the layers in a model may look as follows:

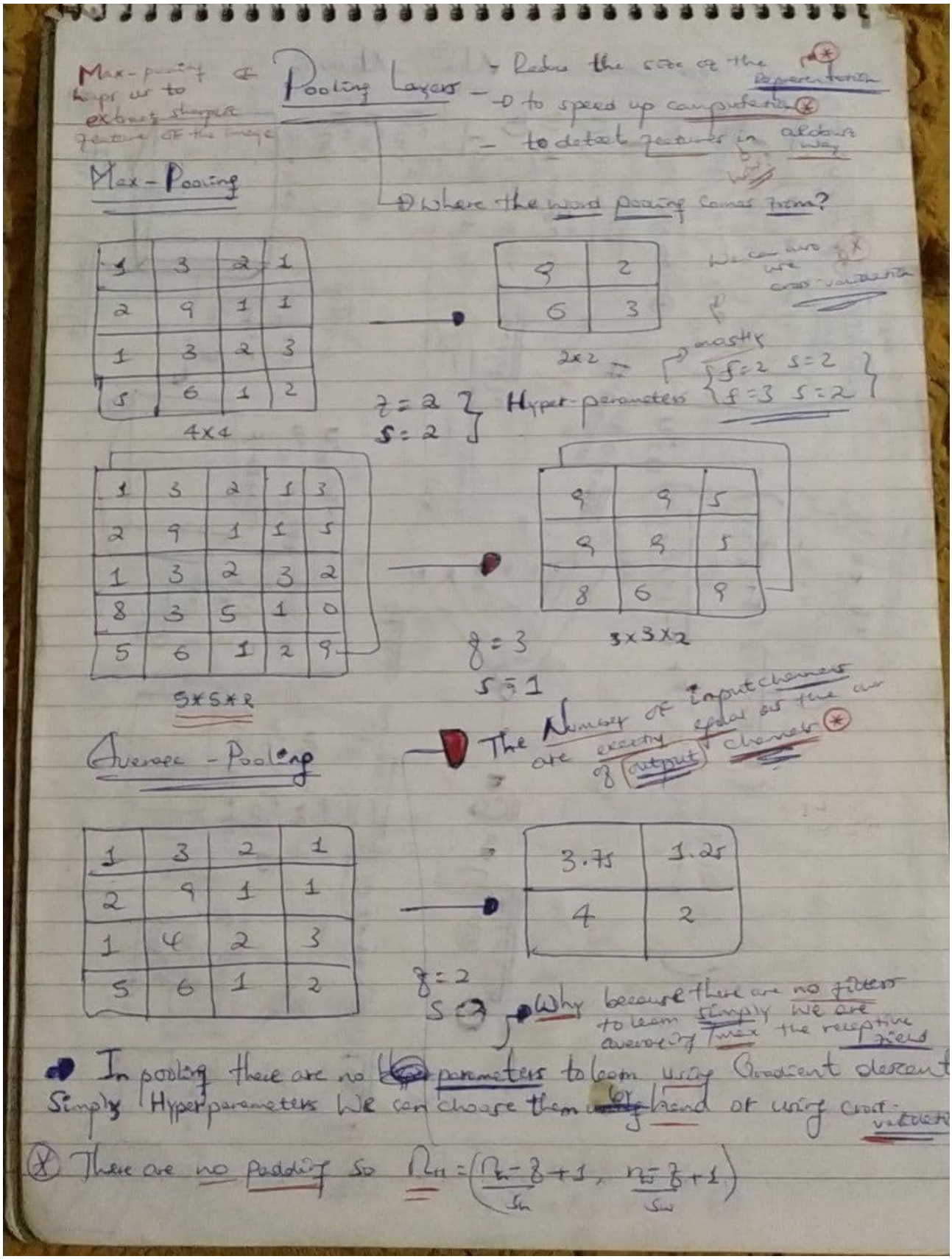
1. Input Image
2. Convolutional Layer
3. Nonlinearity
4. Pooling Layer

Pooling involves selecting a pooling operation, much like a filter to be applied to feature maps. The size of the pooling operation or filter is smaller than the size of the feature map; specifically, it is almost always 2×2 pixels applied with a stride of 2 pixels.

This means that the pooling layer will always reduce the size of each feature map by a factor of 2, e.g. each dimension is halved, reducing the number of pixels or values in each feature map to one quarter the size. For example, a pooling layer applied to a feature map of 6×6 (36 pixels) will result in an output pooled feature map of 3×3 (9 pixels).

The pooling operation is specified, rather than learned. Two common functions used in the pooling operation are:

●Average Pooling: Calculate the average value for each patch on the feature map.
●Maximum Pooling (or Max Pooling): Calculate the maximum value for each patch of the feature map.



The result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input.

They are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation.

In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. How ?

Achieving translation invariance in Convolutional Neural Networks:

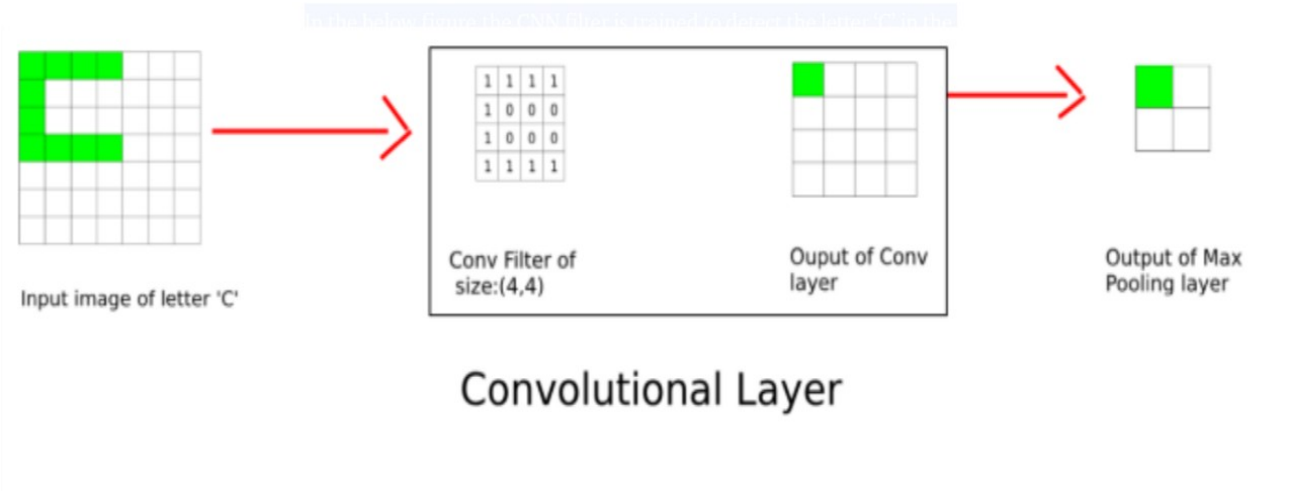
First ,let's give a more formal definition of translation invariance:-
Assume a transformation T, which shifts the position of the target in the input image x by some amount ,

then a Neural Network having translation invariance would satisfy the equation
Neural Network(T(x))=Neural Network(x)

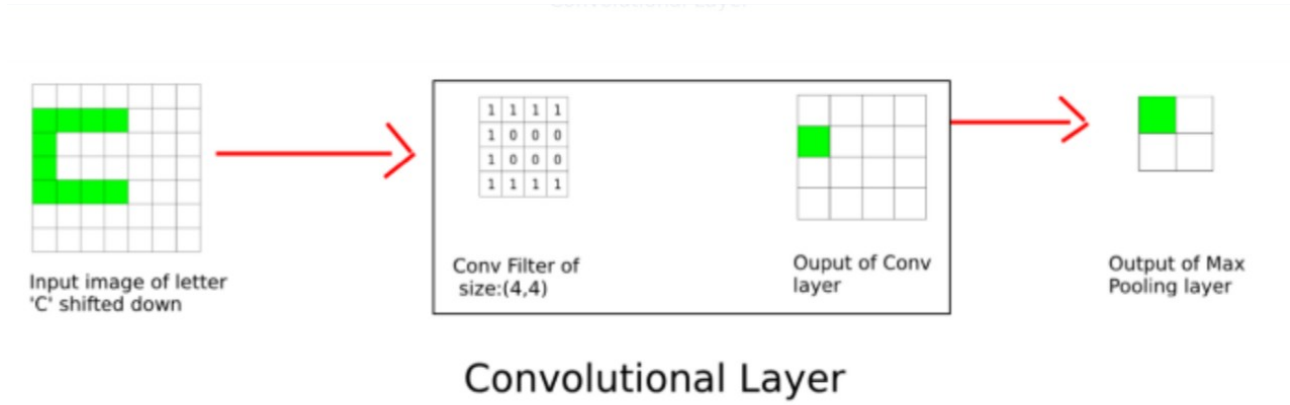
which means , if the Neural Network has translation invariance, the output of the Neural Network will not change when the transformation T is applied.

This translation invariance in the convolutional Neural Network is achieved by a combination of convolutional layers and max pooling layers.
Firstly, the convolutional layer reduces the image to a set of features and their respective positions. Then the max pooling layer takes the output from the convolutional layer and reduces its resolution and complexity.
It does so by outputting only the max value from a grid. So the information about the exact position of the max value in the grid is discarded.
This is what imbues the CNN's with translation invariance.

In the below figure,the CNN filter is trained to detect the letter 'C' in the input image. So lets input an image of letter 'C'.



Now lets shift the letter 'C' down in the image by 1 pixel length.



Comparing the output in the 2 cases, you can see that the max pooling layer gives the same result. The local positional information is lost. This is translation invariance in action. This means that if we train a Convolutional Neural Network on images of a target, the CNN will automatically work for shifted images of that target as well.

But if u think about it … if we had shifted the letter 'C' more severely, the output would have been different.

To make the translation invariance more effective, we could use more convolutional + max pooling layers, as successive max pooling layers

compound the effect of translation invariance.(try to think why this is the case).

Max Pooling Layer

Maximum pooling, or max pooling, is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map.

The results are down sampled or pooled feature maps that highlight the most present feature in the patch, not the average presence of the feature in the case of average pooling. This has been found to work better in practice than average pooling for computer vision tasks like image classification.

Average Pooling Layer

On two-dimensional feature maps, pooling is typically applied in 2×2 patches of the feature map with a stride of (2,2).

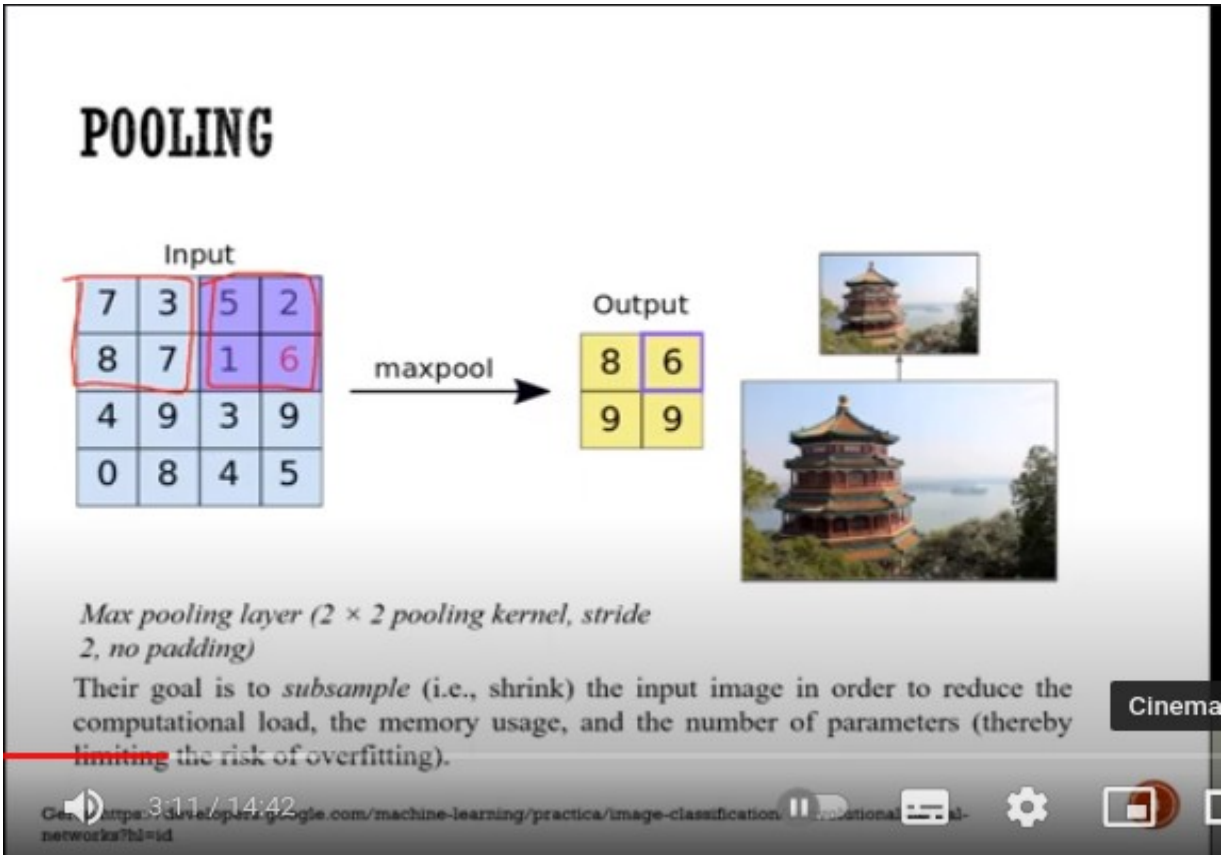
Average pooling involves calculating the average for each patch of the feature map. This means that each 2×2 square of the feature map is down sampled to the average value in the square.

Notes in pooling :-

- It reduces the size of the representation
- To speed up the computation
- To detect features in a Robust ways.
- In pooling layers there are no parameters to learn using gradient descent simply there are hyper parameters , we can choose them using by hand or using cross validation.
- There are no padding in pooling layers
- We always apply pooling after adding non – linearity
- The main advantage of pooling is Transitional Invariance so that CNN's gain the flexibility to recognize an image even if the object in the image has different lighting or orientation , distortion.

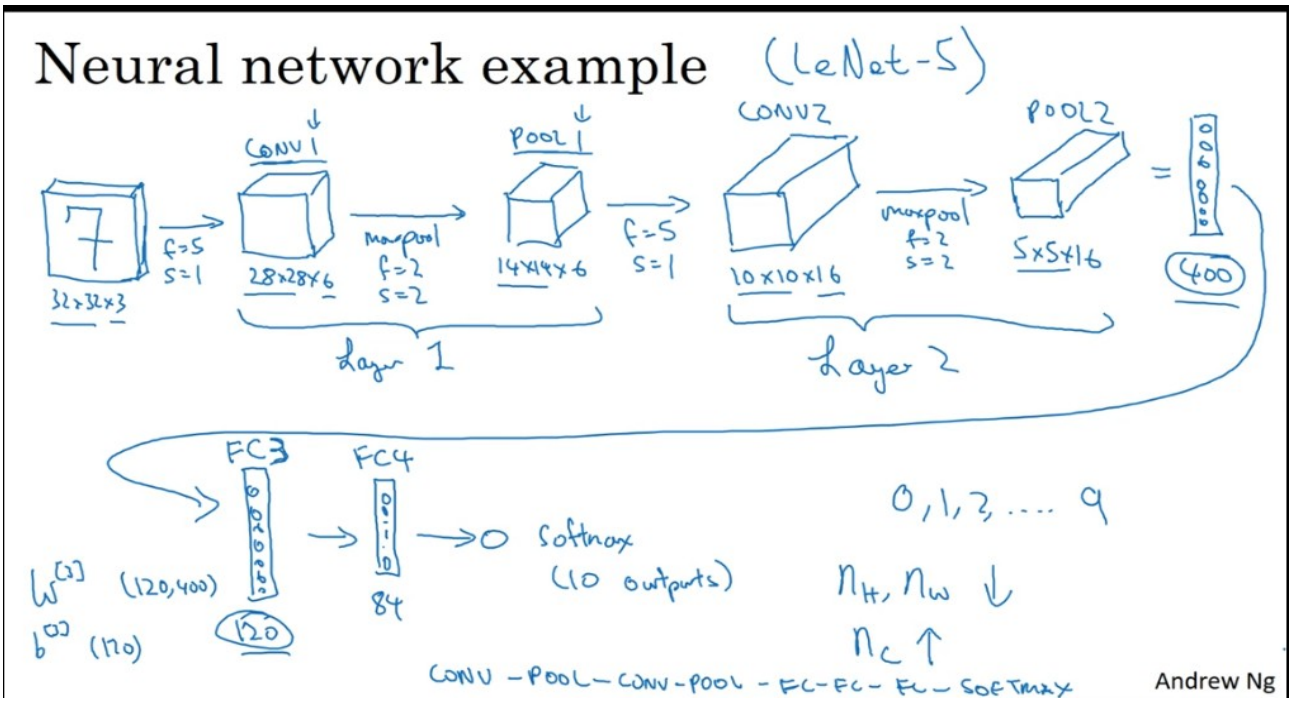
- Pooling layers reduce the size , so that they will be computationally effective and since summarize the feature maps , they are more robust.

- when we actually reducing the size , there will not be that much loss in the information but since we are reducing the size there will be a resolution loss.



so when we actually shrink the input image , we reduce the computational load and the memory usage so the number of parameters passes to the next layer will be very much low , these helps our model to be less proned to over fitting.

Full Connected Convolutional Neural Network



The Activation shape , Activation size and number of parameters of the above fully connected neural networks.

Neural network example

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

Andrew Ng

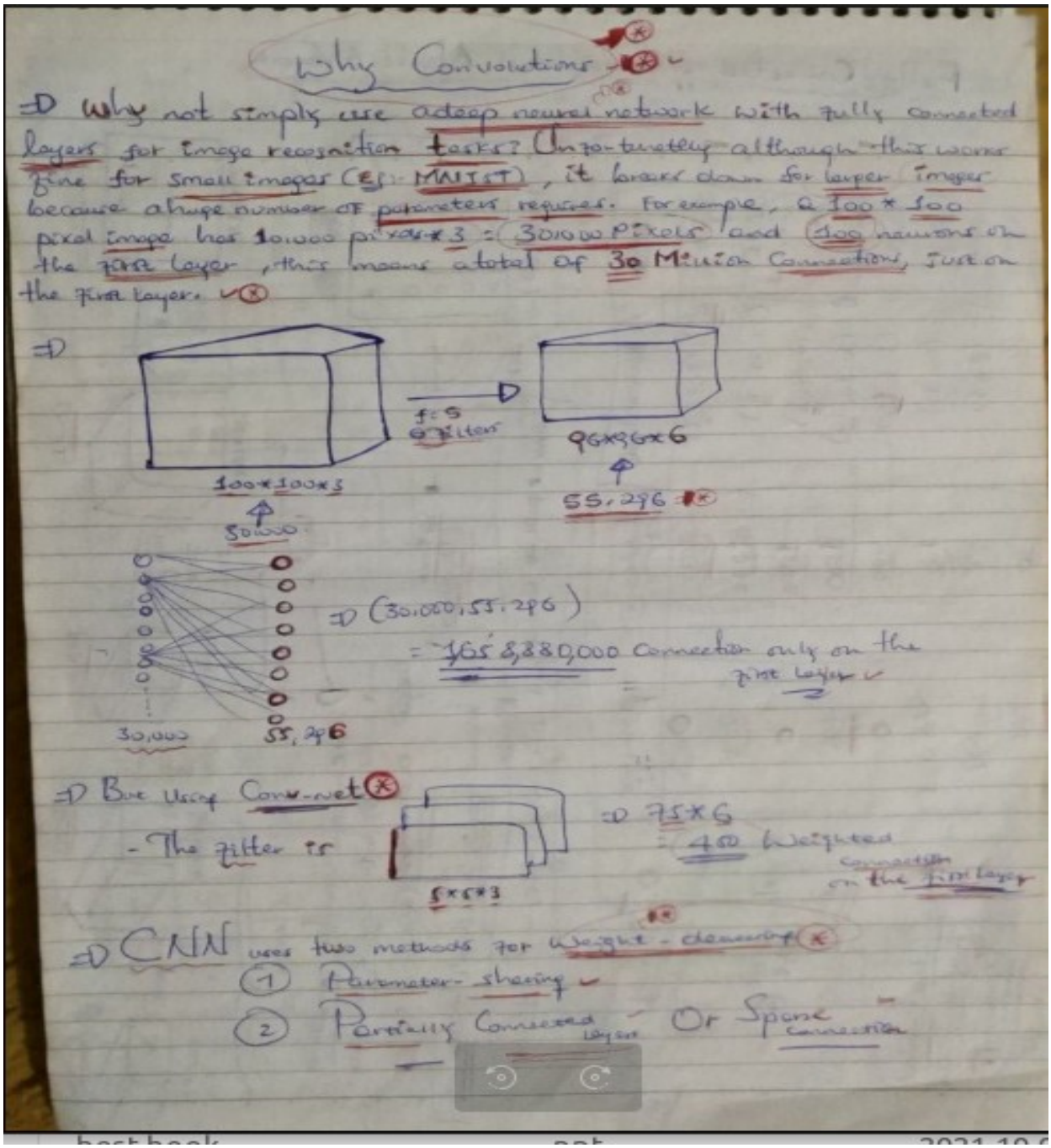
Why Convolutions

Why not simply use a deep neural network with fully connected layers for image recognition tasks ? Unfortunately although this works fine for small images (Example:- MNIST) , it breaks down for larger images because a huge number of parameter requires. For example :- a 100 * 100 pixel image has 10,000 * 3 = 30 ,000 pixels and 100 neurons on the first layer , this means a total of 30 million connections , just on the first layer.

CNN uses two methods for weight decreasing :-
1) Parameter sharing
2) Partially Connected or Sparse Connection

1) Parameter Sharing :- A feature detector (Such as vertical edge detector) that is useful in one part of the image is probably useful in another part of the image.

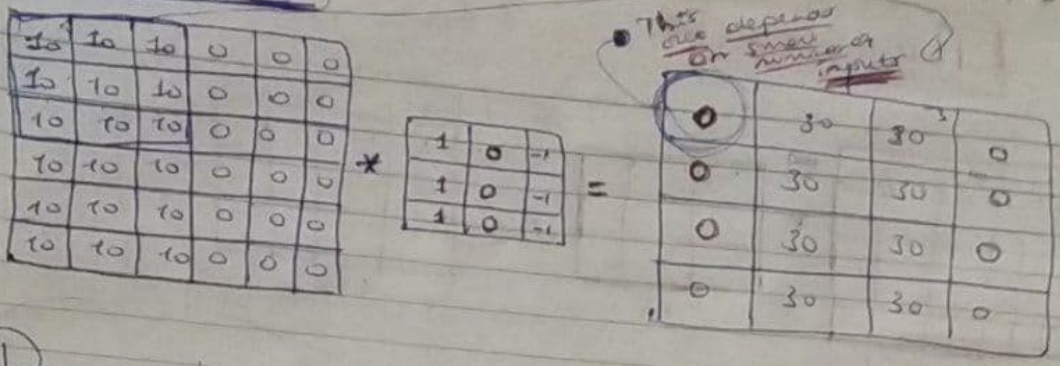
Parameter Sharing is a sharing of weights to all neurons in a particular feature map. This helps to reduce the number of parameters in the whole system and makes the computation more efficient.



How does parameter sharing reduces the number of parameters ?

- The fact that all neurons in the feature map share the same parameters drastically reduces the number of parameters in the model , once the CNN has learned to recognize a pattern in one location , it can also recognize it in another location. In contrast , once a regular DNN has learned to recognize a pattern in one location , it can recognize it only in that particular location.

Why Convolutions



- This is due to small number of inputs

1) **Parameter-sharing** :- A feature detector (such as Vertical edge detector) that is useful in one part of the image is probably useful in another part of the image.

↳ Parameter sharing is sharing of weights to all neurons in a particular feature map → This helps to reduce the number of parameters in the whole system and makes the computation more efficient.

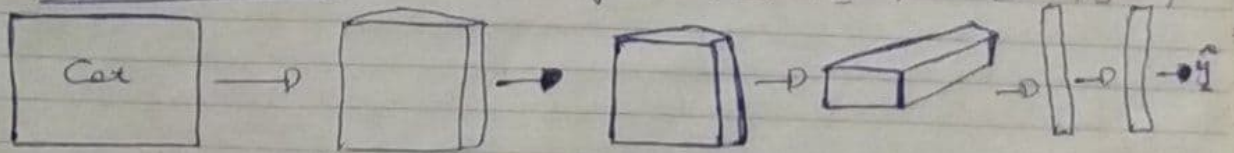
How does parameter sharing Reduces the Number of Parameters ?

- The fact that all neurons in the feature map share the same parameter drastically reduces the number of parameters in the model; once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In Contrast, Once a regular DNN has learned to recognize a pattern in one location, it can Recognize it only in that particular location.

Sparsity of connections :- In each layer, each output value depends on a small number of inputs.

↳ partially connected * Unlike to Deep neural networks, the output values depends on the entire previous inputs.

Putting it together :- Training set $(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$



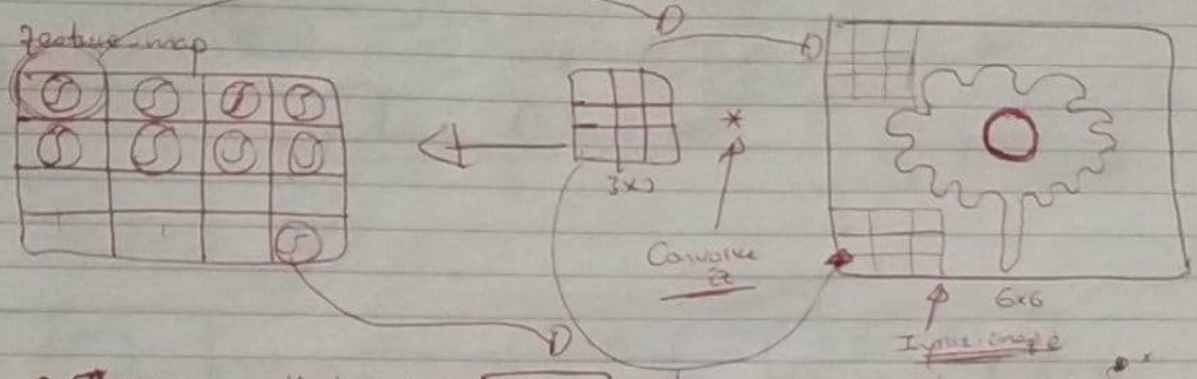
↳ So many parameter

Cost $J = \frac{1}{m} \sum_{i=1}^{m} loss(\hat{y}^{(i)}, y^{(i)})$ ↳ use gradient descent to reduce J.

Share the matrix of parameters across all of the neurons in the feature map.

More about Parameter sharing ⊛ ⊛

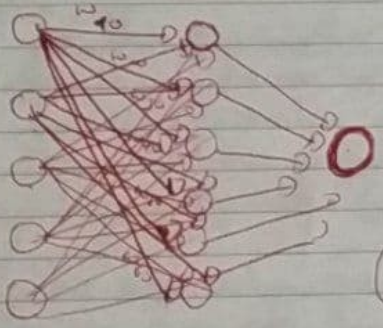
- Share the Matrix of parameters across all of the neurons in the feature map. ⊛



- This means that all the Neurons in the feature map Share the same parameters
- This make computationally more efficient
- Dramatically Reduce the number of parameters

This is ⊛

Unlike to Fully connected Neural-Networks, once learned to recognize a pattern in one location, it can only Recognize it only in that particular location



$$Z_1^{[1]} = W_{10}X_0 + W_{11}X_1 + W_{12}X_2 + W_{13}X_3 + W_{14}X_4$$

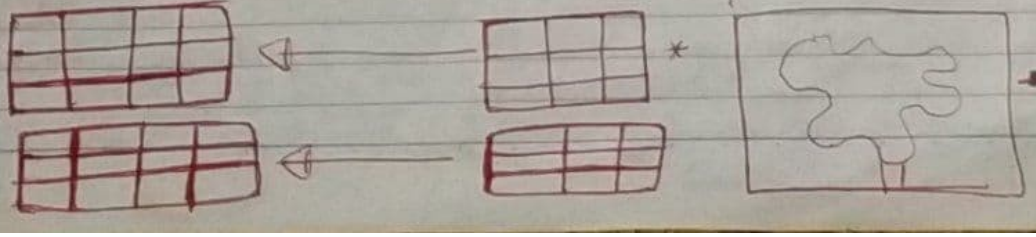
$$Z_2^{[1]} = W_{20}X_0 + W_{21}X_1 + W_{22}X_2 + W_{23}X_3 + W_{24}X_4$$

$$Z_3^{[1]} = W_{30}X_0 + W_{31}X_1 + W_{32}X_2 + W_{33}X_3 + W_{34}X_4$$

- ~~Three~~ different Neurons using different parameters ⊛

- There are computationally expensive ⊛
  : The Number of parameters are high, which possibly Overfit the model ⊛

- In the CNN ⊛ we will extract the same features at every position of the image



This means that all of the neurons in the feature map share the same parameters

- This makes to be computationally more efficient
- Dramatically reduce the number of parameters

Unlike to fully connected neural networks once learned to recognize a pattern in one location , it can only recognize it only in that particular location.

- This makes it Computationally expensive
- The number of parameters will be high , which probably over fit the image.

Sparsity Of Connections (Partially Connected) :- In each layer , each output values depends on a small number of inputs. In contrast , Deep Neural Networks , the output values depends on the entire previous inputs.

CNN Architectures

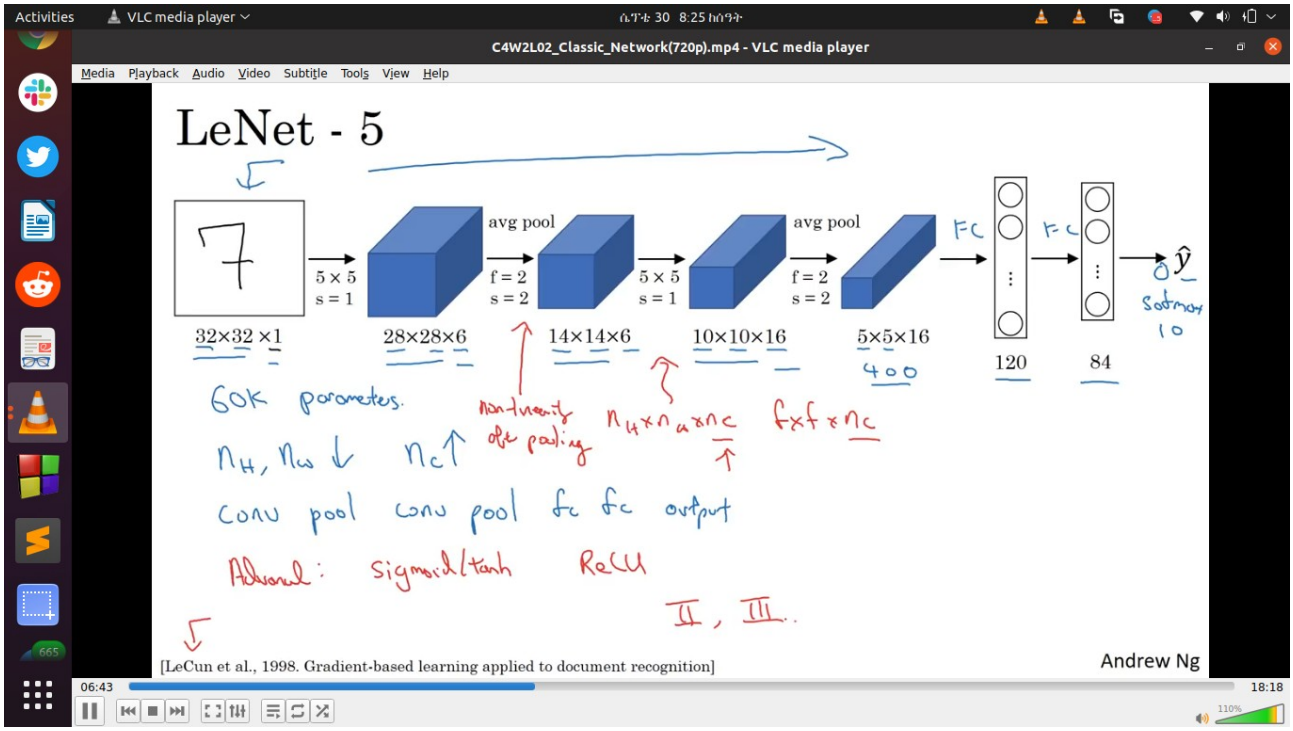
LeNet-5

In 1989, Yann LeCun et al. at Bell Labs initially applied the backpropagation rule to all sensible applications, He combined a convolutional neural network trained by backpropagation algorithms to scan written numbers which he later used in detecting different handwritten postcode numbers(Zip Codes) which were given by the USA communicating Service. This was the elementary example of what later came to be referred to as LeNet.

LeNet-5 CNN architecture is made up of 7 layers(counting the conv and the pooling layers separate). The layer composition consists of 3 convolutional layers, 2 subsampling layers(Average Pooling) and 2 fully connected layers.

The first layer is the input layer — this is generally not considered a layer of the network as nothing is learnt in this layer. The input layer is built to take in *32x32,*and these are the dimensions of images that are passed into the next layer. Those who are familiar with the MNIST dataset will be aware that the MNIST dataset images have the dimensions *28x28. To* get the MNIST images

dimension to the meet the requirements of the input layer, the ***28x28*** images are padded.



The LeNet-5 architecture utilizes two significant types of layer construct: convolutional layers and subsampling layers.

- Convolutional Layers
- Sub Sampling layers (Average Pooling)

Within the research paper and the image below, convolutional layers are identified with the ***'Cx'***, and subsampling layers are identified with ***'Sx'***, where ***'x'*** is the sequential position of the layer within the architecture. ***'Fx'*** is used to identify fully connected layers. This method of layer identification can be seen in the image above.

The official first layer convolutional layer ***C1*** produces as output 6 feature maps, and has a kernel size of ***5x5.*** The kernel/filter is the name given to the window that contains the weight values that are utilized during the convolution of the weight values with the input values. ***5x5*** is also indicative of the local receptive field size each unit or neuron within a convolutional layer.

The dimensions of the six feature maps the first convolution layer produces are ***28x28.***

A subsampling layer ***'S2'*** follows the ***'C1'*** layer'. The ***'S2'*** layer halves the dimension of the feature maps it receives from the previous layer; this is known commonly as downsampling.

The ***'S2'*** layer also produces 6 feature maps, each one corresponding to the feature maps passed as input from the previous layer.

Below is a table that summarises the key features of each layer:

LeNet CNN Structure

Layer	Layer Type	Feature Maps	Kernel/Filter or Units	Input / Feature Map size	Trainable parameters	Connections	Strides	Activation Function
Input	Image	-	-	32x32	-	-	-	-
C1	Convolution	6	5x5	28x28	156	122,304		Hyperbolic tangent (tanh)
S2	Sub Sampling	6	2x2	14x14	12	5,880		Sigmoid
C3	Convolution	16	5x5	10x10	1,516	151,600		Hyperbolic tangent (tanh)
S4	Sub Sampling	16	2x2	5x5	32	2,000		Sigmoid
C5	Convolution	120	5x5	1x1	48,120			Hyperbolic tangent (tanh)
F6	Fully Connected	-	-	84	10,164			Hyperbolic tangent (tanh)
Output	Fully Connected	-	-	10				Softmax

LeNet-5 Architecture features by Author

- The goal of LeNet-5 was to recognize handwritten digits , and it was trained by grayscale images that's why it is **32*32*1.**

- LeNet architecture used average pooling instead of max pooling , back then when this paper was written people didn't use padding or they always use valid convolutions , that's why the size shrinks.

- The architecture got about **60k** parameters , and it was small , today's CNN use **10million** to **100** million parameters.

- In this architecture nH and nW is decreasing while nc is increasing , which is computationally good and detecting complex features , but the problem is we can't train deep conv nets if the image is shrinking every layer , that's why we are using padding.

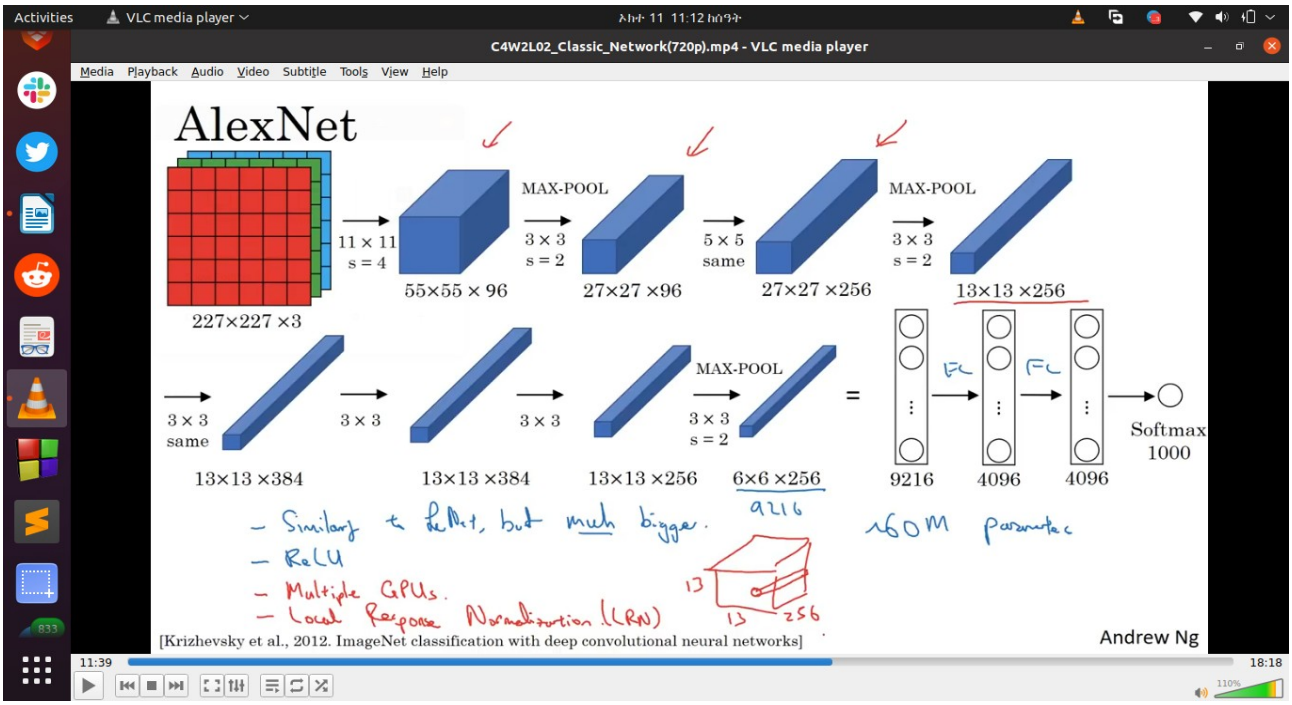
- Back then they were using sigmoid and tanh non linear activation functions. and the Non linearity were applied after the pooling layer.

Why is it called LeNet-5

The network has 5 layers with learnable parameters and hence named LeNet-5. It has three sets of convolution layers with a combination of average pooling. After the convolution and average pooling layers, we have two fully connected layers. We are calling conv nets with the pooling layers as a single layer , that is why we are ending up to the number 5.

AlexNet:- The architecture consists of eight layers: five convolutional layers and three fully-connected layers. a convolutional layer followed by an activation function followed by a max pooling operation, (sometimes the pooling operation is omitted to preserve the spatial resolution of the image).But this isn't what makes AlexNet special; these are some of the features used that are new approaches to convolutional neural networks:

- •ReLU Nonlinearity. AlexNet uses Rectified Linear Units (ReLU) instead of the tanh function, which was standard at the time. ReLU's advantage is in training time; a CNN using ReLU was able to reach a 25% error on the CIFAR-10 dataset six times faster than a CNN using tanh.
- •Multiple GPUs. Back in the day, GPUs were still rolling around with 3 gigabytes of memory (nowadays those kinds of memory would be rookie numbers). This was especially bad because the training set had 1.2 million images(Image Net) AlexNet allows for multi-GPU training by putting half of the model's neurons on one GPU and the other half on another GPU. . These splited up two different GPU's had a way to communicate to each others. This means that the neurons in the feature maps will be splited up to two GPU'S so our first layer forexample will be (55*55*48) *2 , one for each GPU's.
- • The architecture is almost similar to the LeNet-5 but much bigger parameters , which is about 60million.

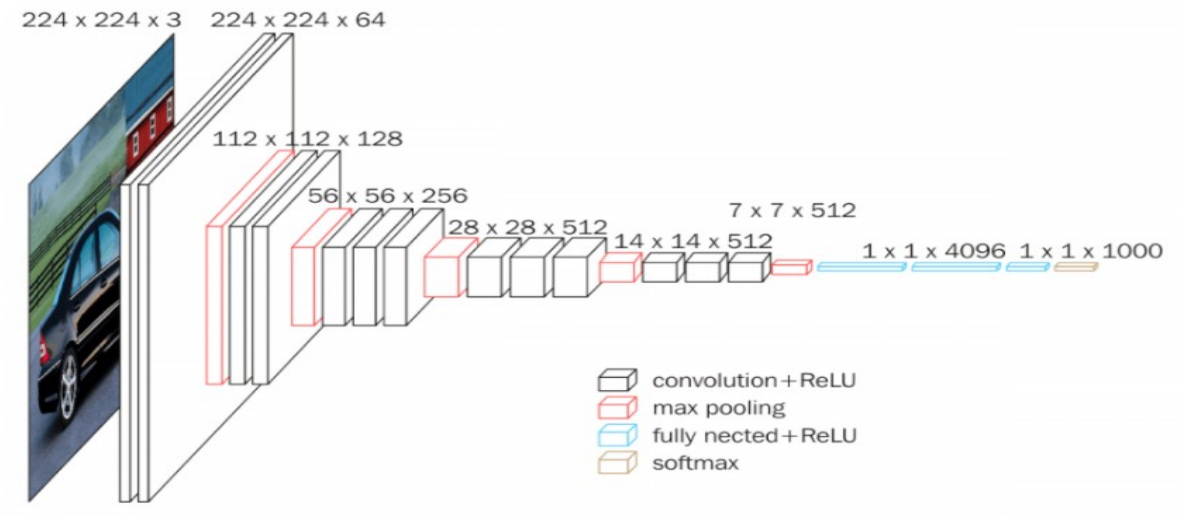


The Overfitting Problem. AlexNet had 60 million parameters, a major issue in terms of overfitting. Two methods were employed to reduce overfitting:

Data Augmentation. The authors used label-preserving transformation to make their data more varied. Specifically, they generated image translations and horizontal reflections, which increased the training set by a factor of 2048.

Dropout. This technique consists of "turning off" neurons with a predetermined probability (e.g. 50%). This means that every iteration uses a different sample of the model's parameters(because at every iteration we will select random neurons to be dropped out so the dropped out neuron's will terminate their weighted connection to the front and to their back) which forces each neuron to have more robust features that can be used with other random neurons. However, dropout also increases the training time needed for the model's convergence.

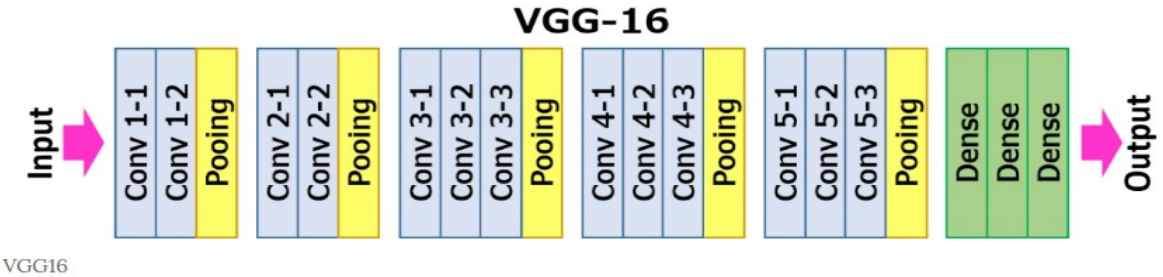
VGG16 – Convolutional Network for Classification and Detection



VGG means:- Visual Geometry Group at University of Oxford. The number 16 in the VGG represents the number of layers.

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition". The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous model submitted to SVRC-2014. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU's.

They said instead of having so many hyper-parameters , Let's use a simpler networks where you focus on having conv layers that are just 3*3 filters with a stride of 1 and same padding and making all of your max - pooling 2*2 with the stride of 2 so that we can go deeper.

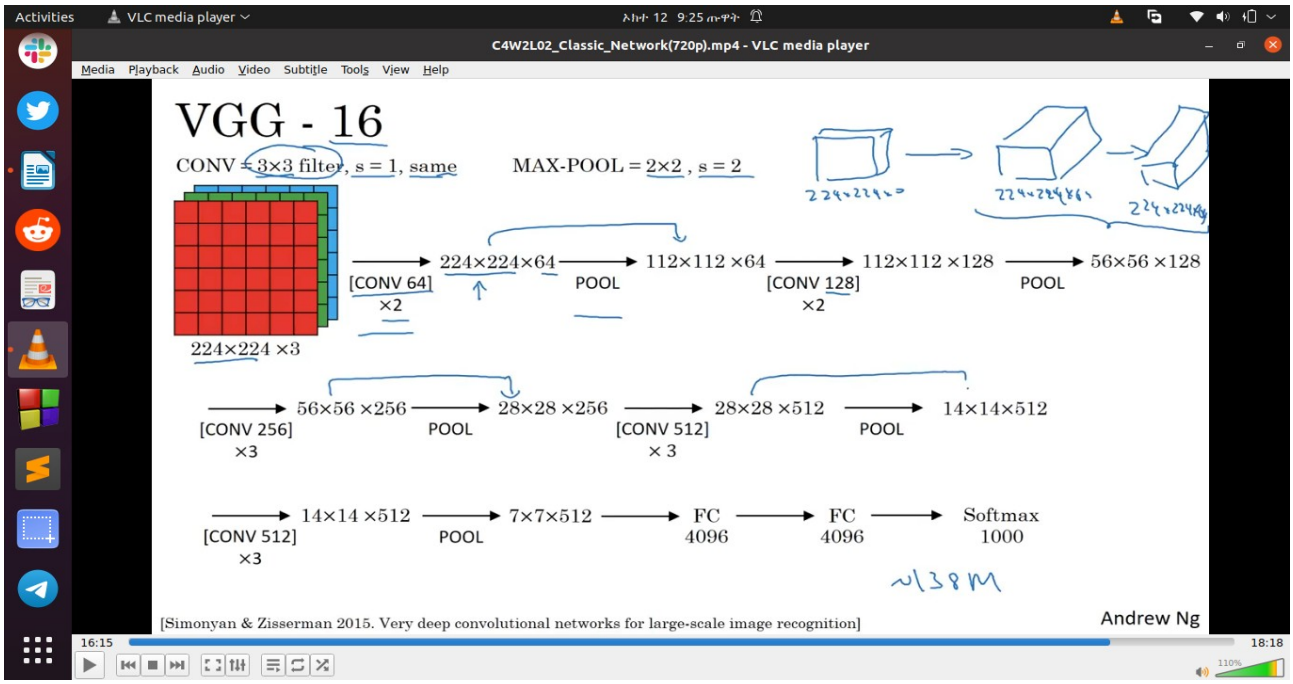


VGG16

DataSet

ImageNet:-is a dataset of over 15 million labeled high-resolution images belonging to roughly 22,000 categories. The images were collected from the web and labeled by human labeler's using Amazon's Mechanical Turk crowd-sourcing tool. Starting in 2010, as part of the Pascal Visual Object Challenge, an annual competition called the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) has been held. ILSVRC uses a subset of ImageNet with roughly 1000 images in each of 1000 categories.

The Architecture



The input to cov1 layer is of fixed size 224 x 224 RGB image. The image is passed through a stack of convolutional (conv.) layers, where the filters were used with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations, it also utilizes 1×1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1-pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv.layers (not all the conv.

layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2.

Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.

All hidden layers are equipped with the rectification (ReLU) non-linearity. The architecture contains about 138 million parameters. There is also an architecture called VGG - 19 where there are 19 layers but similar functionality as VGG 16 so people so many times use VGG 16.

Residual Network (ResNet) - Deep network about 152 layers

Much of the success of Deep Neural Networks has been accredited to these additional layers. The intuition behind their function is that these layers progressively learn more complex features. The first layer learns edges, the second layer learns shapes, the third layer learns objects, the fourth layer learns eyes, and so on. Despite the popular meme shared in AI communities from the Inception movie stating that "We need to go Deeper", He et al. [2] empirically show that there is a maximum threshold for depth with the traditional CNN model.

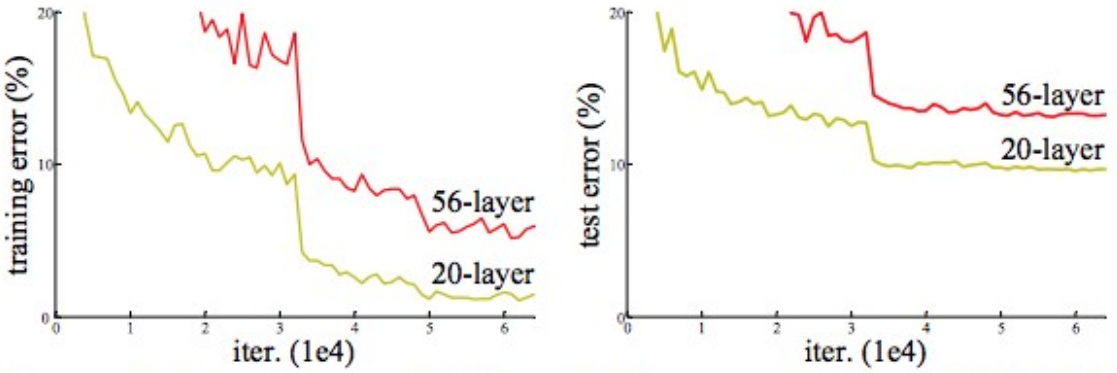


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

He et al. [2] plot the training and test error of a 20-layer CNN versus a 56-layer CNN. This plot defies our belief that adding more layers would create a more complex function, thus the failure would be attributed to overfitting. If this was the case, additional regularization parameters and algorithms such as dropout or L2-norms would be a successful approach for fixing these networks. However, the plot shows that the training error of the 56-layer network is higher than the 20-layer network highlighting a different phenomenon explaining it's failure.

Evidence shows that the best ImageNet models using convolutional and fully-connected layers typically contain between 16 and 30 layers.

The failure of the 56-layer CNN could be blamed on the optimization function, initialization of the network, or the famous vanishing/exploding gradient problem. Vanishing gradients are especially easy to blame for this, however, the authors argue that the use of Batch Normalization ensures that the gradients have healthy norms. Amongst the many theories explaining why Deeper Networks fail to perform better than their Shallow counterparts, it is sometimes better to look for empirical results for explanation and work

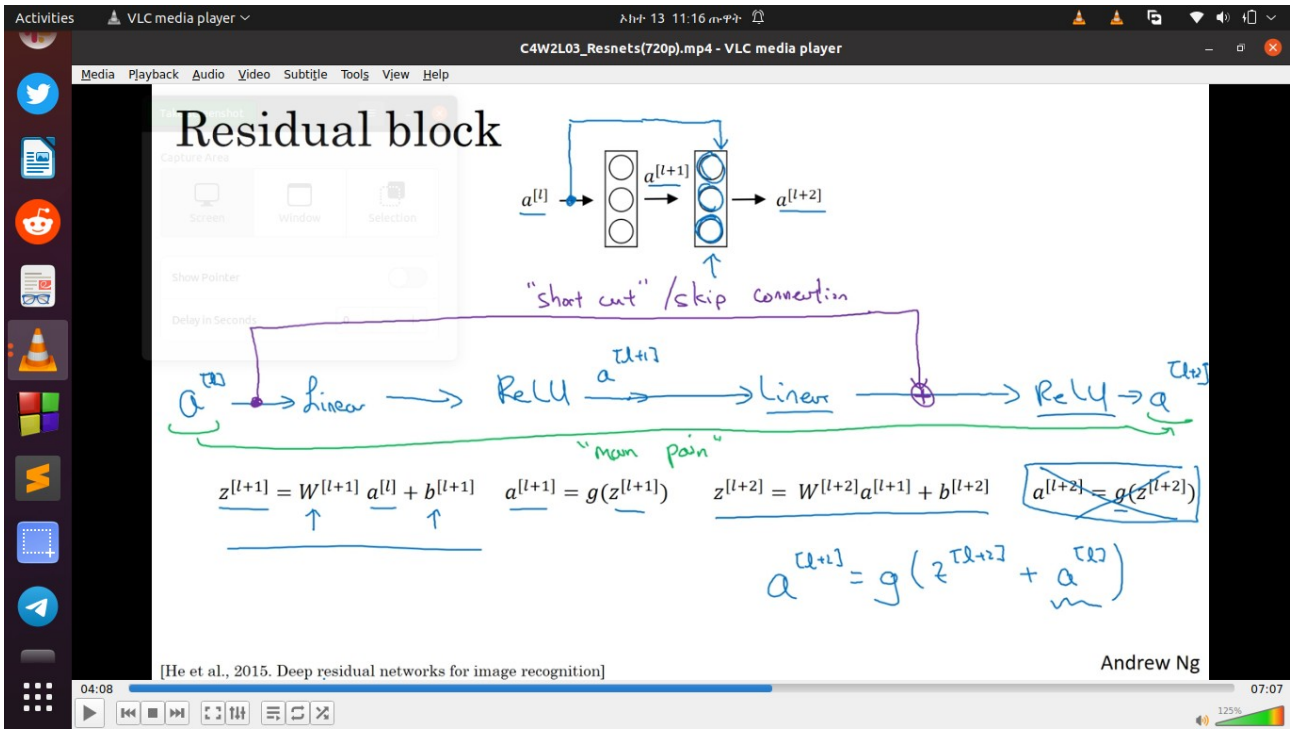
backwards from there. The problem of training very deep networks has been alleviated with the introduction of a new neural network layer —The Residual Block.

The ResNet models were extremely successful which you can guess from the following:

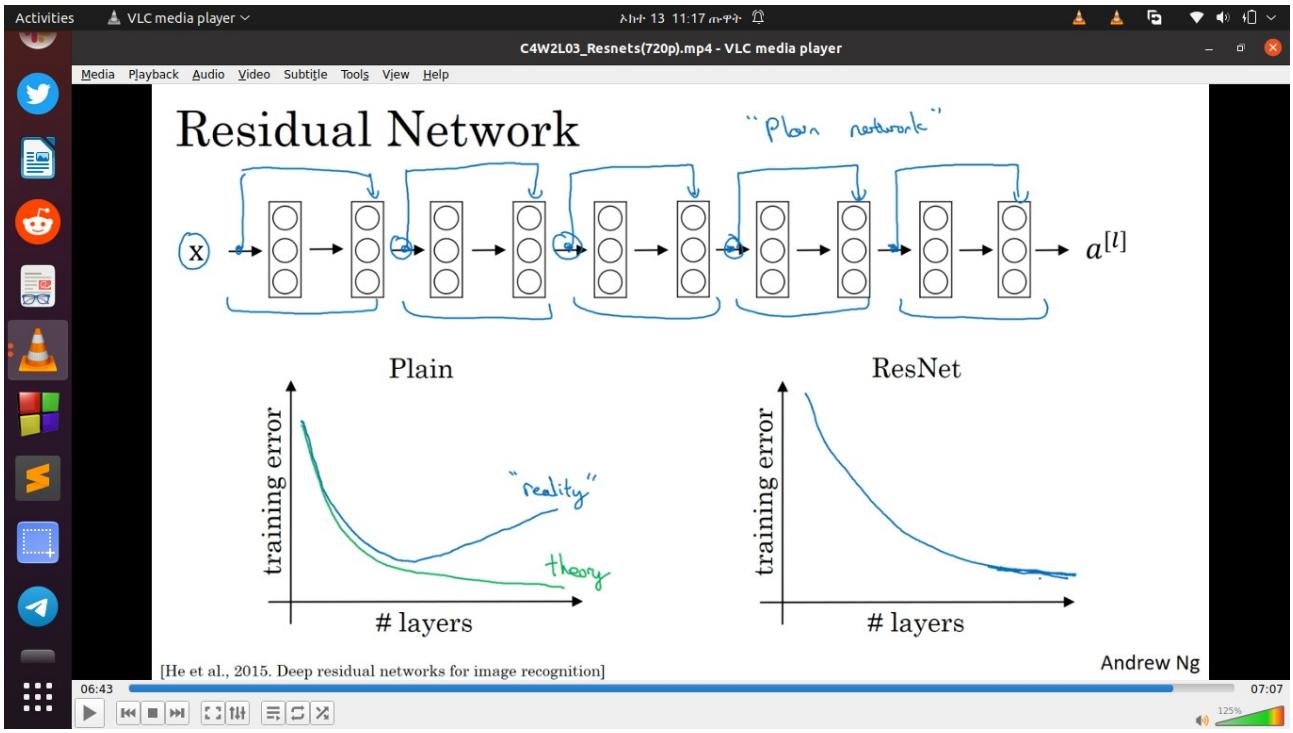
- Won 1st place in the ILSVRC 2015 classification competition with a top-5 error rate of 3.57% (An ensemble model)
- Won the 1st place in ILSVRC and COCO 2015 competition in ImageNet Detection, ImageNet localization, Coco detection and Coco segmentation.
- Replacing VGG-16 layers in Faster R-CNN with ResNet-101. They observed relative improvements of 28%
- Efficiently trained networks with 100 layers and 1000 layers also.

Residual Block

This problem of training very deep networks has been alleviated with the introduction of ResNet or residual networks and these Resnets are made up from Residual Blocks.



So we take many of these residual blocks and stacking them together to form more deeper networks.



Why ResNets Work ?

1*1 convolution

- What is 1*1 convolution
- Why should I use 1*1 convolution
- Is this method hurt the performance of our model , If no , how ?
- How 1*1 convolution works

Pooling can be used to down sample the content of feature maps, reducing their width and height whilst maintaining their salient features.

A problem with deep convolutional neural networks is that the number of feature maps often increases with the depth of the network. This problem can result in a dramatic increase in the number of parameters and computation required when larger filter sizes are used, such as 5×5 and 7×7.

To address this problem, a 1×1 convolutional layer can be used that offers a channel-wise pooling, often called feature map pooling or a projection layer. This simple technique can be used for dimensionality reduction, decreasing the number of feature maps whilst retaining their salient features.

Recap To Convolutions Over Channels

Recall that a convolutional operation is a linear application of a smaller filter to a larger input that results in an output feature map.

A filter applied to an input image or input feature map always results in a single number. The systematic left-to-right and top-to-bottom application of the filter to the input results in a two-dimensional feature map. One filter creates one corresponding feature map.

A filter must have the same depth or number of channels as the input, yet, regardless of the depth of the input and the filter, the resulting output is a single number and one filter creates a feature map with a single channel.

Let's make this concrete with some examples:

- If the input has one channel such as a gray scale image, then a 3×3 filter will be applied in 3x3x1 blocks.
- If the input image has three channels for red, green, and blue, then a 3×3 filter will be applied in 3x3x3 blocks.
- If the input is a block of feature maps from another convolutional or pooling layer and has the depth of 64, then the 3×3 filter will be applied in 3x3x64 blocks to create the single values to make up the single output feature map.

The depth of the output of one convolutional layer is only defined by the number of parallel filters applied to the input.

The Problem With Too Many Feature Maps

The depth of the input or number of filters used in convolutional layers often increases with the depth of the network, resulting in an increase in the number of resulting feature maps. It is a common model design pattern.

Further, some network architectures, such as the inception architecture, may also concatenate the output feature maps from multiple convolutional layers, which may also dramatically increase the depth of the input to subsequent convolutional layers.

A large number of feature maps in a convolutional neural network can cause a problem as a convolutional operation must be performed down through the depth of the input. This is a particular problem if the convolutional operation being performed is relatively large, such as 5×5 or 7×7 pixels, as it can result in considerably more parameters (weights) and, in turn, computation to perform the convolutional operations (large space and time complexity).

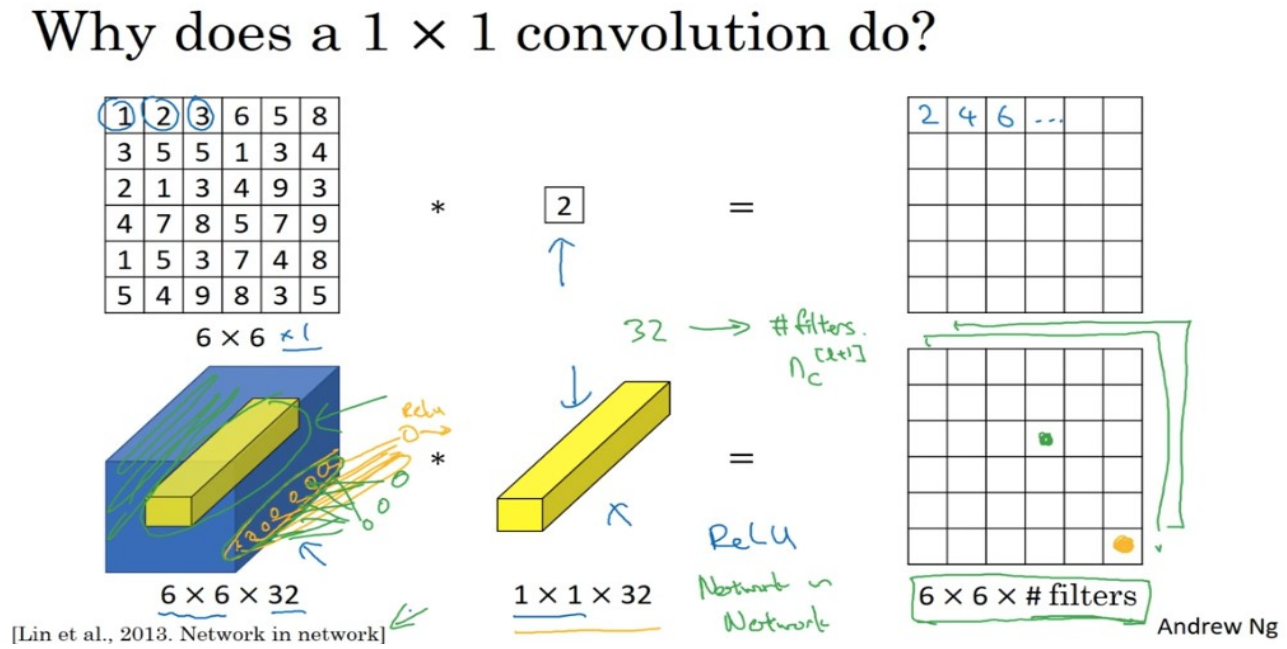
Pooling layers are designed to downscale feature maps and systematically halve the width and height of feature maps in the network. Nevertheless, pooling layers do not change the number of filters in the model, the depth, or number of channels.
Deep convolutional neural networks require a corresponding pooling type of layer that can down sample or reduce the depth or number of feature maps.

Downsample Feature Maps With 1×1 Filters

The solution is to use a 1×1 filter to down sample the depth or number of feature maps.
A 1×1 filter will only have a single parameter or weight for each channel in the input, and like the application of any filter results in a single output value. This structure allows the 1×1 filter to act like a single neuron with an input from the same position across each of the feature maps in the input. This single neuron can then be applied systematically with a stride of one,

left-to-right and top-to-bottom without any need for padding, resulting in a feature map with the same width and height as the input.

The 1×1 filter is so simple that it does not involve any neighboring pixels in the input; it may not be considered a convolutional operation. Instead, it is a linear weighting or projection of the input. Further, a nonlinearity is used as with other convolutional layers, allowing the projection to perform non-trivial computation on the input feature maps.



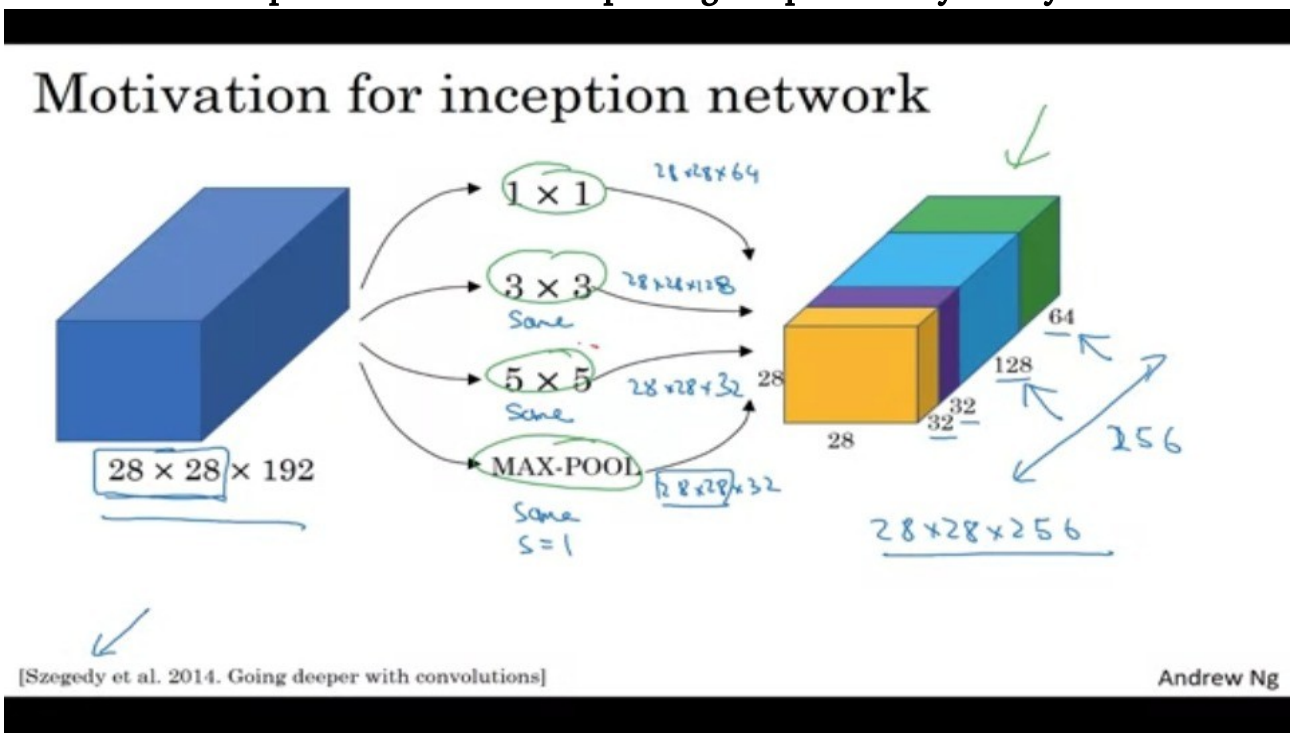
Inception Network

What is Inception Network ?
How does this Architecture work ?

What is the concept of Inception Network ?

When designing a layer for a ConvNet, you might have to pick, do you want a 1 by 3 filter, or 3 by 3, or 5 by 5, or do you want a pooling layer? What the inception network does is it says, why should you do them all? And this makes the network architecture more complicated, but it also works remarkably well. Let's see how this works. Let's say for the sake of example that you have inputted a 28 by 28 by 192 dimensional volume. So what the

inception network or what an inception layer says is, instead choosing what filter size you want in a Conv layer, or even do you want a convolutional layer or a pooling layer? Let's do them all. So what if you can use a 1 by 1 convolution, and that will output a 28 by 28 by something. Let's say 28 by 28 by 64 output, and you just have a volume there. But maybe you also want to try a 3 by 3 and that might output a 20 by 20 by 128. And then what you do is just stack up this second volume next to the first volume. And to make the dimensions match up, let's make this a same convolution. So the output dimension is still 28 by 28, same as the input dimension in terms of height and width. But 28 by 28 by in this example 128. And maybe you might say well I want to hedge my bets. Maybe a 5 by 5 filter works better. So let's do that too and have that output a 28 by 28 by 32. And again you use the same convolution to keep the dimensions the same. And maybe you don't want to convolutional layer. Let's apply pooling, and that has some other output and let's stack that up as well. And here pooling outputs 28 by 28 by 32.



Now in order to make all the dimensions match, you actually need to use padding for max pooling. So this is an unusual formal pooling because if you want the input to have a higher than 28 by 28 and have the output, you'll match the dimension everything else also by 28 by 28, then you need to use the same padding as well as a stride of one for pooling. So this detail might seem a bit funny to you now, but let's keep going. And we'll make this all work later. But with a inception module like this, you can input some volume and output. In this case I guess if you add up all these numbers, 32 plus 32

plus 128 plus 64, that's equal to 256. So you will have one inception module input 28 by 28 by 192, and output 28 by 28 by 256. And the basic idea is that instead of you needing to pick one of these filter sizes or pooling you want and committing to that, you can do them all and just concatenate all the outputs, and let the network learn whatever parameters it wants to use, whatever the combinations of these filter sizes it wants.

Now it turns out that there is a problem with the inception layer as we've described it here, which is computational cost ,

So just focusing on the 5 by 5 pot on the previous slide, we had as input a 28 by 28 by 192 block, and you implement a 5 by 5 same convolution of 32 filters to output 28 by 28 by 32. On the previous slide I had drawn this as a thin purple slide. So I'm just going draw this as a more normal looking blue block here. So let's look at the computational costs , the way we are computing the computational cost is by this formula

Computation cost = (filter parameters) *(filter positions on the input image) * (number of filters)

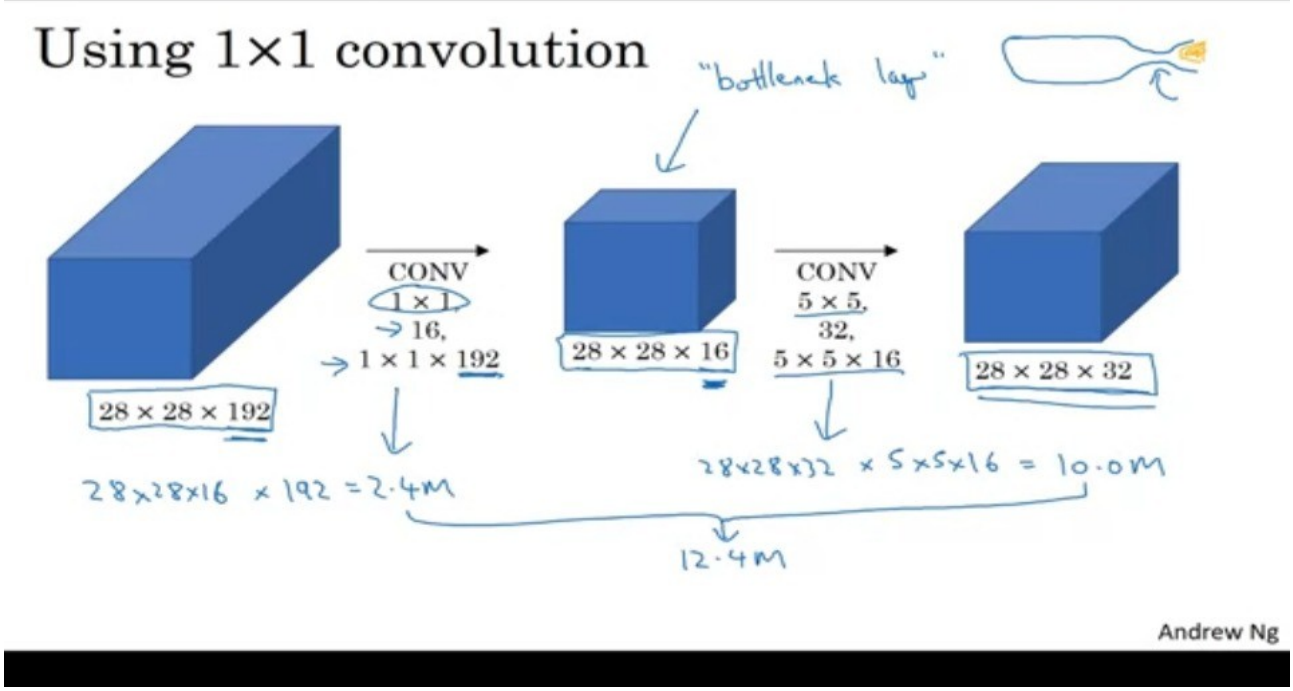
(5 * 5 * 192) * (28 * 28) * (32)

of outputting this 20 by 20 by 32. So you have 32 filters because the outputs has 32 channels, and each filter is going to be 5 by 5 by 192. And so the output size is 20 by 20 by 32, and so you need to compute 28 by 28 by 32 numbers. And for each of them you need to do these many multiplications, right? 5 by 5 by 192. So the total number of multiplies you need is the number of multiplies you need to compute each of the output values times the number of output values you need to compute. And if you multiply all of these numbers, this is equal to 120 million. And so, while you can do 120 million multiplies on the modern computer, this is still a pretty expensive operation.

On the next slide you see how using the idea of 1 by 1 convolutions, which you learnt about in the previous, you'll be able to reduce the computational costs by about a factor of 10. To go from about 120 million multiplies to about one tenth of that. So please remember the number 120 so you can compare it with what you see on the next slide, 120 million. Here is an alternative architecture for inputting 28 by 28 by 192, and outputting 28 by 28 by 32, which is falling. You are going to input the volume, use a 1 by 1 convolution to reduce the volume to 16 channels instead of 192 channels, and then on this much smaller volume, run your 5 by 5 convolution to give you

your final output. So notice the input and output dimensions are still the same. You input 28 by 28 by 192 and output 28 by 28 by 32, same as the previous slide. But what we've done is we're taking this huge volume we had on the left, and we shrunk it to this much smaller intermediate volume, which only has 16 instead of 192 channels. Sometimes this is called a bottleneck layer, I guess because a bottleneck is usually the smallest part of something. So I guess if you have a glass bottle that looks like this, then you know this is I guess where the cork goes. And then the bottleneck is the smallest part of this bottle. So in the same way, the bottleneck layer is the smallest part of this network. We shrink the representation before increasing the size again.

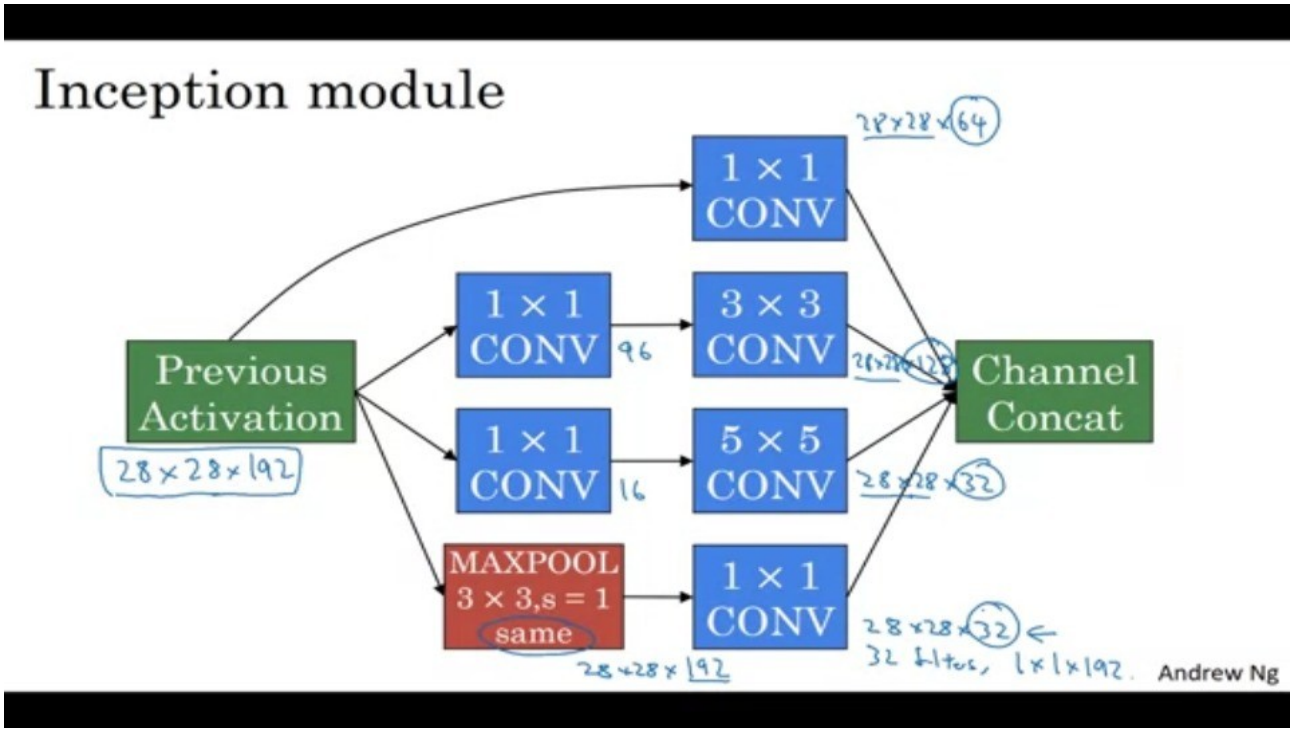
Now let's look at the computational costs involved. To apply this 1 by 1 convolution, we have 16 filters. Each of the filters is going to be of dimension 1 by 1 by 192 , this 192 matches that 192. And so the cost of computing this 28 by 28 by 16 volumes is going to be well, you need these many outputs, and for each of them you need to do 192 multiplications. I could have written 1 times 1 times 192, Which is this. And if you multiply this out, this is 2.4 million, it's about 2.4 million. How about the second? So that's the cost of this first convolutional layer. The cost of this second convolutional layer would be that well, you have these many outputs. So 28 by 28 by 32. And then for each of the outputs you have to apply a 5 by 5 by 16 dimensional filter. And so by 5 by 5 by 16. And you multiply that out is equals to 10.0. And so the total number of multiplications you need to do is the sum of those which is 12.4 million multiplications. And you compare this with what we had on the previous slide, you reduce the computational cost from about 120 million multiplies, down to about one tenth of that, to 12.4 million multiplications. And the number of additions you need to do is about very similar to the number of multiplications you need to do. So that's why I'm just counting the number of multiplications.



So to summarize, if you are building a layer of a neural network and you don't want to have to decide, do you want a 1 by 1, or 3 by 3, or 5 by 5, or pooling layer, the inception module let's you say let's do them all, and let's concatenate the results. And then we run to the problem of computational cost. And what you saw here was how using a 1 by 1 convolution, you can create this bottleneck layer thereby reducing the computational cost significantly. Now you might be wondering, does shrinking down the representation size so dramatically, does it hurt the performance of your neural network? It turns out that so long as you implement this bottleneck layer so that within reason, you can shrink down the representation size significantly, and it doesn't seem to hurt the performance, but saves you a lot of computation. So these are the key ideas of the inception module.

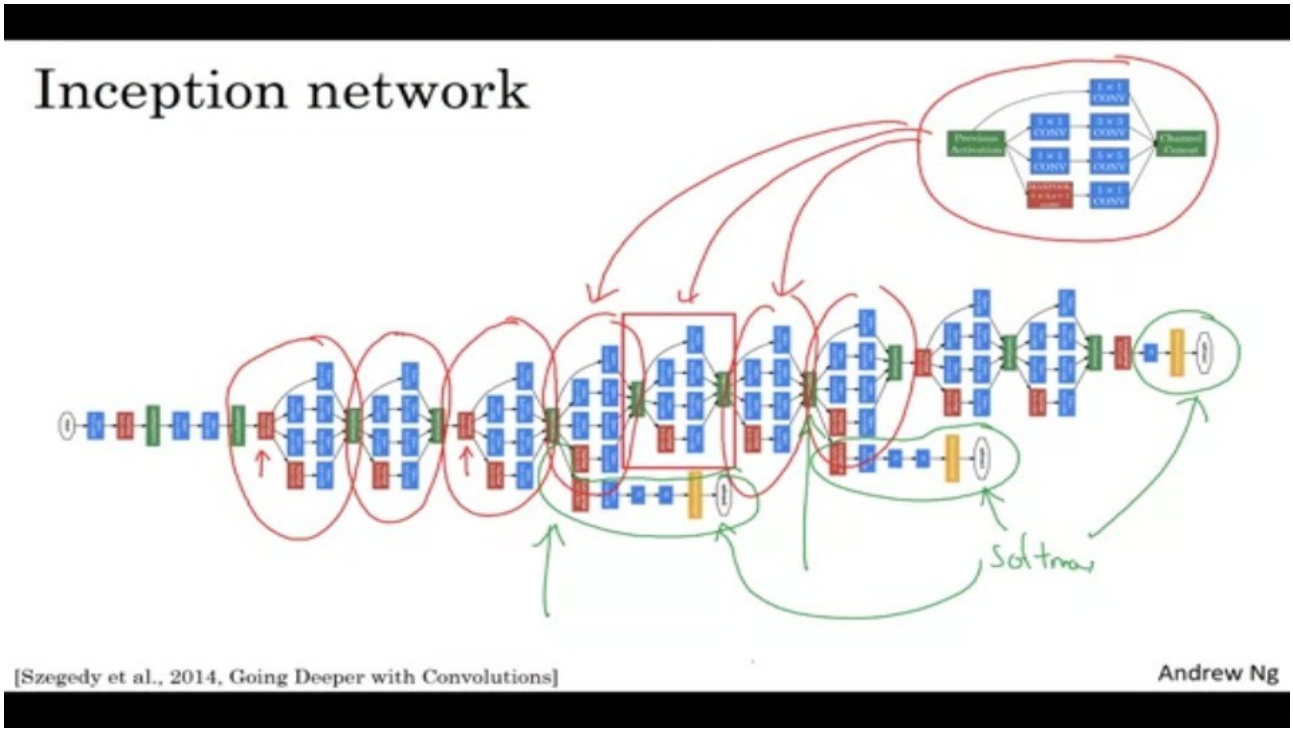
In a previous, you've already seen all the basic building blocks of the Inception network. In this part, let's see how you can put these building blocks together to build your own Inception network. So the inception module takes as input the activation or the output from some previous layer. So let's say for the sake of argument this is 28 by 28 by 192, same as our previous . The example we worked through in depth was the 1 by 1 followed by 5 by 5. There, so maybe the 1 by 1 has 16 channels and then the 5 by 5 will output a 28 by 28 by, let's say, 32 channels. And this is the example we worked through on the last slide of the previous part. Then to save computation on your 3 by 3 convolution you can also do the same here. And then the 3 by 3

outputs, 28 by 28 by 1 by 28. And then maybe you want to consider a 1 by 1 convolution as well. There's no need to do a 1 by 1 conv followed by another 1 by 1 conv so there's just one step here and let's say these outputs 28 by 28 by 64. And then finally is the pulling layer. So here I'm going to do something funny. In order to really concatenate all of these outputs at the end we are going to use the same type of padding for pooling. So that the output height and width is still 28 by 28. So we can concatenate it with these other outputs. But notice that if you do max-pooling, even with same padding, 3 by 3 filter is tried at 1. The output here will be 28 by 28, By 192. It will have the same number of channels and the same depth as the input that we had here. So, this seems like is has a lot of channels. So what we're going to do is actually add one more 1 by 1 conv layer to then to what we saw in the one by one convolutional video, to strengthen the number of channels. So it gets us down to 28 by 28 by let's say, 32. And the way you do that, is to use 32 filters, of dimension 1 by 1 by 192. So that's why the output dimension has a number of channels shrunk down to 32. So then we don't end up with the pulling layer taking up all the channels in the final output. And finally you take all of these blocks and you do channel concatenation. Just concatenate across this 64 plus 128 plus 32 plus 32 and this if you add it up this gives you a 28 by 28 by 256 dimension output.



Concat is just this concatenating the blocks that we saw in the previous part. So the above is one inception module, and what the inception network does,

is, more or less, put a lot of these modules together. Maybe this picture looks really complicated. But if you look at one of the blocks there, that block is basically the inception module that you saw on the previous slide. And subject to little details I won't discuss, this is another inception block. This is another inception block. There's some extra max pooling layers here to change the dimension of the heightened width. But that's another inception block. And then there's another max put here to change the height and width but basically there's another inception block. But the inception network is just a lot of these blocks that you've learned about repeated to different positions of the network. But so you understand the inception block from the previous slide, then you understand the inception network. It turns out that there's one last detail to the inception network if we read the optional research paper. Which is that there are these additional side-branches that I just added So what do they do? Well, the last few layers of the network is a fully connected layer followed by a softmax layer to try to make a prediction. What these side branches do is it takes some hidden layer and it tries to use that to make a prediction. So this is actually a softmax output and so is that. And this other side branch, again it is a hidden layer passes through a few layers like a few connected layers. And then has the softmax try to predict what's the output label. And you should think of this as maybe just another detail of the inception that's worked. But what is does is it helps to ensure that the features computed. Even in the heading units, even at intermediate layers. That they're not too bad for protecting the output cause of a image. And this appears to have a regularizing effect on the inception network and helps prevent this network from over fitting.

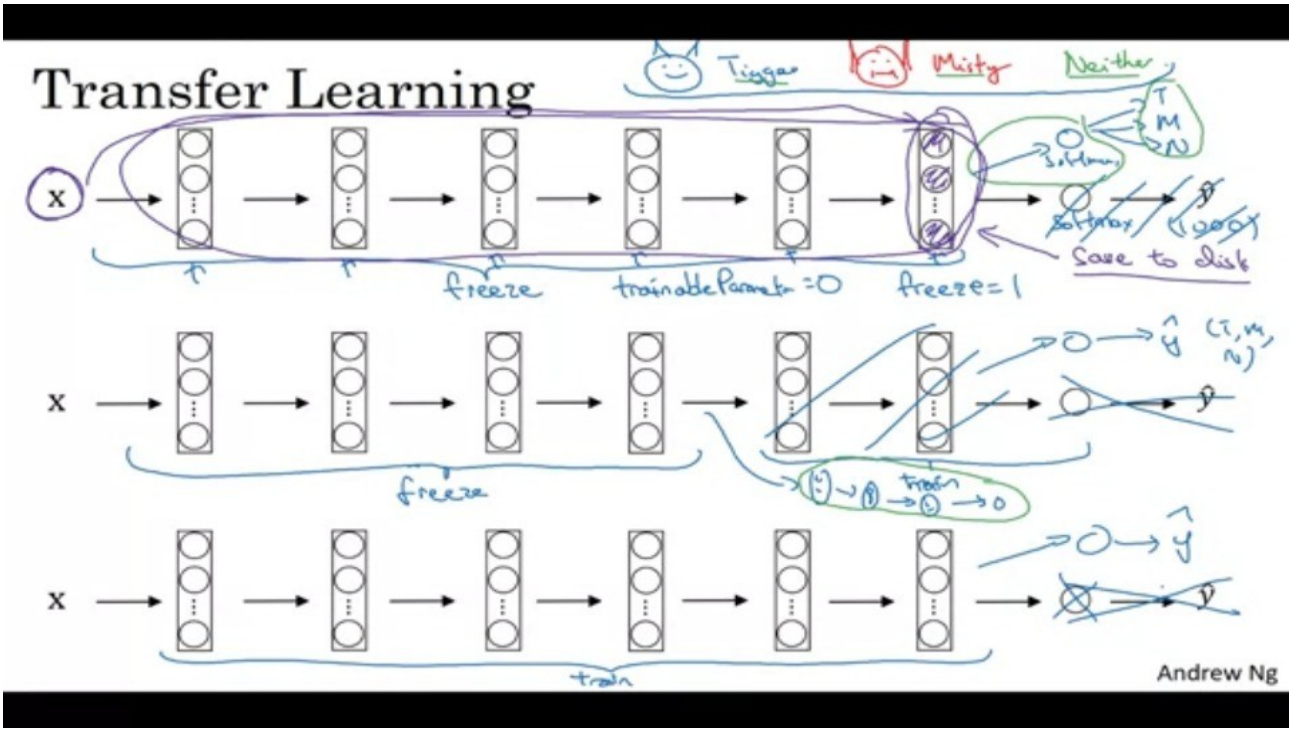


And by the way, this particular Inception network was developed by authors at Google. Who called it GoogLeNet, spelled like that, to pay homage to the network. That you learned about in an earlier video as well. So I think it's actually really nice that the Deep Learning Community is so collaborative. And that there's such strong healthy respect for each other's' work in the Deep Learning Learning community. Finally here's one fun fact. Where does the name inception network come from? The inception paper actually cites this meme for we need to go deeper. And this URL is an actual reference in the inception paper, which links to this image. And if you've seen the movie titled The Inception, maybe this meme will make sense to you. But the authors actually cite this meme as motivation for needing to build deeper new networks. And that's how they came up with the inception architecture. So I guess it's not often that research papers get to cite Internet memes in their citations. But in this case, I guess it worked out quite well.

So to summarize, if you understand the inception module, then you understand the inception network. Which is largely the inception module repeated a bunch of times throughout the network. Since the development of the original inception module, the author and others have built on it and come up with other versions as well. So there are research papers on newer versions of the

inception algorithm. And you sometimes see people use some of these later versions as well in their work, like inception v2, inception v3, inception v4. There's also an inception version. This combined with the resonant idea of having skipped connections, and that sometimes works even better.

Transfer Learning



If you're building a computer vision application rather than training the ways from scratch, from random initialization, you often make much faster progress if you download ways that someone else has already trained on the network architecture and use that as pre-training and transfer that to a new task that you might be interested in. The computer vision research community has been pretty good at posting lots of data sets on the Internet so if you hear of things like Image Net, or MS COCO, or Pascal types of data sets, these are the names of different data sets that people have post online and a lot of computer researchers have trained their algorithms on. Sometimes these training takes several weeks and might take many GPU and the fact that someone else has done this and gone through the painful high-performance search process, means that you can often download open source ways that took someone else many weeks or months to figure out and use that as a very good initialization for your own neural network. And use transfer learning to sort of transfer knowledge from some of these very large public data sets to your own problem. Let's start with the example, let's say you're building a cat detector

to recognize your own pet cat. According to the internet, Tigger is a common cat name and Misty is another common cat name. Let's say your cats are called Tiger and Misty and there's also neither. You have a classification problem with three clauses. Is this picture Tigger, or is it Misty, or is it neither. And in all the case of both of you cats appearing in the picture. Now, you probably don't have a lot of pictures of Tigger or Misty so your training set will be small. What can you do? I recommend you go online and download some open source implementation of a neural network and download not just the code but also the weights. There are a lot of networks you can download that have been trained on for example, the ImagNet data sets which has a thousand different classes so the network might have a softmax unit that outputs one of a thousand possible classes can do is then get rid of the softmax layer and create your own softmax unit that outputs Tigger or Misty or neither. In terms of the network, I'd encourage you to think of all of these layers as frozen so you freeze the parameters in all of these layers of the network and you would then just train the parameters associated with your softmax layer. Which is the softmax layer with three possible outputs, Tigger, Misty or neither. By using someone else's free trade ways, you might probably get pretty good performance on this even with a small data set. Fortunately, a lot of people learning frameworks support this mode of operation and in fact, depending on the framework it might have things like trainable parameter equals zero, you might set that for some of these early layers. In others they just say, don't train those ways or sometimes you have a parameter like freeze equals one and these are different ways and different deep learning program frameworks that let you specify whether or not to train the ways associated with particular layer. In this case, you will train only the softmax layers ways but freeze all of the earlier layers ways.

One other neat trick that may help for some implementations is that because all of these early leads are frozen, there are some fixed function that doesn't change because you're not changing it, you not training it that takes this input image acts and maps it to some set of activations in that layer. One of the trick that could speed up training is we just pre-compute that layer, the features of re-activations from that layer and just save them to disk. What you're doing is using this fixed function, in this first part of the neural network, to take this input any image X and compute some feature vector for it and then you're training a shallow softmax model from this feature vector to make a prediction. One step that could help your computation as you just pre-compute that layers activation, for all the examples in training sets and save them to disk and then just train the softmax clause right on top of that. The

advantage of the safety disk or the pre-compute method or the safety disk is that you don't need to recompute those activations everytime you take a epoch or take a post through a training set. This is what you do if you have a pretty small training set for your task. Whether you have a larger training set.

One rule of thumb is if you have a larger label data set so maybe you just have a ton of pictures of Tigger, Misty as well as I guess pictures neither of them, one thing you could do is then freeze fewer layers. Maybe you freeze just the top layers and then train the later layers. Although if the output layer has different classes then you need to have your own output unit any way Tigger, Misty or neither. There are a couple of ways to do this. You could take the last few layers ways and just use that as initialization and do gradient descent from there or you can also blow away these last few layers and just use your own new hidden units and in your own final softmax outputs. Either of these matters could be worth trying. But maybe one pattern is if you have more data, the number of layers you've freeze could be smaller and then the number of layers you train on top could be greater. And the idea is that if you pick a data set and maybe have enough data not just to train a single softmax unit but to train some other size neural network that comprises the last few layers of this final network that you end up using.

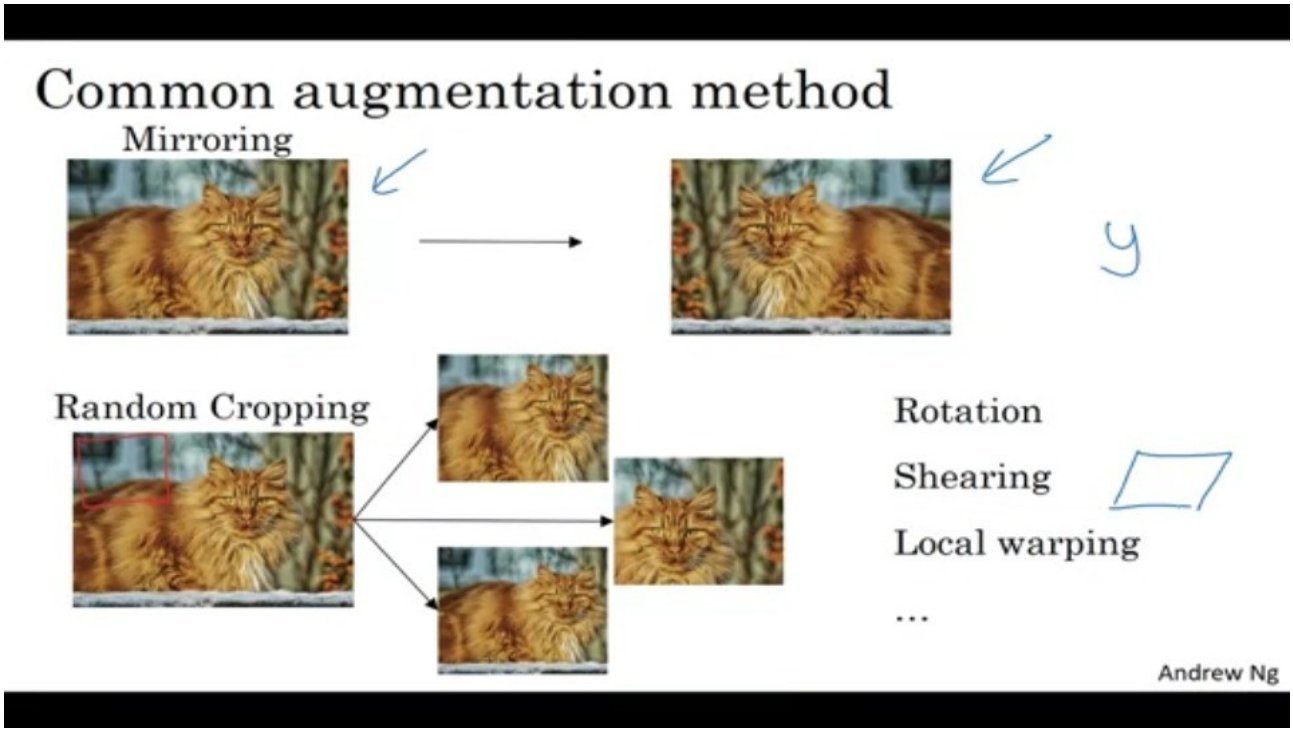
Finally, if you have a lot of data, one thing you might do is take this open source network and ways and use the whole thing just as initialization and train the whole network. Although again if this was a thousand of softmax and you have just three outputs, you need your own softmax output. The output of labels you care about. But the more label data you have for your task or the more pictures you have of Tigger, Misty and neither, the more layers you could train and in the extreme case, you could use the ways you download just as initialization so they would replace random initialization and then could do gradient descent, training updating all the ways and all the layers of the network. That's transfer learning for the training of ConvNets.

In practice, because the open data sets on the internet are so big and the ways you can download that someone else has spent weeks training has learned from so much data, you find that for a lot of computer vision applications, you just do much better if you download someone else's open source ways and use that as initialization for your problem. In all the different disciplines, in all the different applications of deep learning, I think that computer vision is one where transfer learning is something that you should almost always do unless, you have an exceptionally large data set to train

everything else from scratch yourself. But transfer learning is just very worth seriously considering unless you have an exceptionally large data set and a very large computation budget to train everything from scratch by yourself.

Data Augmentation

Most computer vision task could use more data. And so data augmentation is one of the techniques that is often used to improve the performance of computer vision systems. I think that computer vision is a pretty complicated task. You have to input this image, all these pixels and then figure out what is in this picture. And it seems like you need to learn the decently complicated function to do that. And in practice, there almost all competing visions task having more data will help. This is unlike some other domains where sometimes you can get enough data, they don't feel as much pressure to get even more data. But I think today, this data computer vision is that, for the majority of computer vision problems, we feel like we just can't get enough data. And this is not true for all applications of machine learning, but it does feel like it's true for computer vision. So, what that means is that when you're training in computer vision model, often data augmentation will help. And this is true whether you're using transfer learning or using someone else's pre-trained ways to start, or whether you're trying to train something yourself from scratch. Let's take a look at the common data augmentation that is in computer vision. Perhaps the simplest data augmentation method is mirroring on the vertical axis, where if you have this example in your training set, you flip it horizontally to get that image on the right.



And for most computer vision task, if the left picture is a cat then mirroring it is though a cat. And if the mirroring operation preserves whatever you're trying to recognize in the picture, this would be a good data augmentation technique to use. Another commonly used technique is random cropping. So given this dataset, let's pick a few random crops. So you might pick that,and take that crop or you might take that, to that crop, take this, take that crop and so this gives you different examples to feed in your training sample, sort of different random crops of your datasets. So random cropping isn't a perfect data augmentation. What if you randomly end up taking that empty image crop which will look much like a cat but in practice and worthwhile so long as your random crops are reasonably large subsets of the actual image. So, mirroring and random cropping are frequently used and in theory, you could also use things like rotation, shearing of the image, so that's if you do this to the image, distort it that way, introduce various forms of local warping and so on. And there's really no harm with trying all of these things as well, although in practice they seem to be used a bit less, or perhaps because of their complexity.

The second type of data augmentation that is commonly used is color shifting. So, given a picture like this, let's say you add to the R, G and B channels different distortions. In this example, we are adding to the red and blue channels and subtracting from the green channel. So, red and blue make

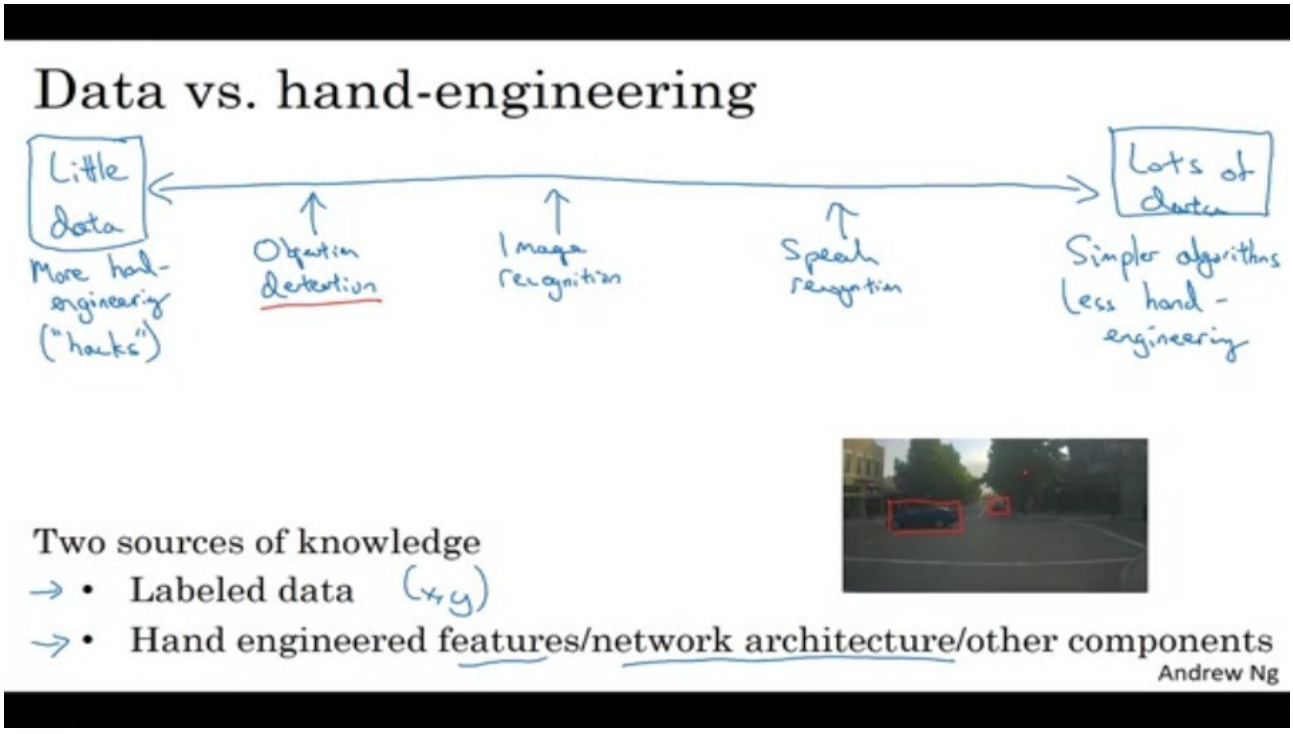
purple. So, this makes the whole image a bit more purpley and that creates a distorted image for training set. For illustration purposes, I'm making somewhat dramatic changes to the colors and practice, you draw R, G and B from some distribution that could be quite small as well. But what you do is take different values of R, G, and B and use them to distort the color channels. So, in the second example, we are making a less red, and more green and more blue, so that turns our image a bit more yellowish. And here, we are making it much more blue, just a tiny little bit longer. But in practice, the values R, G and B, are drawn from some probability distribution. And the motivation for this is that if maybe the sunlight was a bit yellow or maybe the in-goal illumination was a bit more yellow, that could easily change the color of an image, but the identity of the cat or the identity of the content, the label y, just still stay the same. And so introducing these color distortions or by doing color shifting, this makes your learning algorithm more robust to changes in the colors of your images. similar to other parts of training a deep neural network, the data augmentation process also has a few hyper-parameters such as how much color shifting do you implement and exactly what parameters you use for random cropping? So, similar to elsewhere in computer vision, a good place to get started might be to use someone else's open source implementation for how they use data augmentation. But of course, if you want to capture more in variances, then you think someone else's open source implementation isn't, it might be reasonable also to use hyper-parameters yourself. So with that, I hope that you're going to use data augmentation, to get your computer vision applications to work better.

State Of Computer Vision

Deep learning has been successfully applied to computer vision, natural language processing, speech recognition, online advertising, logistics, many, many, many problems. There are a few things that are unique about the application of deep learning to computer vision, about the status of computer vision. In this part, we will see some observations about deep learning for computer vision and I hope that that will help you better navigate the literature, and the set of ideas out there, and how you build these systems yourself for computer vision. So, you can think of most machine learning problems as falling somewhere on the spectrum between where you have relatively little data to where you have lots of data. So, for example I think

that today we have a decent amount of data for speech recognition and it's relative to the complexity of the problem. And even though there are reasonably large data sets today for image recognition or image classification, because image recognition is just a complicated problem to look at all those pixels and figure out what it is. It feels like even though the online data sets are quite big like over a million images, feels like we still wish we had more data. And there are some problems like object detection where we have even less data. So, just as a reminder image recognition was the problem of looking at a picture and telling you is this a cat or not. Whereas object detection is look in the picture and actually you're putting the bounding boxes are telling you where in the picture the objects such as the car as well. And so because of the cost of getting the bounding boxes is just more expensive to label the objects and the bounding boxes. So, we tend to have less data for object detection than for image recognition. And object detection is something we'll discuss next week. So, if you look across a broad spectrum of machine learning problems, you see on average that when you have a lot of data you tend to find people getting away with using simpler algorithms as well as less hand-engineering. So, there's just less needing to carefully design features for the problem, but instead you can have a giant neural network, even a simpler architecture, and have a neural network. Just learn whether we want to learn we have a lot of data. Whereas, in contrast when you don't have that much data then on average you see people engaging in more hand-engineering. And if you want to be ungenerous you can say there are more hacks. But I think when you don't have much data then hand-engineering is actually the best way to get good performance. So, when I look at machine learning applications I think usually we have the learning algorithm has two sources of knowledge. One source of knowledge is the labeled data, really the (x,y) pairs you use for supervised learning. And the second source of knowledge is the hand-engineering. And there are lots of ways to hand-engineer a system. It can be from carefully hand designing the features, to carefully hand designing the network architectures to maybe other components of your system. And so when you don't have much labeled data you just have to call more on hand-engineering. And so I think computer vision is trying to learn a really complex function. And it often feels like we don't have enough data for computer vision. Even though data sets are getting bigger and bigger, often we just don't have as much data as we need. And this is why this data computer vision historically and even today has relied more on hand-engineering. And I think this is also why that either computer vision has developed rather complex network architectures, is because in the absence of more data the way to get good performance is to spend more time architecting , or fooling around with

the network architecture. And in case you think I'm being derogatory of hand-engineering that's not at all my intent. When you don't have enough data hand-engineering is a very difficult, very skillful task that requires a lot of insight. And someone that is insightful with hand-engineering will get better performance, and is a great contribution to a project to do that hand-engineering when you don't have enough data. It's just when you have lots of data then I wouldn't spend time hand-engineering, I would spend time building up the learning system instead. But I think historically the fear the computer vision has used very small data sets, and so historically the computer vision literature has relied on a lot of hand-engineering. And even though in the last few years the amount of data with the right computer vision task has increased dramatically, I think that that has resulted in a significant reduction in the amount of hand-engineering that's being done. But there's still a lot of hand-engineering of network architectures and computer vision. Which is why you see very complicated hyper frantic choices in computer vision, are more complex than you do in a lot of other disciplines. And in fact, because you usually have smaller object detection data sets than image recognition data sets, when we talk about object detection that is task like this next week.



You see that the algorithms become even more complex and has even more specialized components. Fortunately, one thing that helps a lot when you have little data is transfer learning. And I would say for the example from the previous slide of the tigger, misty, neither detection problem, you have soluble

data that transfer learning will help a lot. And so that's another set of techniques that's used a lot for when you have relatively little data. If you look at the computer vision literature, and look at the sort of ideas out there, you also find that people are really enthusiastic.

Tips For Bench Marking and Winning Competitions

They're really into doing well on standardized benchmark data sets and on winning competitions. And for computer vision researchers if you do well and the benchmark is easier to get the paper published. So, there's just a lot of attention on doing well on these benchmarks. And the positive side of this is that, it helps the whole community figure out what are the most effective algorithms. But you also see in the papers people do things that allow you to do well on a benchmark, but that you wouldn't really use in a production or a system that you deploy in an actual application. So, here are a few tips on doing well on benchmarks. These are things that I don't myself pretty much ever use if I'm putting a system to production that is actually to serve customers. But one is ensembling. And what that means is, after you've figured out what neural network you want, train several neural networks independently and average their outputs. So, initialize say 3, or 5, or 7 neural networks randomly and train up all of these neural networks, and then average their outputs. And by the way it is important to average their outputs y hats. Don't average their weights that won't work. Look and you say seven neural networks that have seven different predictions and average that. And this will cause you to do maybe 1% better, or 2% better. So is a little bit better on some benchmark. And this will cause you to do a little bit better. Maybe sometimes as much as 1 or 2% which really help win a competition. But because ensembling means that to test on each image, you might need to run an image through anywhere from say 3 to 15 different networks quite typical. This slows down your running time by a factor of 3 to 15, or sometimes even more. And so ensembling is one of those tips that people use doing well in benchmarks and for winning competitions. But that I think is almost never use in production to serve actual customers. I guess unless you have huge computational budget and don't mind burning a lot more of it per customer image. Another thing you see in papers that really helps on benchmarks, is multi-crop at test time. So, what I mean by that is you've seen how you can do data augmentation. And multi-crop is a form of applying data augmentation to your test image as well. So, for example let's see a cat image and just copy it four times including two more versions. There's a technique called the 10-crop, which basically says let's say you take this central region

that crop, and run it through your crossfire. And then take that crop up the left hand corner run through a crossfire, up right hand corner shown in green, lower left shown in yellow,
lower right shown in orange, and run that through the crossfire. And then do the same thing with the mirrored image. Right. So I'll take the central crop, then take the four corners crops. So, that's one central crop here and here, there's four corners crop here and here. And if you add these up that's 10 different crops that you mentioned. So hence the name 10-crop. And so what you do, is you run these 10 images through your crossfire and then average the results. So, if you have the computational budget you could do this. Maybe you don't need as many as 10-crops, you can use a few crops. And this might get you a little bit better performance in a production system. By production I mean a system you're deploying for actual users. But this is another technique that is used much more for doing well on benchmarks than in actual production systems. And one of the big problems of ensembling is that you need to keep all these different networks around. And so that just takes up a lot more computer memory. For multi-crop I guess at least you keep just one network around. So it doesn't suck up as much memory, but it still slows down your run time quite a bit. So, these are tips you see and research papers will refer to these tips as well. But I personally do not tend to use these methods when building production systems even though they are great for doing better on benchmarks and on winning competitions. Because a lot of the computer vision problems are in the small data regime, others have done a lot of hand-engineering of the network architectures. And a neural network that works well on one vision problem often may be surprisingly, but they just often would work on other vision problems as well. So, to build a practical system often you do well starting off with someone else's neural network architecture. And you can use an open source implementation if possible, because the open source implementation might have figured out all the finicky details like the learning rate, case scheduler, and other hyper parameters. And finally someone else may have spent weeks training a model on half a dozen GP use and on over a million images. And so by using someone else's pretrained model and fine tuning on your data set, you can often get going much faster on an application. But of course if you have the compute resources and the inclination, don't let me stop you from training your own networks from scratch. And in fact if you want to invent your own computer vision algorithm, that's what you might have to do. So, that's it for this week, I hope that seeing a number of computer vision architectures helps you get a sense of what works. In this week's programming exercises you actually learn another programming framework and use that to implement

resonance. So, I hope you enjoy that programming exercise and I look forward to seeing you next week.

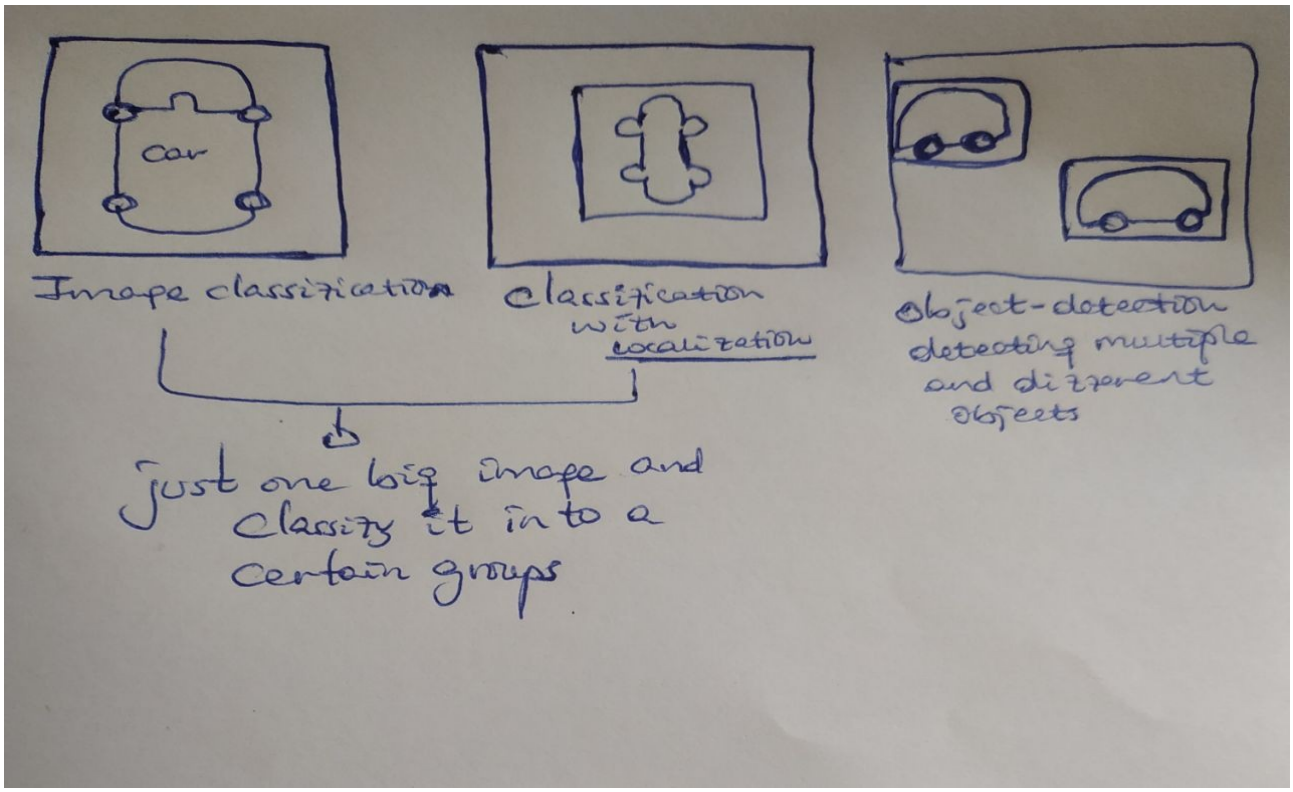
Inception Network

Transfer Learning

Data Augmentation

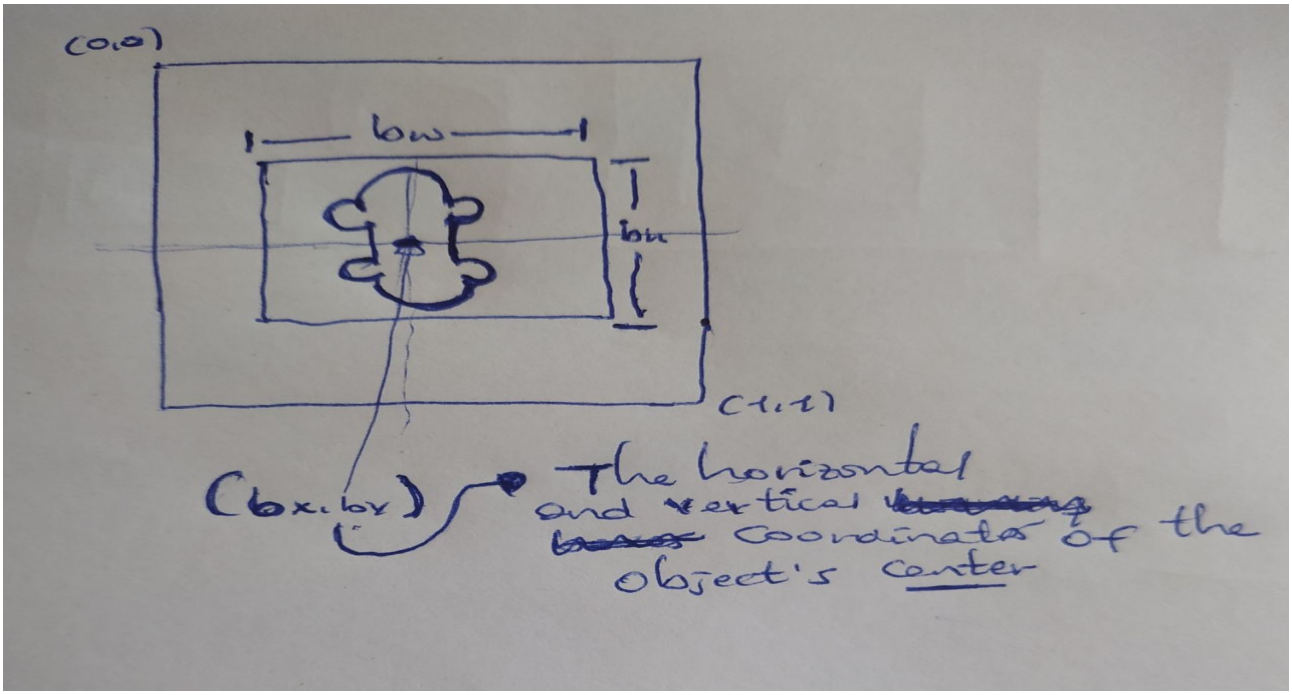
Classification and Localization

Until now we have seen Image Classification problems.



So first we will see classification and localization then we will use that concept for object detection.

Localizing an object in a picture can be expressed as a Regression task , to predict a bounding box around the object , a common approach is to predict the horizontal and vertical coordinates of the object's center as well as the

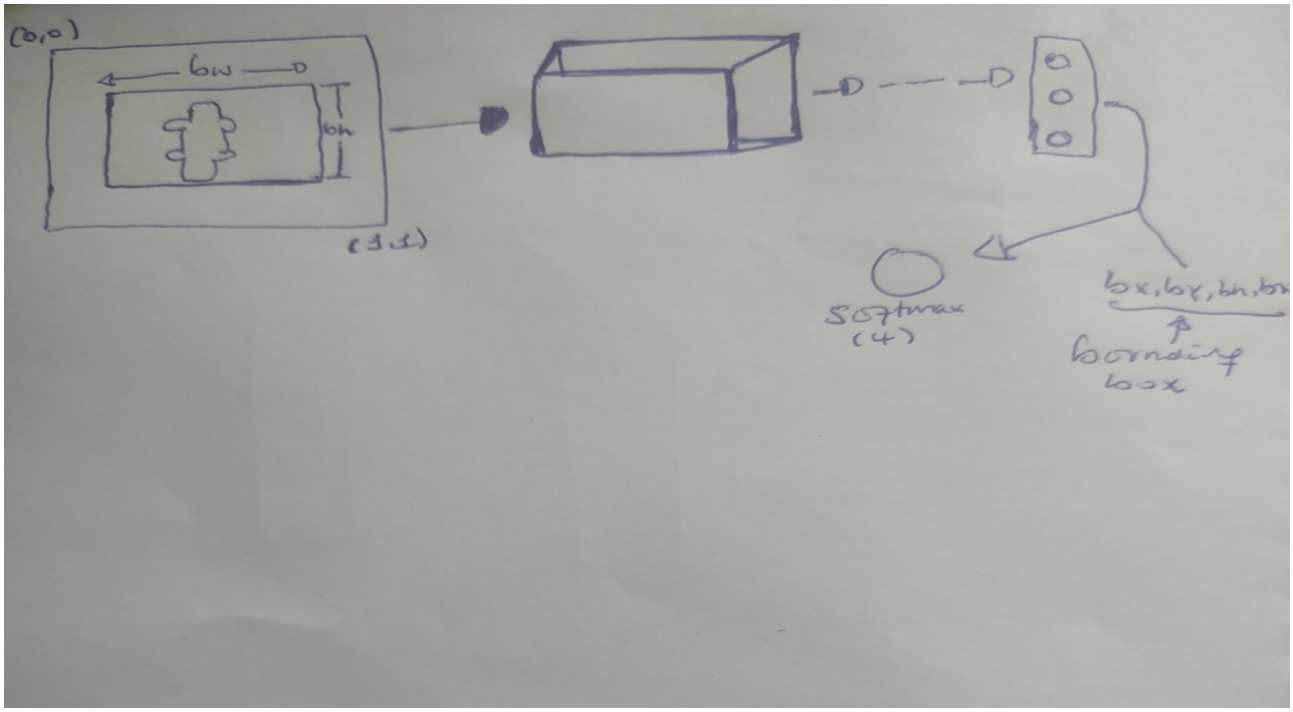


bounding box's height and width. This means we have four numbers to predict.

But the problem is , most of the datasets does not have a bounding boxes around the object , so we need to add them ourselves. This is often one of the hardest and most costly parts of a Machine Learning project : getting the labels. It's good idea to spend time looking for the right tools. To annotate images with bounding boxes , we might want to use an open source image labeling tool like VGG Image Annotator , LabelImg , OpenLabeler or ImgLab or commercial tool like Label Box or supervisely. We may also want to consider crowdsourcing platforms such as Amazon Mechanical Turk if we have a large number of images to annotate. However it's quite a lot of work to set up a crowd souring platform , prepare the form to be sent to the workers , supervise them and ensure that the quality of the bounding boxes they produce is good , so make sure it is worth the effort. If there are just a few thousand images to label , and we don't plan to do this frequently , it may be preferable to do it ourselves.

Let's suppose we've obtained the bounding boxes for every image in our datasets (for now we will assume there is a single bounding box per image) .

We then need to create a dataset whose items will be batches of pre-processed images along with their class labels and their bounding boxes. Each item should be a tuple of the form (images , (class_labels , bounding_boxes)) , then we are now ready to train our model.



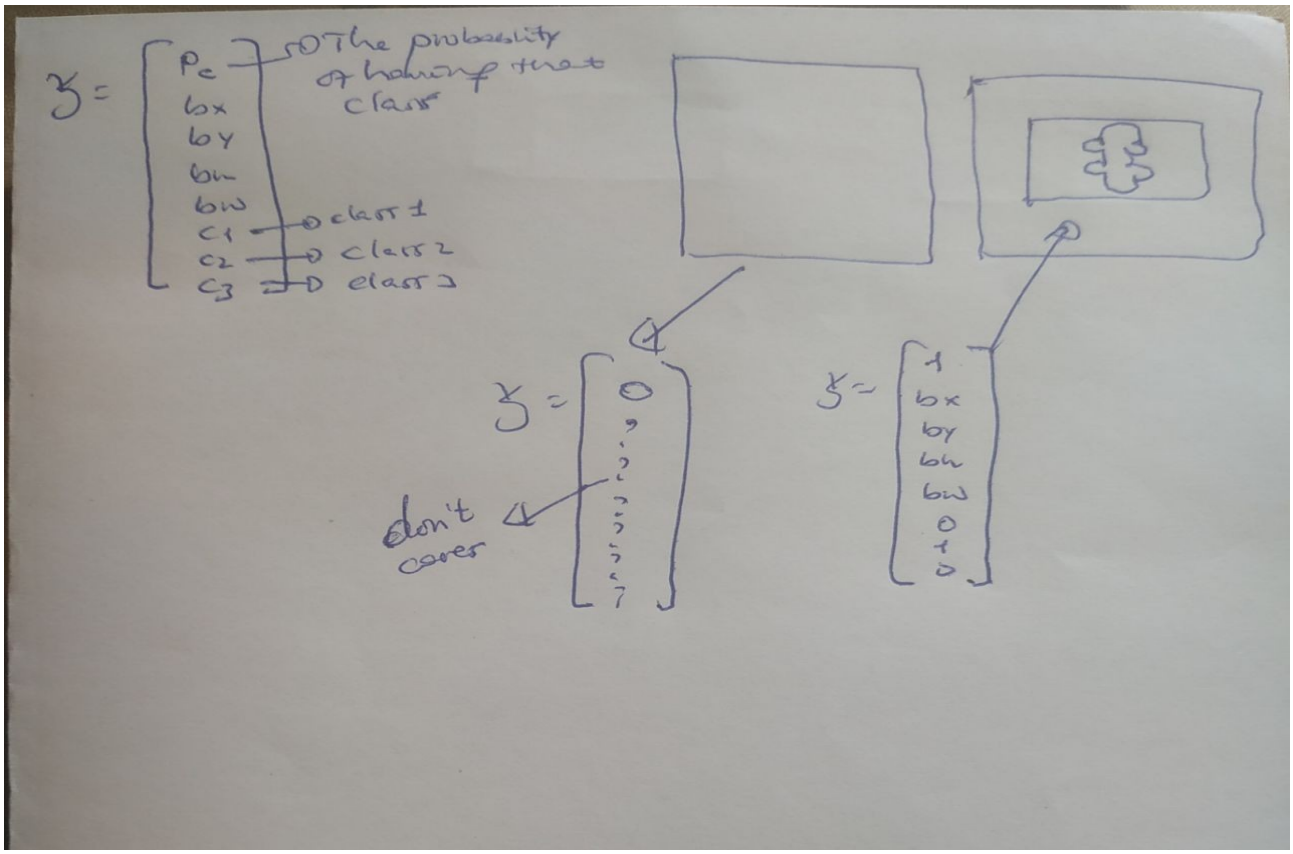
In CNN-based classifiers , initial layers are convolutional neural network layers ranging from couple of layers to 100 layers (eg. Resnet) and that depends on the application , amount of data and computational resources available. The number of layers is itself a large area of research. After the CNN layers there are pooling layer and then one or two fully connected layers. The last layer is the output layer which gives a probability of an object being present in an image. For example suppose an algorithm identifies 100 different objects in a given image. Then the last layer gives an array length of 100 and values ranging from 0 to 1 that denote the probability of an object being present in an image.

In an image localization algorithm , everything is the same except the output layer. In classification algorithms , the final layer gives a probability value ranging from 0 to 1. In contrast, localization algorithms give an output in four

real numbers , as localization is a regression problem , as these four values are used to draw a box around the object.

Note :- The bounding boxes should be normalized so that horizontal and vertical coordinates , as well as the height and width , all range from 0 to 1.

Defining the Target Label y



The loss function for Object Localization

We can calculate the loss function using MSE:-

loss($y^$, y) = $(y^_1 - y_1)^2$ + $(y^_2 - y_2)^2$ + $(y^_3 - y_3)^2$ + $(y^_4 - y_4)^2$ + $(y^_5 - y_5)^2$ + $(y^_6 - y_6)^2$ + $(y^_7 - y_7)^2$ + $(y^_8 - y_8)^2$ if $y^_1 = 1$, because we have 8 labels which is 4 bounding boxes , 3 number of classes , 1 to check the probability of having a class or not

loss($y^$, y) = $(y^_1 - y_1)^2$ if $y^_1 = 0$,

The Mean squared error often works fairly well as a cost function to train the model , but it is not a great metric to evaluate how well the model can

predict bounding boxes. The most common metric for this is the Intersection over union (IOU)

Evaluating The Object Localization

When we build machine learning models for the real world, we are require to evaluate such models to check their performance on unseen data points. with the same intention in mind we rely on an evaluation metric to check the effectiveness of different models on real-world data points. For the Image Localization task , IOU is the widely used evaluation metrics. IOU stands for Intersection over Union.

To calculate IoU , we consider both the ground truth bounding box and the predicted bounding box.

Intersection over Union (IoU): the area of overlap between the predicted bounding box and the target bounding box divided by the are of their union. IoU is simply the ratio of intersection and union , where the intersection is the area of overlap between the ground truth bound box and the predicted bounding box. In contrast , the union is the total area covered by the both bounding boxes together.

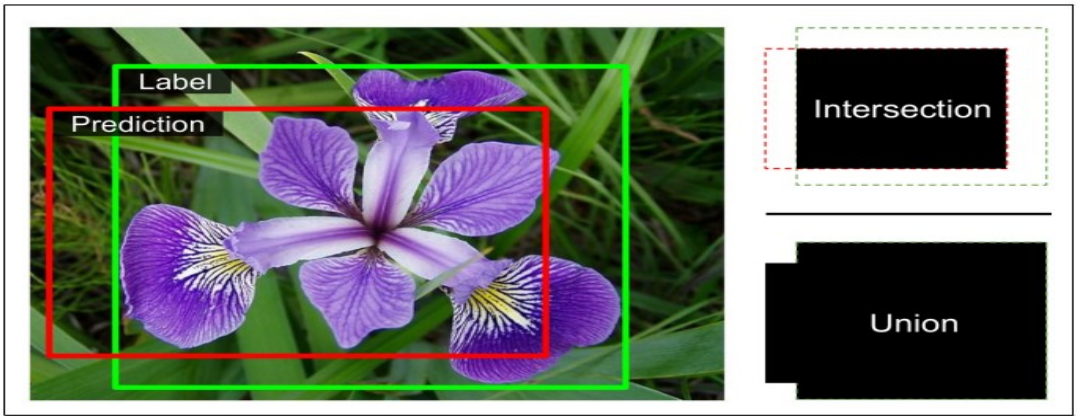
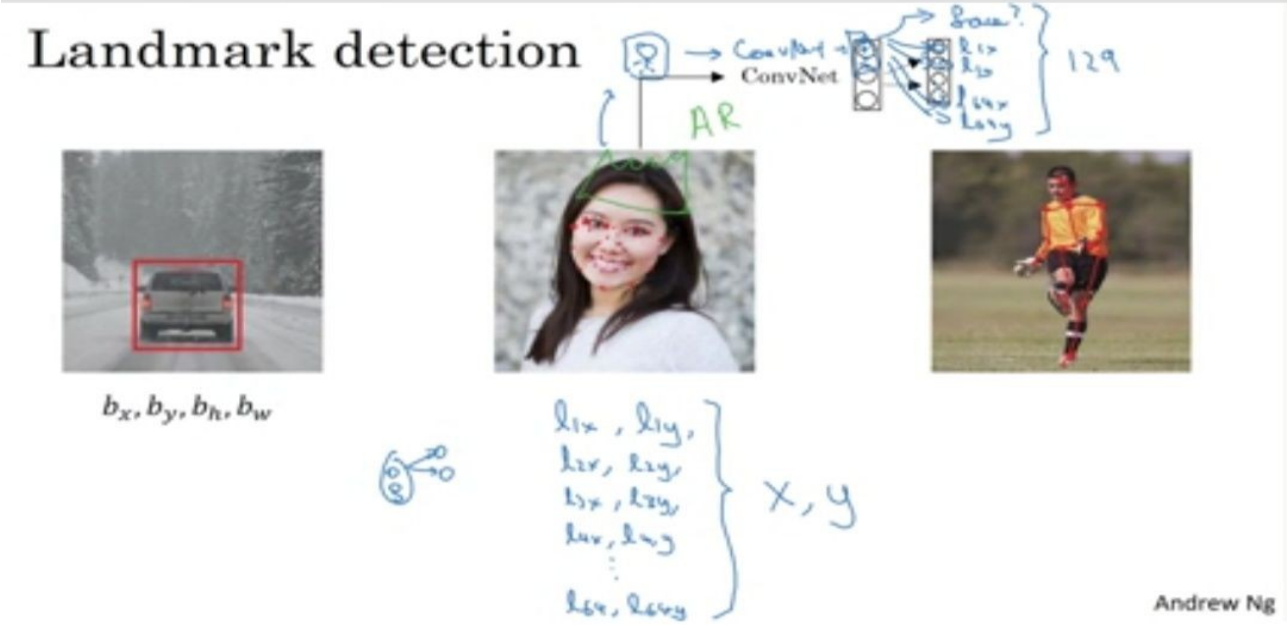


Figure 14-23. Intersection over Union (IoU) metric for bounding boxes

IoU always ranges between 0 and 1. one is the ideal performance and it means prediction is the same as the ground truth and vice versa for 0. When the intersection are will be larger , IoU will be close to 1. Most experts say the accuracy of the localization would be correct if IoU >= 0.5 , it means that both boxes are detecting the same thing.

Land Mark Detection

- What is the name land mark refers for ?
- Why should i need to use land mark ? isn't object localization enough ?
- How to understand land mark detection ?
- can I use IoU for landmark detection ?
- who will set the land marks for the image during training ?
- Does back propagation update the position of the landmarks ?



In the image localization we've seen adding a bounding boxes on it but if we want to recognize a person's face or standing position of a person , we will do coordinates of key points or land marks to recognize motions in the image.

Land Mark detection means key point detection , detecting key points on a certain object may be face or standing position of a person.

previously we saw how we can get a neural network to output four numbers of bx , by, bh and bw to specify the bounding box of an object we want a neural network to localize. In more general cases , we can have a neural network just output X and Y coordinates of important points and image , sometimes called landmarks that we want the neural networks to recognize.

Let's say we're building a face recognition application and for some reason we want the algorithm to tell us where is the corner of someone's eye. so that

point has an and Y coordinate so we can just have a neural network have it's final layer and have it just output two more numbers which we are going to call ℓx and ℓy to just tell the coordinates for the corner of the person's eye.

Now what if we want it to tell all four corners of the eye , really both eyes. So if we call the points , the first , the second , the third and fourth points going from left to right , then we could modify the neural network to output ℓ1x , ℓ1y for the first point and ℓ2x , ℓ2y for the second point and so on , so that the neural network can output the estimated position of all those four points of the person's face. But what if we don't want just those four points ? what if we want to point along the eye , or may be along the mouth , so we can extract the mouth shape and tell if the person is smiling or frowning , may be extract a few key points along the edges of the nose but you could define some number , for the sake of argument , let's say 64 points or 64 landmarks on the face.

So what we do is we have this image , a person's face as input , have it go through a convnet and have a convnet , then have some set of features , may be have it output 0 or 1 , like zero face changes or not and then have it also output ℓ1x , ℓ1y and so on down to ℓ64x , ℓ64y and here we are using ℓ to stand for a landmark. so this example would have 129 output units and one is our face or not ? Nd then we have 64 landmarks , that is 64* 2 so 128 plus one output units and this can tell us if there's a face as well as where all the key landmarks on the face.

So this is the basic building blocks for recognizing emotions from faces and if we played with the Snapchat and other entertainment, also AR augmented really filters like Snapchat photos can draw a crown on the face and have other special effects.

One last example , if we are interested in people pose detection , we could also define a few key positions like the midpoint of the chest , the left shoulder , left elbow , the wrist and so on and just have a neural network to annotate key positions in the person's pose as well and by having a neural network output , all of those points now annotating , we could have the neural network output the pose of the person. so this idea might seem quite simple of just adding a bunch of output units to output the X,Y coordinates of different landmarks we want to recognize.

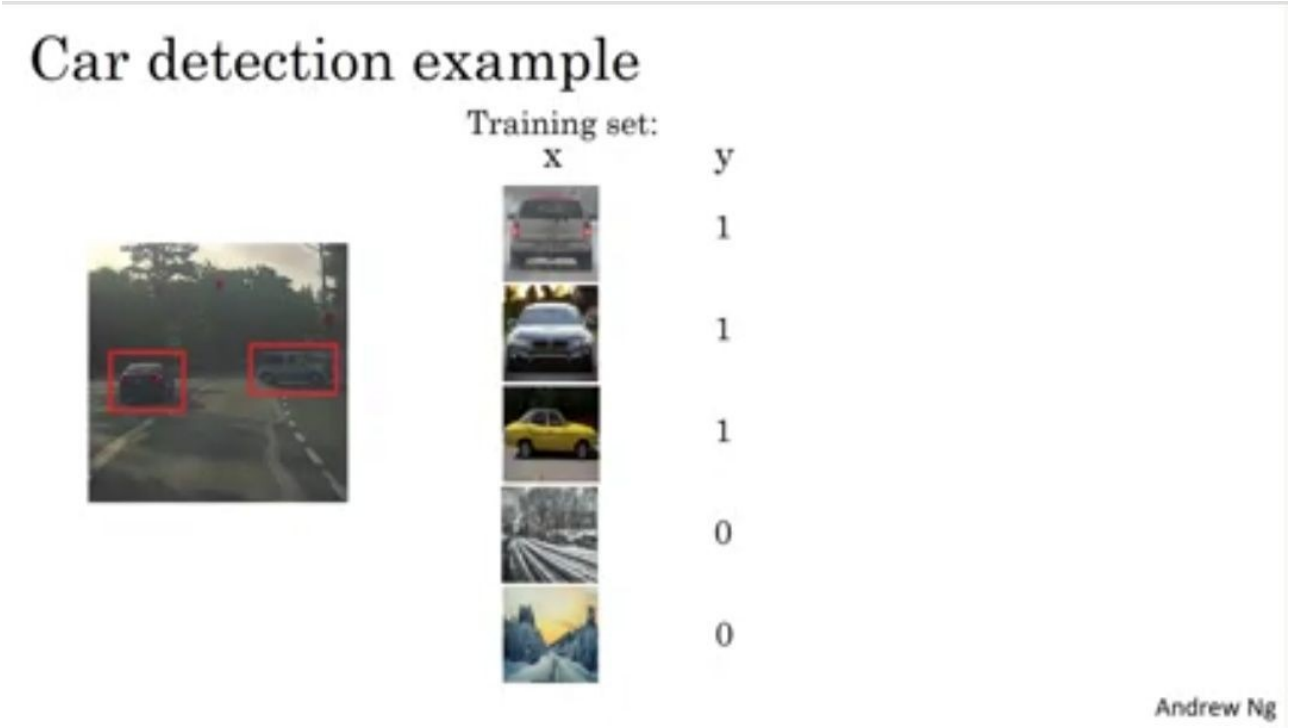
To be clear , the identity of the landmark one must be consistent across different images like may be landmark one is always the corner of the eye , landmark two is always the second corner of the eye , landmark three , landmark four and so one , so the labels have to be consistent across different images.

If we can hire labelers or label ourselves a big enough dataset to do this then a neural network can output all of these a landmarks which is going to used to carry out other interesting effect such as with the pose of the person , may be try to recognize someone's emotion from a picture.

Object Detection

we've seen about object localization as well as Landmark Detection. Now let's build up to other object detection algorithm. so Now we will learn how to use a ConvNet to perform object detection using something called the sliding windows Detection Algorithm.

Let's say we want to build a car detection algorithm. We first create a label training set , so x and y with closely cropped examples of cars. So this is image x has a positive example which is a car and there is not a car which is not a car.



So we can take picture and crop out and cut out anything else that' not a part of a car. So we end up with a car centered in pretty much the entire image. Given this label training set we can train a ConvNet that inputs an image , like one of these closely cropped images and then the job is to output y , zero or one , is there a car or not. Once we've trained up this ConvNet we can then use it in sliding window detection.

The way we do is , if we have a test image like one in the bottom , then we start by picking a certain window size and then we would input in to the convnet a small rectangular region so take just this below red square , input that in to ConvNet and have ConvNet make predictiona and presumably for this little region in the red square , it'll say no that little red square does not contain a car.

In the sliding windows detection Algorithm , what we do is we pass as input a second image now bounded by this red square shifted a little bit over and feed that to the ConvNet. so we are feeding just the region of the image in the red squares of the ConvNet and run the ConvNet again. and then we do that with a third image and so on. And we keep going until we've slid the window across every image and now we are using pretty large stride in this example just to make the animation go faster. but the idea is we basically go through every region of the size and pass lots of little cropped images in to the ConvNet and have it classified zero or one for each position of some stride. Now having done this once with running this was called the sliding window through the image.

We can then repeat it but now use a larger window so now we take a slightly larger region and run that region. so resize this region in to whatever input

size the ConvNet is expecting and feed that to the ConvNet anf have it output zero or one.

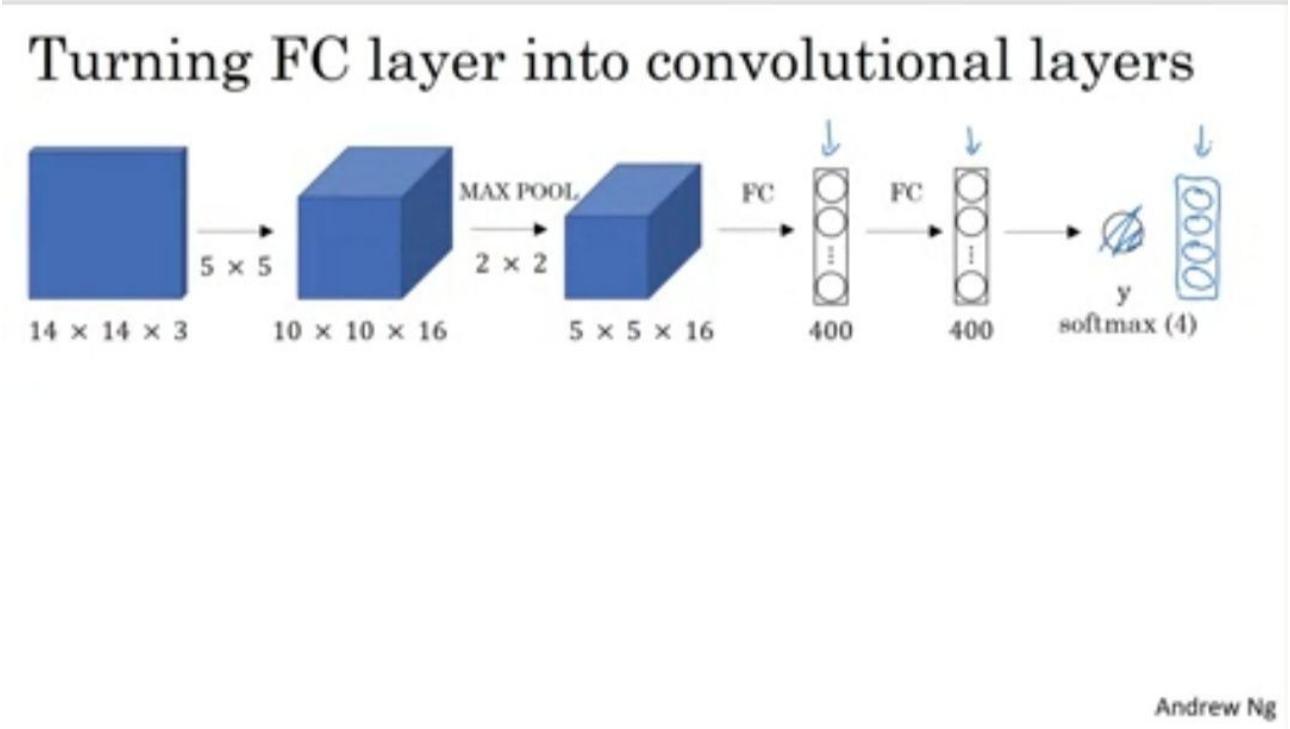
and then slide the window over again using some stride and so on. And we run that through our entire image until we get to the end and then we might do the third time using even larger windows and so on. so this algorithm is called sliding window detection because we take these windows , these square boxes and slide them across the entire image and classify every square region with some stride as containing a car or not. Now there's a huge disadvantage of sliding windows detection , which is the computational cost because we're cropping out so many different square regions in the image and running each of them independently through ConvNet and if we use a very coarse stride , a very big stride , a very big step size , then that will reduce the number of windows we need to pass through the ConvNet but that may hurt performance. whereas if we use a very small stride , then huge number of all these little regions we're passing through the ConvNet means that there is a very high computational cost. so before the rise of Neural Networks people used to use much simpler classifiers like a simple linear classifier over a hand engineer features in order to perform object detection and in that era because

each classifier was relatively cheap to compute , it was just a linear function , sliding windows detection ran okay , it was not a bad method but with ConvNet now running a single classification task is much more expensive and sliding windows this way is infeasibility slow. and unless we use a very small stride we end up not able to localize the objects that accurately. The sliding windows object detection can be implemented convolutionally or much more efficiently.

Convolutional Implementation Of Sliding Windows

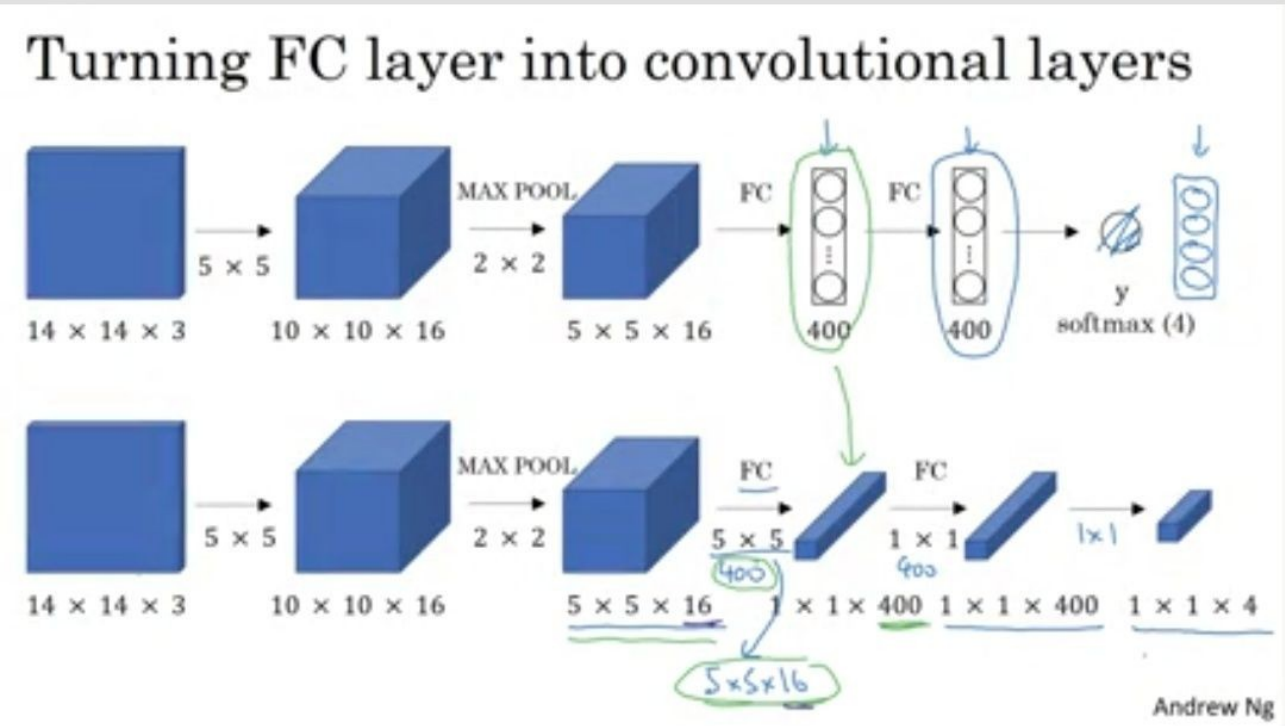
previously we learned about the sliding windows object detection algorithm using a ConvNet but we saw that it was too slow. To build up towards the convolutional implementation of sliding windows let's first see how we can turn fully connected layers in a neural network in to convolutional layers.

so let's say that our object detection algorithm inputs 14 * 14 by 3 images. this is quite small but just for illustrative purposes and let's say it then uses 5 by 5 filters and let's say it uses 16 of them to map it from 14 * 14 by 3 to 10 by 10 by 16 and then does a 2*2 max pooling to reduce it to 5*5*16. Then has a fully connected layer to connect to 400 units. Then now they're fully connected layer and then finally outputs a Y using softmax unit.

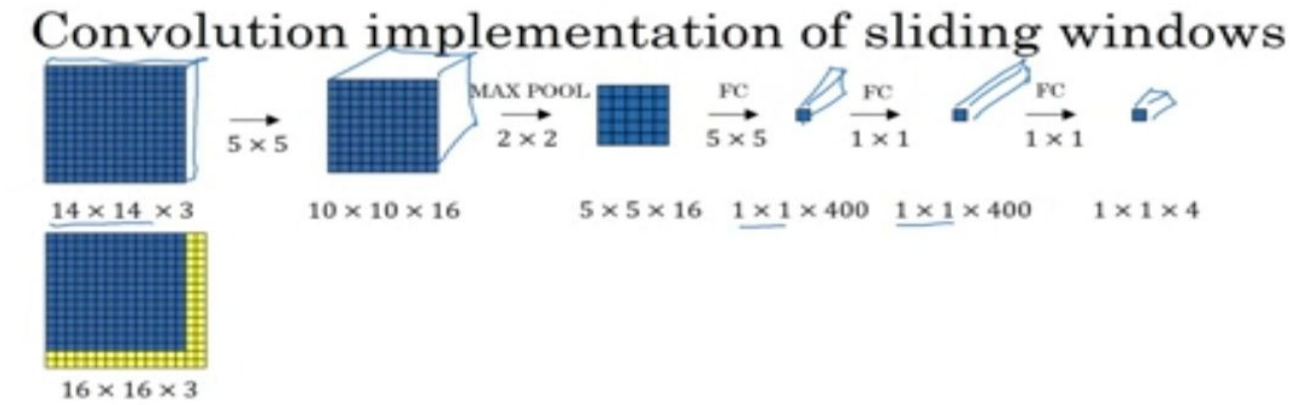


Now let's see how these layers can be turned in to convolutional layers and now one way of implementing this next layer , this fully connected layer is to implement this as a 5*5 filter and let's use 400 features with 5*5 filters.

so if we take 5 by 5 by 16 image and convolve it by 5 by 5 filter with 16 features then our output will be 1 by 1 and if we have 400 of these 5 by 5 by 16 filters , then the output dimension is going to be 1 by 1 by 400. so rather than viewing these 400 as just a set of nodes , we're going to view as a 1 by 1 by 400 volume. Next to implement the next convolutional layer , we're going to implement a 1*1 convolution , if we have 400 1 by 1 filters then with 400 filters the next layer will again be 1 by 1 by 400. and then finally we are going to have a 1 by 1 filter followed by a softmax activation , so as to give a 1 by 1 by 4 volume to take the place of these four numbers that the network was operating.



So now let's say that our convnet inputs 14 by 14 images or 14 by 14 by 3 images and our tested image is 16 by 16 by 3 , so now added that yellow stripe to the border of this image.

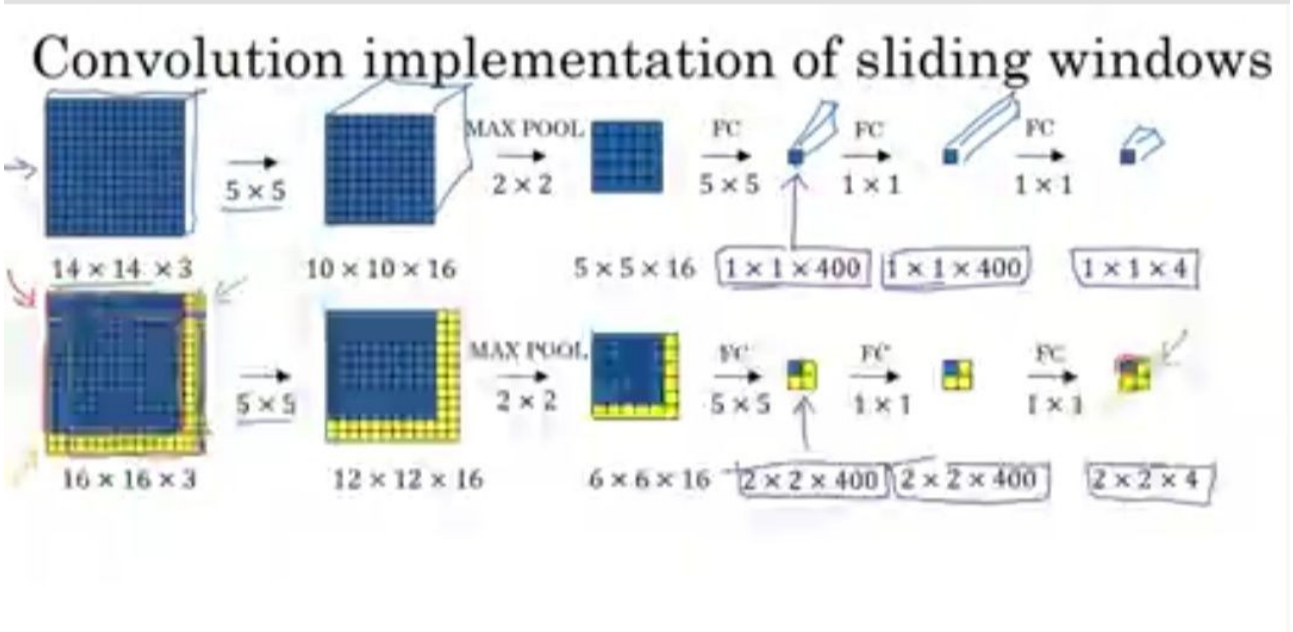


In the original sliding windows algorithm we might want to input the blue region in to convnet and run that once generate a classification 0 or 1 and then slightly down a bit , let's say we used a stride of two pixels and then we might slide that to the right by two pixels to input this green rectangle in to the convnet and we run the whole convnet and get another label 0 or 1. Then we might input this orange region in to the convnet and run it one more time to get another label and then do it the fourth and final time with this lower right purple square. To run sliding windows on this 16 by 16 by 3 image is pretty small image , we run this convnet four times in order to get four labels but it turns out a lot of this computation done by these four convnets is highly duplicative. so what the convolutional implementation of sliding windows does is it allows these four forward passes to the convnet to share a lot of computation.

Here's what we can do , we can take the convnet and just run it with the same parameters , the same 5 by 5 filters , also 16 five by 5 filters and run it. Now we can have 12 by 12 by 16 output volume , then do the max pool , same as before , now we have 6 by 6 by 16 , runs through our same 400 five by 5 to get now 2 by 2 by 400 volume. so now instead of a 1 by 1 by 400 volume , we have instead a 2 by 2 by 400 volume. Run it through a 1 by 1 filter gives you another 2 by 2 by 400 instead of 1 by 1 like 400.

Do that one more time and now we're left with a 2 by 2 by 4 output volume instead of 1 by 1 by 4. It turns out that this blue 1 by 1 by 4 subset gives us the result of running the upper left hand corner 14 by 14 image. This upper right 1 by 1 by 4 volumes gives us the upper right result.

The lower left gives us the results of implementing the convnet on the lower left 14 by 14 region and the lower right 1 by 1 by 4 volume gives us the same result as running the convnet on the lower right 14 by 14 medium.

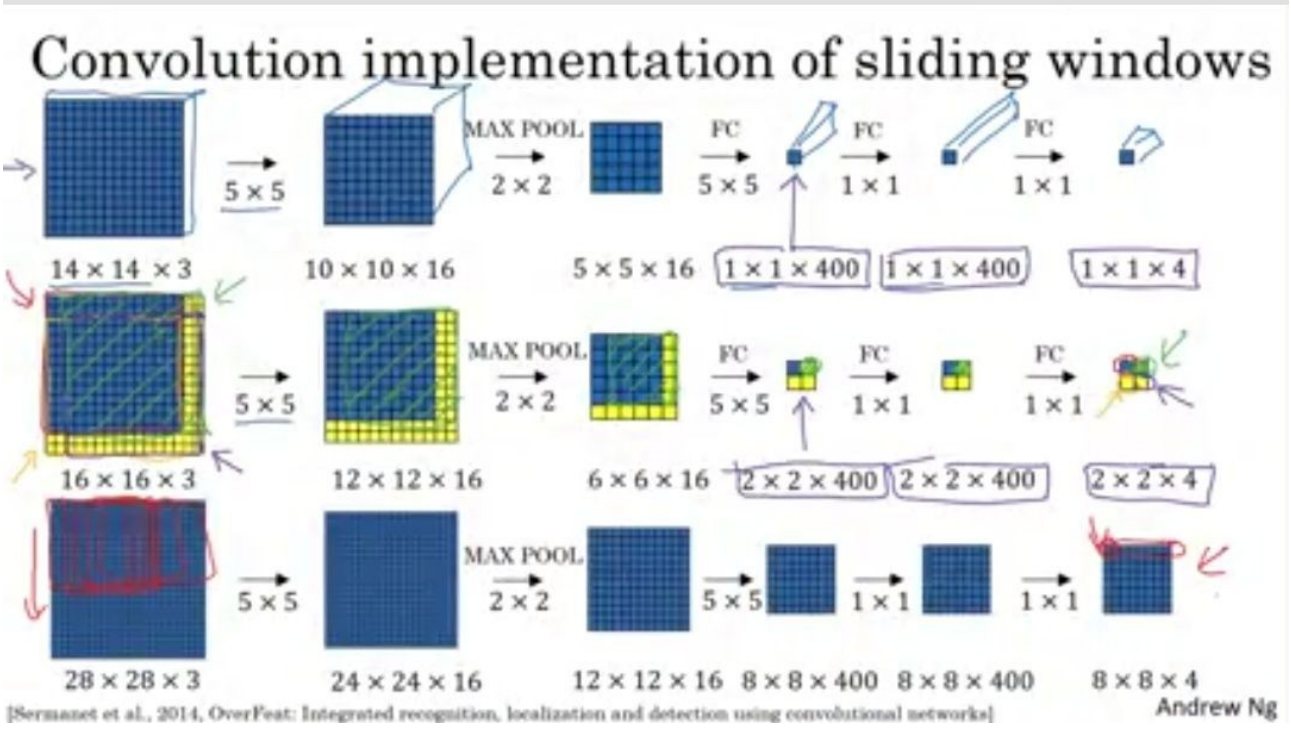


So what this process does , what this convolution implementation does is , instead of forcing us to run four propagation on four subsets of the input image independently instead it combines all four in to one form of

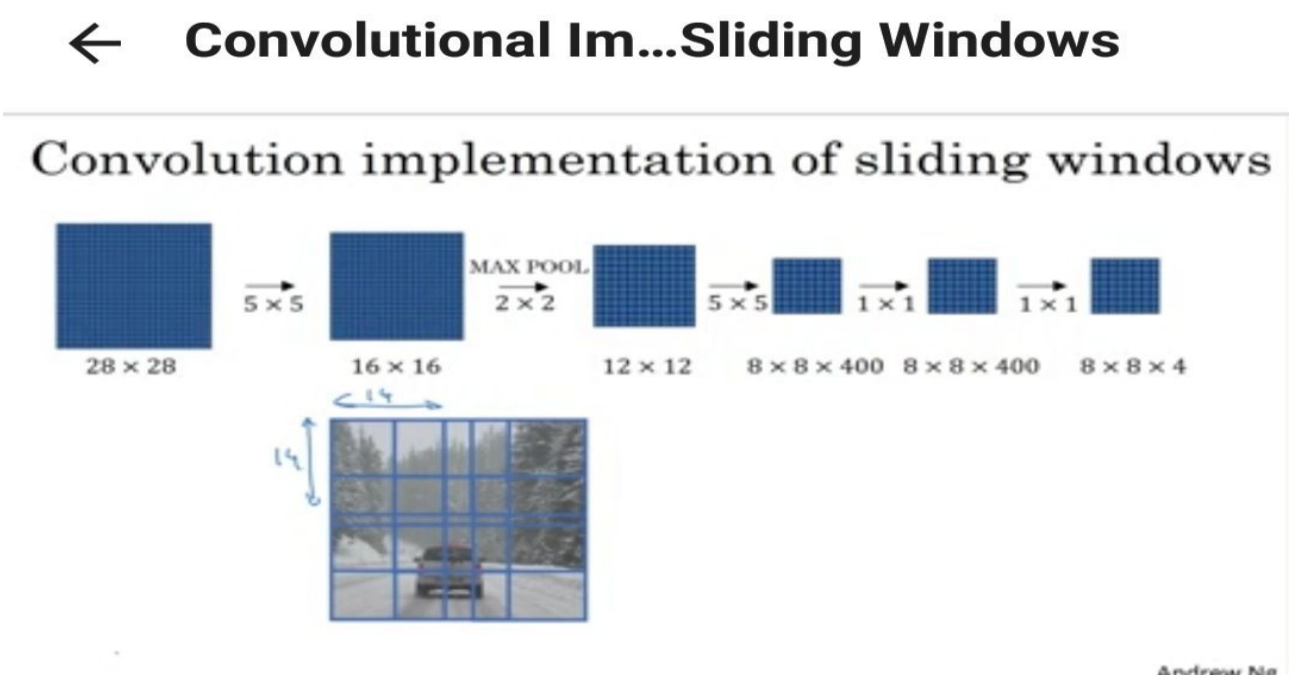
computation and shares a lot of the computation in the regions of the image that are common.

Now let's just go through a bigger example , let's say we now want to run sliding windows on a 28 by 28 by 3 image. It turns out if we run forward prop the same way we end up with 8 by 8 by 4 output.

← Convolutional Im...Sliding Windows



so to implement sliding windows , previously what we do is we crop out a region , let's say this is 14 by 14 and run that through our convnet and do that the next region over , then do that for the next 14 by 14 region, then the next one , the next one and the next one , until hopefully we recognizes the



car.

But now instead of doing it sequentially with this convolutional implementation, we can implement the entire image , all 28 by 28 and convolutionally make all the predictions at the same time by one forward pass through this big convnet and hopefully have it recognize the position of the car. But this algorithm still has one weakness which is the position of the bounding boxes is not going to be accurate.

Bounding Box Prediction

With sliding windows , we take this grid set of locations and run the classifier through it and in this case none of the boxes really match up perfectly with the position of the car.

← Bounding Box Predictions

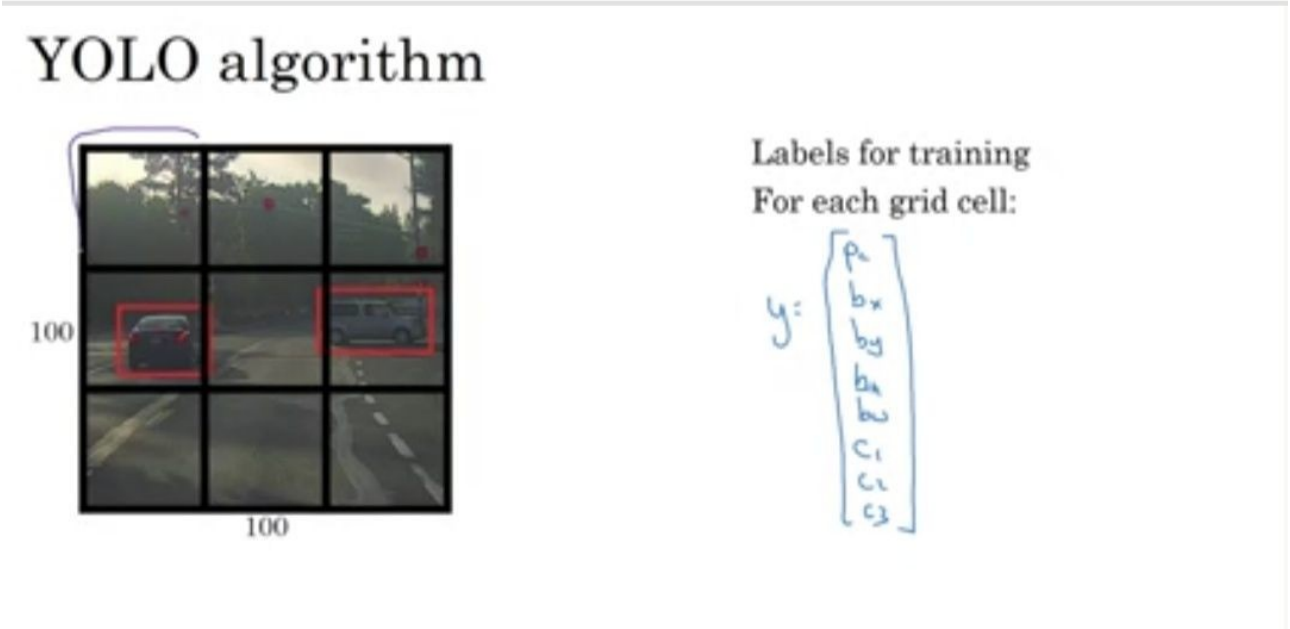
so may be the green box might be the best match but the perfect ground truth isn't even quite square , it's actually has a slightly wider rectangle or slightly horizontal aspect ratio , so is there a way to get this algorithm outputs more accurate bounding boxes ?

A good way to get this output more accurate bounding boxes is with the YOLO Algorithm. YOLO stands for , you only look once. Let's say we have an input image at 100 by 100 , we're going to place down a grid on this image and for the purposes of illustration we are going to use a 3 by 3 grid (the output after convolutional way of implementing) although in an actual implementation we use a finer one , like may be a 19 by 19 grid. and the basic idea is we're going to take the image classification and localization algorithm that we saw previously and apply it to each of the nine grids. which means apply the image classification and localization after the object detection for finding accurate bounding box prediction.

so the basic idea is we're going to take the image classification and localization algorithm that we saw and apply that to each of the nine grid cells of this image. So for each of the nine grid cells we specify a label y, where the label y is this eight dimensional vector , we first output Pc , it will be 0 or 1 depending on whether or not there's an image in that grid cell and

then Bx , By , Bh , Bw to specify the bounding box if there is an image , if there is an object associated with that grid cell and then C1,C2,C3 if we have three classes without counting the background class. so we try to recognize pedastrian class , motorcycles and motor cycle classes. so in this image we have nine grid cells so we have a vector like this for each of the grid cells.

← Bounding Box Predictions



So let's start with the upper left grid cell , for that no there's no object. so the label vector Y for the upper left grid cell would be zero and then don't cares for the rest of the vector. The output label Y would be the same for the second and third grid cells which is there is no interesting object in them , but how about the four grid cell ?

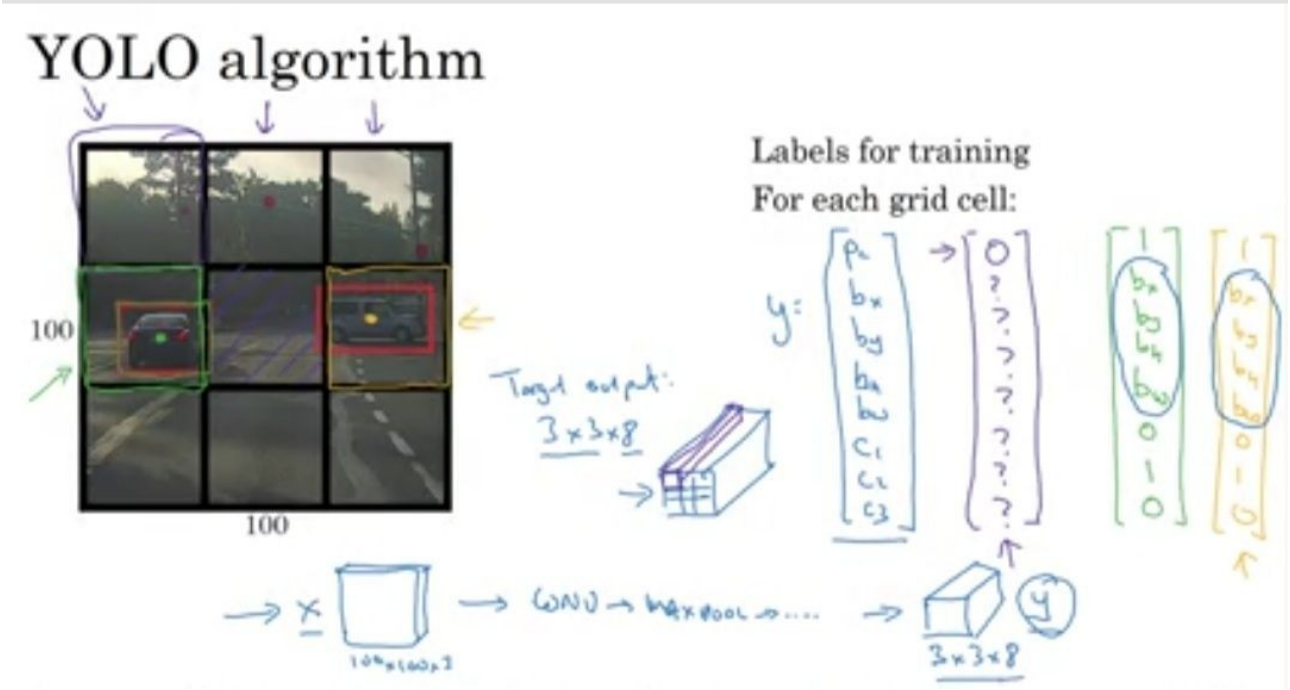
To give a bit more detail , this image has two objects and what the YOLO algorithm does is it takes the mid point of each the two objects and then assigns the object to the grid cell containing the mid point. How do we find the mid point ? So the left car is assigned to the fourth grid cell and the car in the right is this midpoint is assigned to the sixth grid cell. and so even though the central grid cell has some parts of both cars , we'll pretend the central grid cell has no interesting object so that the central grid cell the class label Y also looks like a vector with no object and the first component Pc will be 0 and the rest 7 components will be don't cares.

So for each of these nine grid cells , we end up with eight dimensional output vector and because we have nine grid cells , the total volume of the output is going to be 3 by 3 by 8 so the target output is going to be 3 by 3 by 8 because we have 3 by 3 grid cells.

So now to train a neural network with input image of 100 by 100 by 3 , that's the input image and then we have a usual convnet with conv layers , max pool layers and so on , so that in the end our output will be 3 by 3 by 8 output volume. and so what we do is we have an input X which is the input image and we have these target labels Y which are 3 by 3 by 8 and we use back propagation to train the neural network to map the input X to output y.

So the advantage of this algorithm is that the neural network outputs precise bounding box , so at the test time what we do is we feed an input image X and run forward propagation until we get this output Y.

← Bounding Box Predictions



And then for each of the nine outputs if each 3 by 3 positions in which of the output , we can just read 1 or 0 , is there an object associated with that one of this nine positions ?

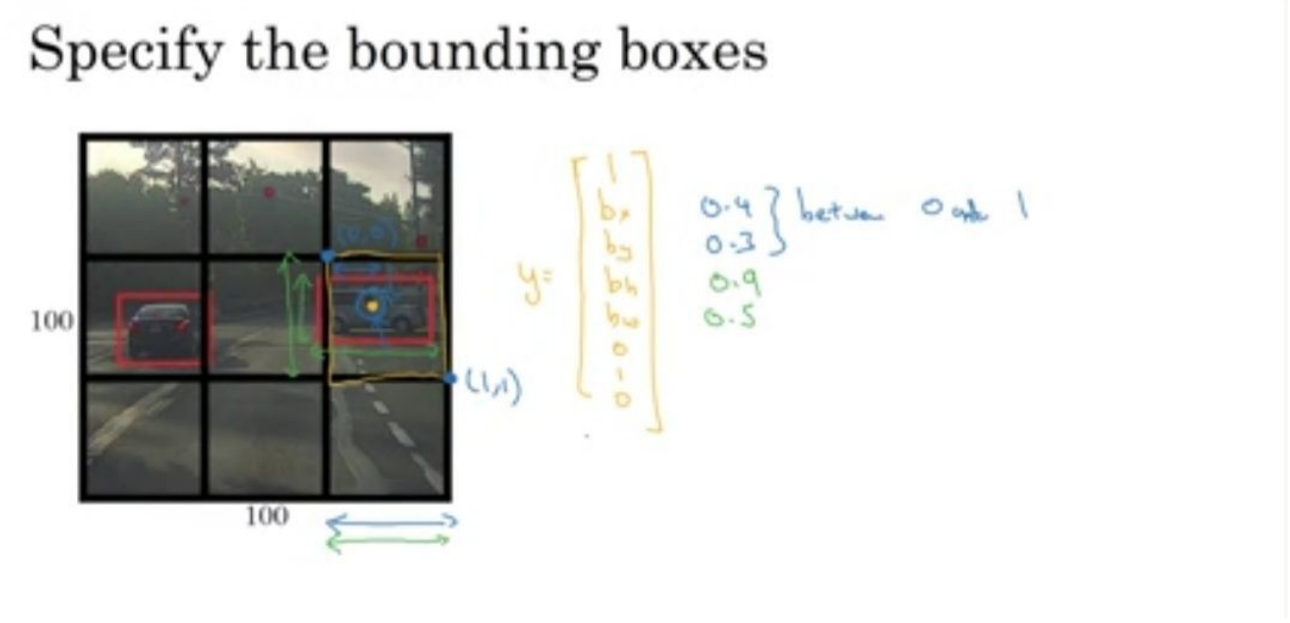
And that there is an object , what object it is, and where is the bounding box for the object in that grid cell ? and as long as we don't have more than one object in each grid cell , this algorithm should work okay. and the problem of having multiple object with in a grid cell is something we will address later. of use relatively small 3 by 3 grid , in practice we might use a much finer , grid may be 19 by 19 so we end up with 19 by 19 by 8 and that also makes our grid much finer. It reduces the chance that there are multiple objects assigned to the same grid cell. and just as a reminder , the way we assign an object to the grid cell as we look at the mid point of an object and then we assign that object to whichever one grid cell contains the mid point of the object. so each object , even if the objects spends multiple grid cells , that object is assigned only to one of the nine grid cells or one of the 3 by 3 or one of the 19 by 19 grid cells.

So notice two things, first , this is a lot like Image Classification and Localization algorithm that we talked about and that it outputs the bounding boxes cordinates explicitly and so this allows in our network to output bounding boxes of any aspect ration , as well as , output much more precise coordinates that are not just dictated by the stripe size of our sliding windows classifier. and second , this is convolutional implementation and we're not implementing this algorithm nine times on the 3 by 3 grid or if we are using a 19 by 19 grid , 19 squared is 361 , so we are not running the same algorithm 361 times or 19 squared times. Instead this is one single convolutional implementation , where we use one conv net with a lot of shared computation needed for all of out 3 by 3 or all of our 19 by 19 grid cells , so this is pretty efficient algorithm.

So one nice thing about YOLO algorithm, which is gains popularity is because of this is a convolutional implementation , it actually runs very vast so this works even for real time object detection.

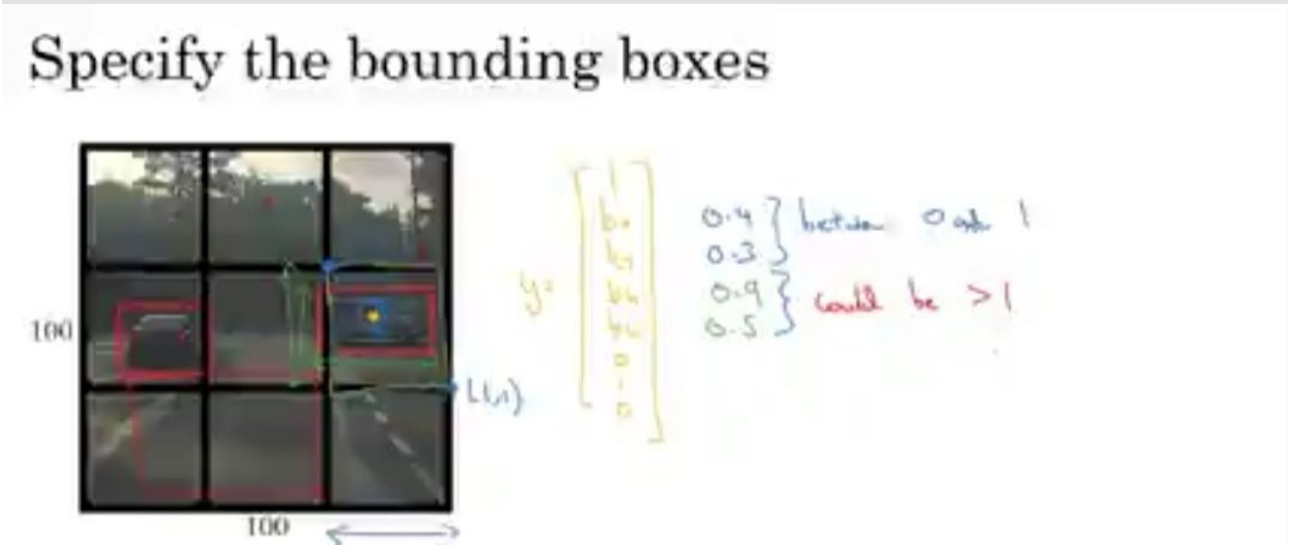
Question , How do we encode these bounding boxes , bx , by , bh and bw. Let's take the example of the car on the right. So in this grid cell there's an object and so the target label y will be on , that was Pc is equal to one and then bx , by , bh and bw and then 0 , 1 , 0 so how do we specify the bounding boxes ?

Bounding Box Predictions



In the YOLO algorithm , relative to this square(grid in the right car) , when we take the convention that the upper left point here is (0 , 0) and this lower right point is (1 , 1) so to specify the position of that mid point , that orange dot , bx might be , let's say it's about 0.4 , which means it's about 0.4 way to the right and then y , looks i guess may be 0.3 and then the height of the bounding box is specified as a fraction of the over all width of this box , so the width of this red box is may be 90% of that blue line and also Bh is 0.5 and the height of this is may be one half of the overall height of the grid cell , so this is how we set bx , by , BH , BW as relative to the grid cell. and so bx and by has to be between 0 and 1 because pretty much by definition that orange dot is with in the bounds of that grid cell is assigned to , if it wasn't between 0 and 1 then it means it was outside the square , then we'll have been assigned to a different grid cell but BH and BW could be greater than one.

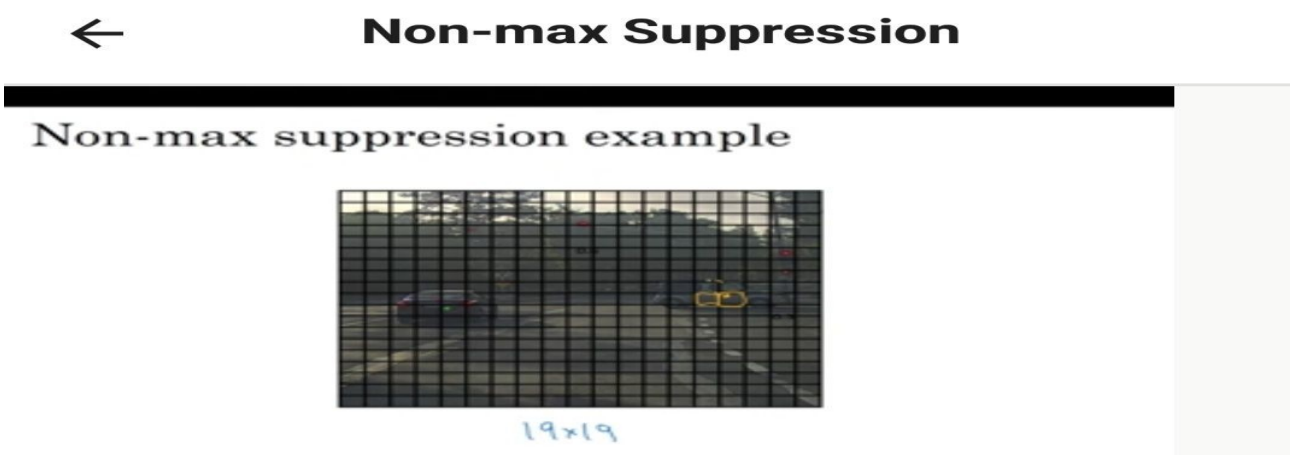
← Bounding Box Predictions



In particular if we have a car where the bounding box was that(the big red line in the bottom of the above image) , then the height and width of the bounding could be greater than one.

Intersection Over Union :- This is just the same as the one we saw in the object localization part , we also use now , in YOLO , first object detection , then image classification and object localization and we will use IOU for evaluating our object localization.

Non-Max Suppression :- One of the problems of object detection as we have seen so far is that our algorithm may find multiple detections of the same objects rather than detecting an object once , it might detect it multiple times. Non max suppression is a way to make sure that our algorithm detects each



object only once.

Let's say we want to detect pedestrians , cars and motorcycles in this image. we might place a grid over this and this is 19 by 19 grid. Now while technically this car has just one midpoint , so it should be assigned just one grid cell and the car on the left also has just one midpoint , so technically only one of those grid cells should predict that there is a car. In practice, we 're running an object classification and localization algorithm for everyone of these splits (these are outputs from the convolutional way of sliding windows). so it's quite possible that this split cell might think that the center of a car is in it and so might this and so might this and for the car on the left as well.

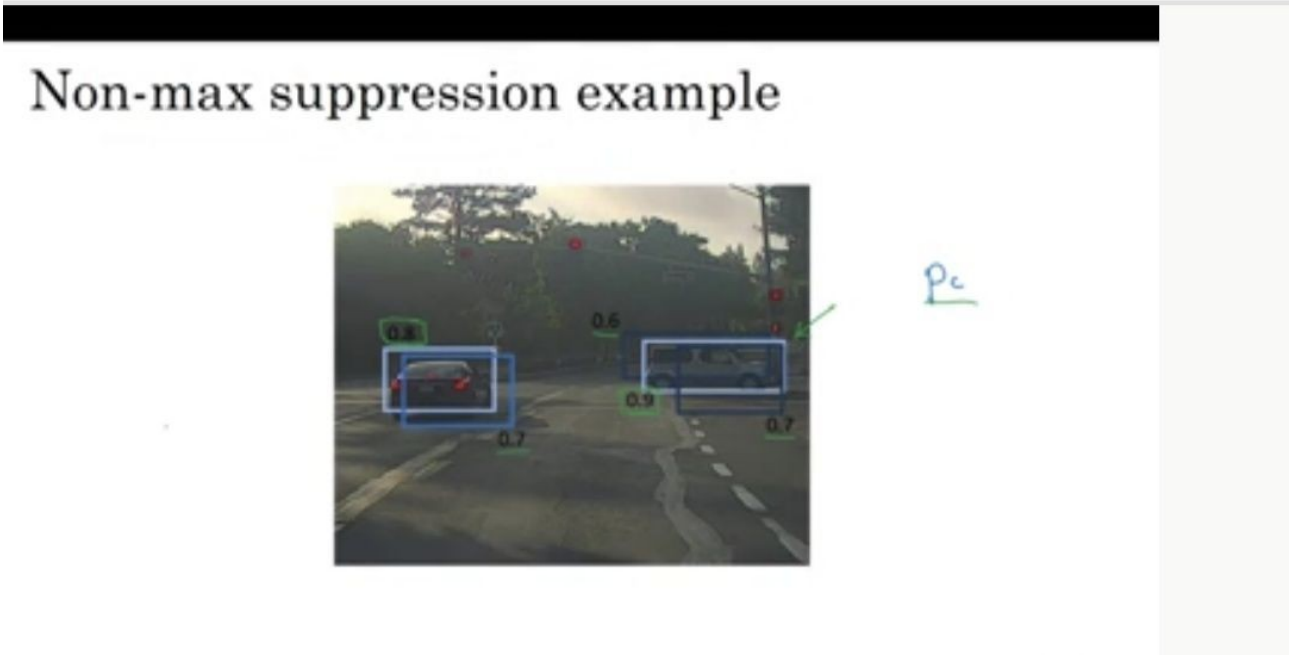
If this is a test image we've seen before , not only one box might decide things that's on the car may be that box but also other boxes will also think that they've found the car.

So because we're running the image classification and localization algorithm on every grid cell , on 361 grid cells , it's possible that many of them will raise their hand and say "My Pc , my chance of thinking i have an object in it is large". So when we run our algorithm , we might end up with multiple detections of each object so what non - max suppression does , it cleans up these directions. so they end up with just one detection per car , rather than multiple detections per car , so concretely what it does is , it first looks at the probabilities associated with each of these detections.

And it first takes the largest one which in this case is 0.9 and says “That’s my most confident detection, so let’s highlight that and just say i found the car there” , having done that the non-max suppression part then looks at all of the remaining rectangles and all the ones with a high overlap, with a hight IOU, with this one that we ‘ve just output will get suppressed. so those two rectangles with the 0.6 and the 0.7

Both of those overlap a lot with the light blue rectangle. so those, we are going to suppress and darken them to show that they are being suppressed. next, we then go through the remaining rectangles and find the one with the highest probability , the highest Pc , which in this case is the one with 0.8 , so let’s commit to that and just say “I’ve detected a car there”. and then the non-max suppression part is to get rid of any other ones with a high IOU , so now , every rectangle has been either highlighted or darkened rectangles , we are left with just the highlighted ones and these are our two final predictions.

← Non-max Suppression



so this is non-max suppression and non-max means that we’re going to output our maximal probabilities classifications but suppresses the close-by ones that are no non-maximal. Hence the name , non-max suppression. Let’s go through the details of the algorithm.

Let's say we have 19 by 19 grid and let's get rid of the C1,C2,C3 and pretend we are getting an output that the chance there's an object and then the bounding box.

Now , to intimate non-max suppression , the first thing we can do is discard all the boxes , discard all the predictions of the bounding boxes with Pc less than or equal to some threshold , let's say 0.6//

So we are going to say that unless we think there's at least a 0.6 chance it is an object there , let's just get rid of it. This has discard all of the low probability output boxes. The way to think about this is for each of the 361 positions , we output a bounding box together with a probability of that bounding box being a good one. So we're just going to discard all the bounding boxes that were assigned a low probability.

Next , while there are any remaining bounding boxes that we've not yet discarded or processed , we are going to repeatedly pick the box with the highest probability , with the highest Pc and then output that as a prediction. So of taking one of the bounding boxes and making it a lighter color so we commit to outputting that as a predictions for that is a car there,next we then discard any remaining box. Any box that we have not output as a prediction and that was not previously discarded as a prediction , so discard any remaining box with a high overlap with a high IOU with the box that we just output in the previous step.

And so we keep doing this while there's still any remaining boxes that we've not yet processed , until we've taken each of the boxes and either output it as a prediction or discarded it as having too high an overlap or too high an IOU , with one of the boxes that we have just output as our predicted position for one of the detected objects.

We described the algorithm using just a single object , if we actually tried to detect three objects say pedestrians , cars and motorcycles , then the output vector will have three additional components and it turns out , the right thing

to do is to independently carry out non-max suppression three times , one on each output classes.

← Non-max Suppression

Non-max suppression algorithm

19× 19

Each output prediction is: p_c, b_x, b_y, b_h, b_w

Discard all boxes with $p_c \leq 0.6$

While there are any remaining boxes:

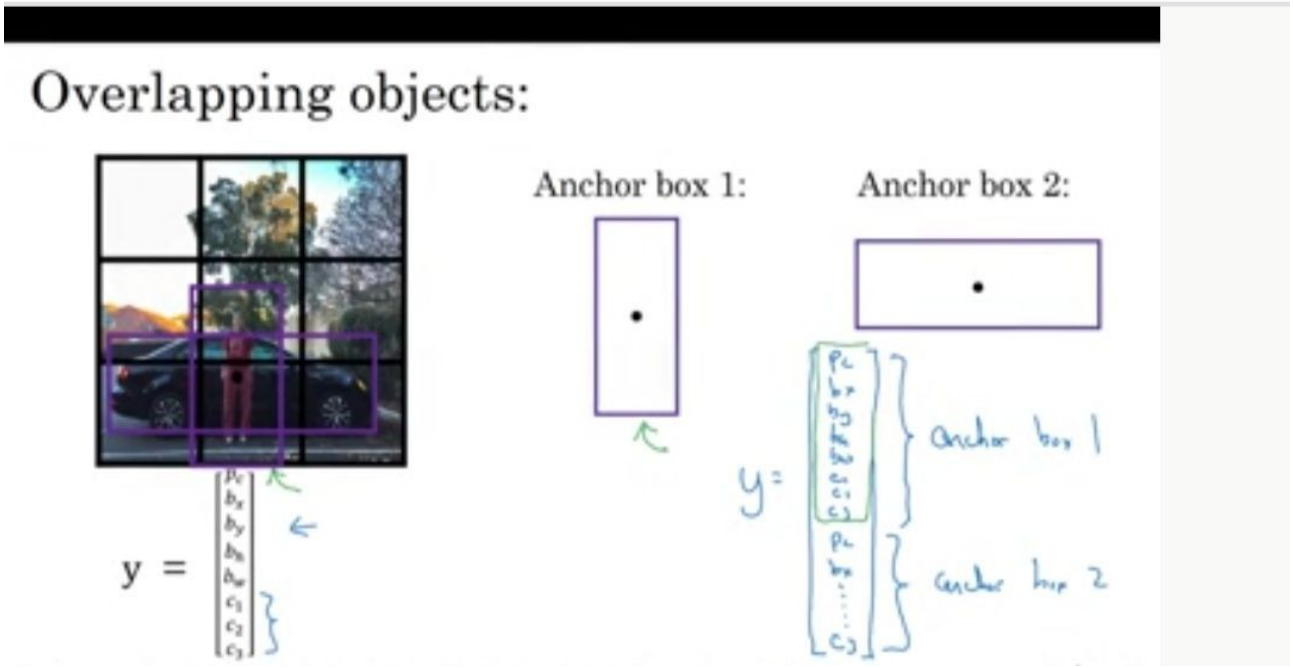
- Pick the box with the largest p_c Output that as a prediction.
- Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step

Andrew Ng

Anchor Boxes :- One of the problems with object detection as we seen it so far is that each of the gird cells can detect only one object. what if a grid cells can detect multiple objects ? we will use the idea of anchor boxes.

Let's say we have an image with a 3 by 3 grid , notice that the midpoint of the pedestrian and the mid point of the car in almost the same place and both of them fall in to the same grid cell. so for that grid cell , if Y outputs this vector where we are detecting three classes , pedestrians , cars and motorcycles , it won't be able to output two detections. So we have to pick one of the two detections to output.

Anchor Boxes



With the idea of anchor boxes , what we are going to do is pre-define two different shapes called anchor boxes or anchor box shapes and what we are going to do is to be able to associate two predictions with the two anchor boxes.

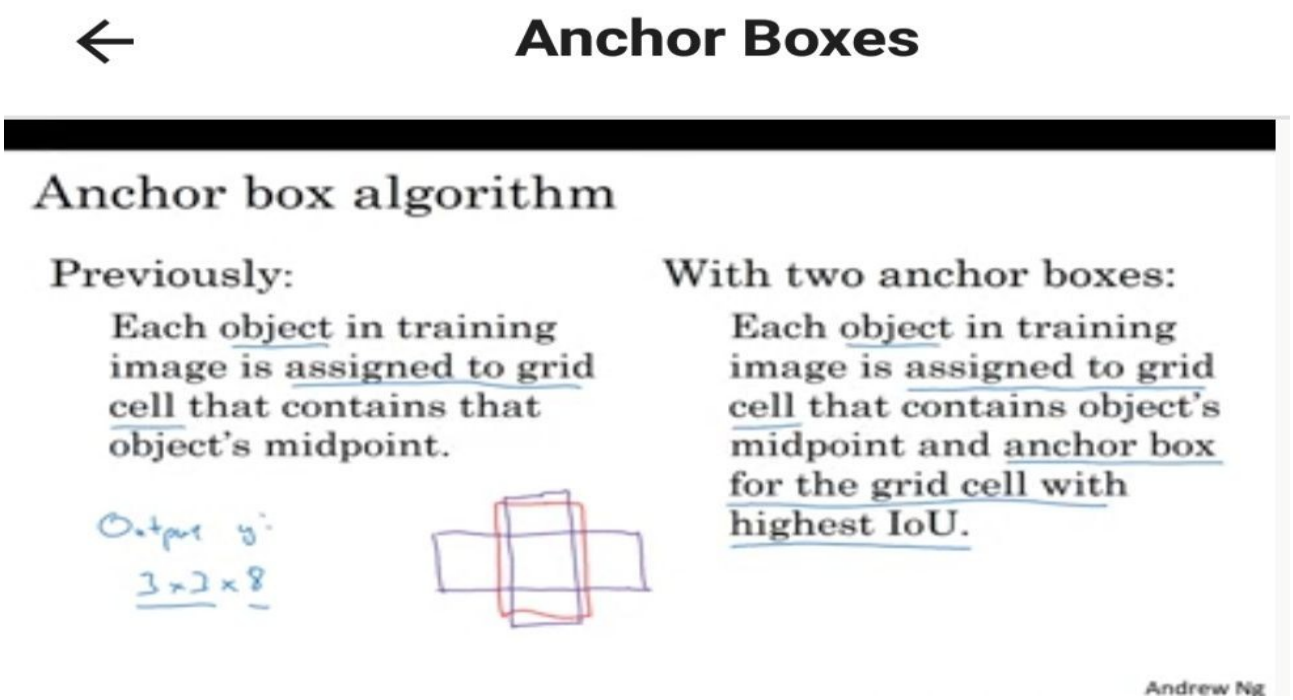
In practice we might use more anchor boxes , may be five or even more but for now we are going to use two anchor boxes just to make the description easier. so what we do is we define the cross label to be , instead of the vector on the left (one vector contain eight elements) , we basically will repeat that twice. So we will have Pc , Px , Py , Ph , Pw , C1 , C2 , C3 and these are eight outputs associated with anchor box 1 and then we repeat that Pc , Px and so on down to C1 , C2 , C3 and other eight outputs associated with anchor box 2.

So because the shape of the pedestrian is more similar to the shape of anchor box 1 than anchor box 2 , we can use these eight numbers to encode that PC as one , yes there is a pedestrian. so we will use to encode the bounding box around the pedestrian and then use this to encode that object is a pedestrian.

And then because the box around the car is more similar to the shape of anchor box two than anchor box 1 , we can then use this to encode that the second object here is the car and have the bounding box and so on be all the parameters associated with the detected car.

So to summarize , previously before we are using anchor boxes , we did the following ,which is for each object in the training set and the training set image it was assigned to the grid cell that corresponds to that object's midpoint and so the output Y was 3 by 3 by 8 because we have a 3 by 3 grid and for each grid position we had that output vector which is Pc , then the bounding box and C1 , C2 , C3 with the anchor box.

Now each object is assigned to the same grid cell as before , assigned to the grid cell that contains the object's midpoint but it is assigned to a grid cell and anchor box with the highest IoU with the object's shape so we have two anchor boxes we will take an object and see.

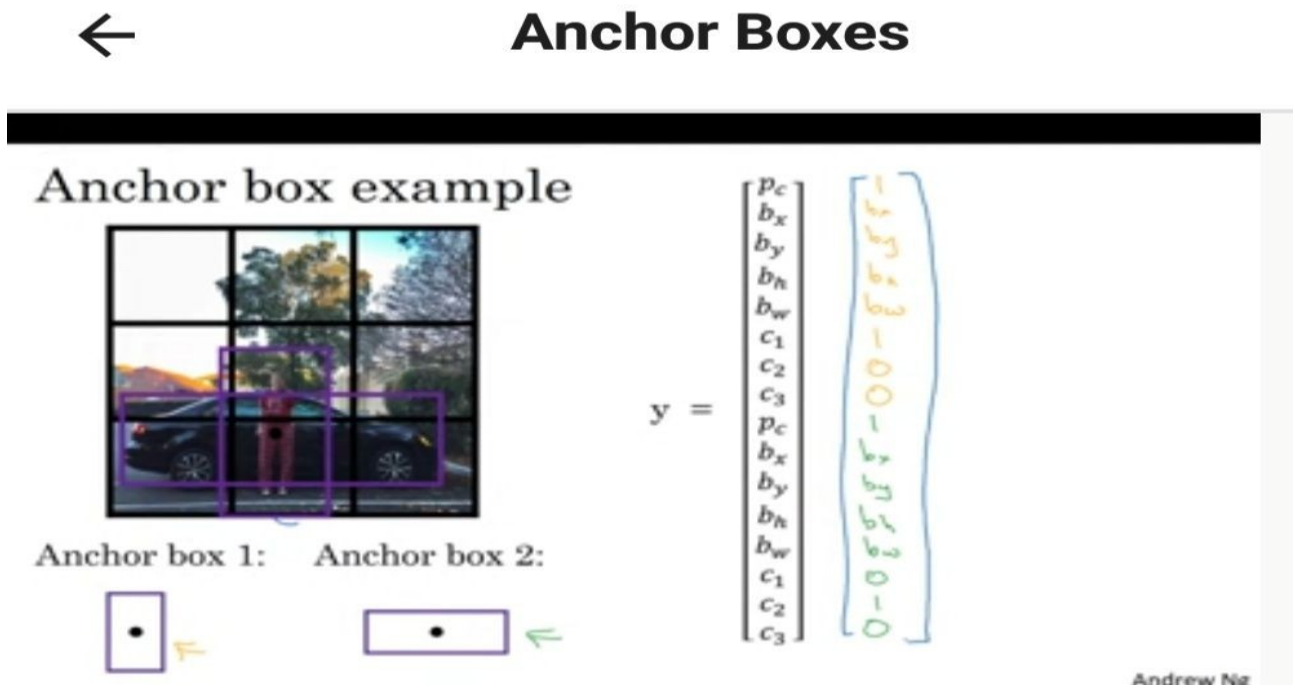


So if we have an object with a shape like long square(the red one) , what we do is take our two anchor boxes may be one anchor box is a shape looks like the object (long square) , that is anchor box 1 and then we see anchor box 2 with a shape of short widened square and then we see which of the two anchor boxes has a higher IoU , will be drawn through bounding box.

And whichever it is , that object then gets assigned not just to a grid cell but to a pair. It gets assigned to grid cell , anchor box pair and that's how that objects gets encoded in the target label and so now , the output Y is going to be 3 * 3 * 16 //

Because as we saw , Y is now 16 dimensional or if we want we can also view this as 3 by 3 by 2 by 8 , because there are now two anchor boxes and Y is eight dimensional.

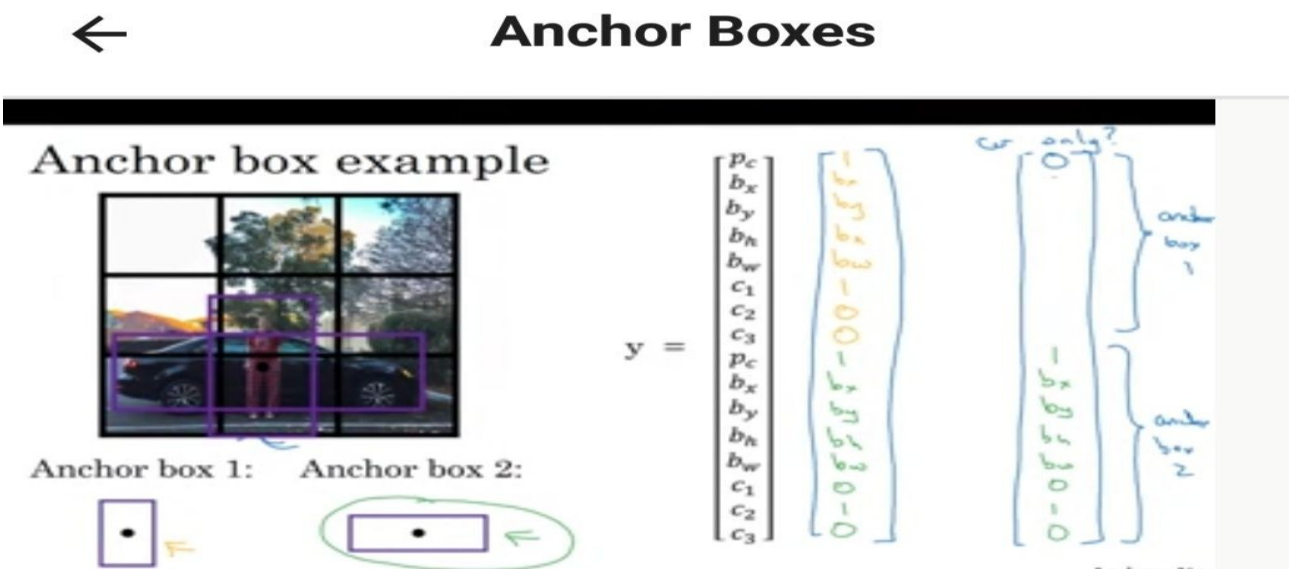
And dimension of Y being eight was because we have three objects causes , if we have more objects than the dimension of Y would be even higher. so for this grid cell , let's specify what is Y , so the pedestrian is more similar to the shape of anchor box 1 , so for the pedestrian we're going to assign it to the top half of this vector and then the shape of the car is more similar to anchor box 2.



Now , what if this grid cell only had a car and had no pedestrian ? If it only had a car, then assuming that the shape of the bounding box around the car is still more similar to anchor box 2 , then the target label Y , if there was

just a car there and the pedestrian had gone away , it will still be the same for the anchor box 2 component.

Remember that this is a part of the vector corresponding to anchor box 2 and for the part of the vector corresponding to anchor box 1 , what we do id we



just say there is no object there , so Pc is zero and then the rest of these will be don't cares.

What if we have two anchor boxes but three objects in the same grid cell ? That's one case that the algorithm doesn't handle well , Hopefully it won't happen but it does , this algorithm doesn't have a great way of handling it. or what if we have two objects associated with the same grid cell but both of them have the same anchor box shape ? Again , that's another case that this algorithm doesn't handle well.

The chance of two objects having the same midpoint out of these 361 cells , it does happen but it doesn't happen that often. In particular if our dataset has some tall skinny objects like pedestrians and some white objects like cars then this allows a learning algorithm to specialize in detecting wide fat objects like cars and some of the output units can specialize in detecting tall , skinny objects like pedestrians.

So finally how do we choose the anchor boxes ? And people used to choose them by hand or choose may be five or ten anchor boxes that spans a variety

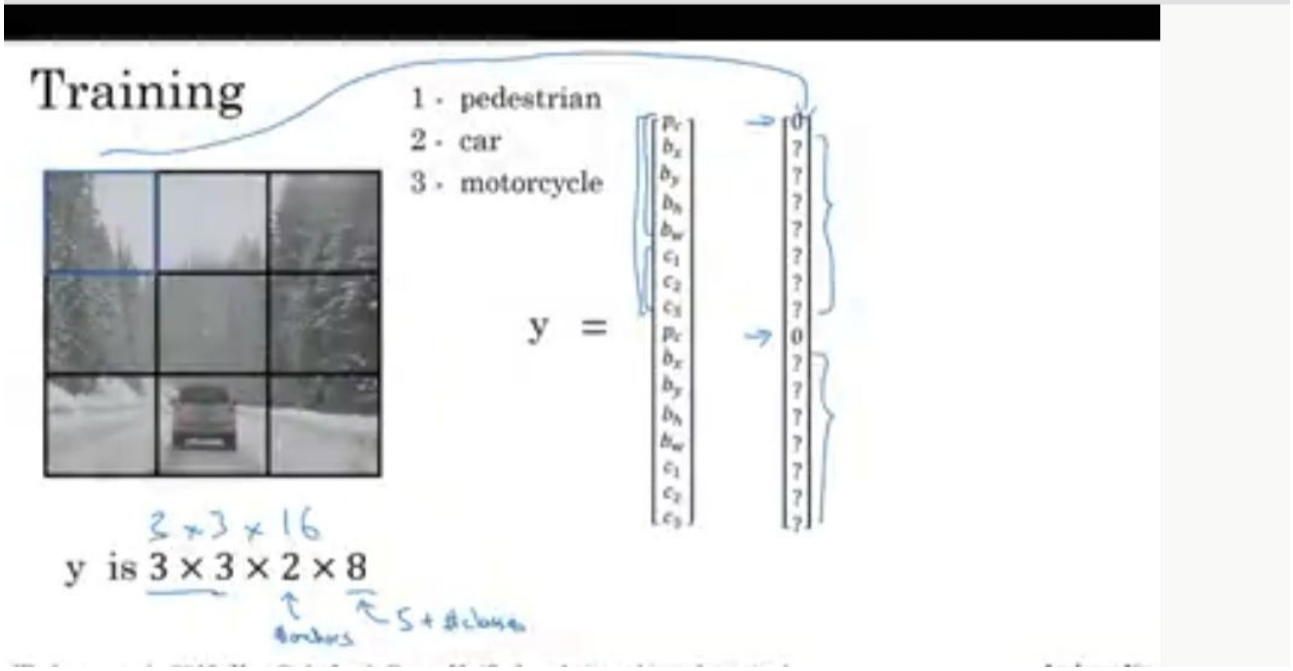
of shapes that seems to cover the types of objects we seem to detect. As a much more advanced version what we will do is to use K means algorithm , to group together types of object shapes we tend to get and then to use that to select a set of anchor boxes that this most representative of the may be multiple , of the may be dozens of object causes we ‘re trying to detect. This a way of automatically choose the anchor boxes but if we choose by hand a variety of shapes that reasonably expands the set of object shapes , we expect to detect some tall skinny ones , some fat wide ones so that will be our anchor boxes.

YOLO Algorithm

suppose we are trying to train an algorithm to detect three objects : pedestrians , cars and motor cycles and we will need to explicitly have the full background class. If we are trying to use two anchor boxes then the outputs Y will be three by three because we are using three by three grid cell , by two , this is the number of anchor boxes , and by eight because that’s the dimension , eight is actually five which is plus the number of classes , so this is because we have Pc and then the bounding boxes , that’s five and then C1,C2,C3. So we can either view it as three by three by two by eight or three by three by sixteen.

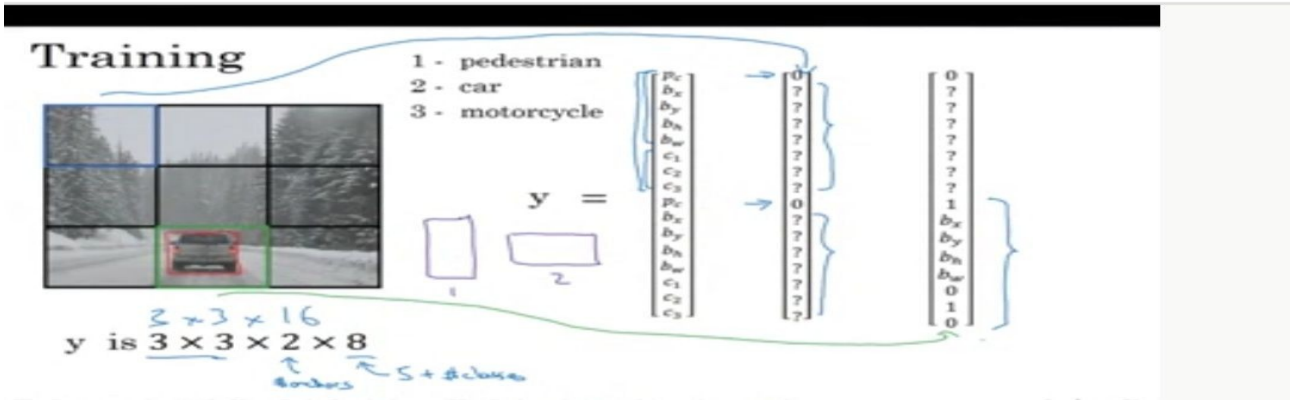
None of the three classes pedestrian,car and motorcycle appear in the upper left grid cell and the target y corresponding to that grid cell would be equal to this. Where Pc for the first anchor box is zero because there’s nothing associated for the first anchor box and is also zero for the second anchor box and so on , all of values will be don’t cares.

YOLO Algorithm



Now most of the grid cells has nothing in them but for the box in the 8th grid we would have a target label y so assuming that our training set has a bounding box like this for the car , it's just a little bit wider than it's a tall (So the IoU is higher for the anchor box 2). and so if our anchor boxes are that , then the red box (object) has just slightly higher IoU with anchor box two and so the car is associated with lower portion of the vector so notice that Pc associated with anchor box one is zero.

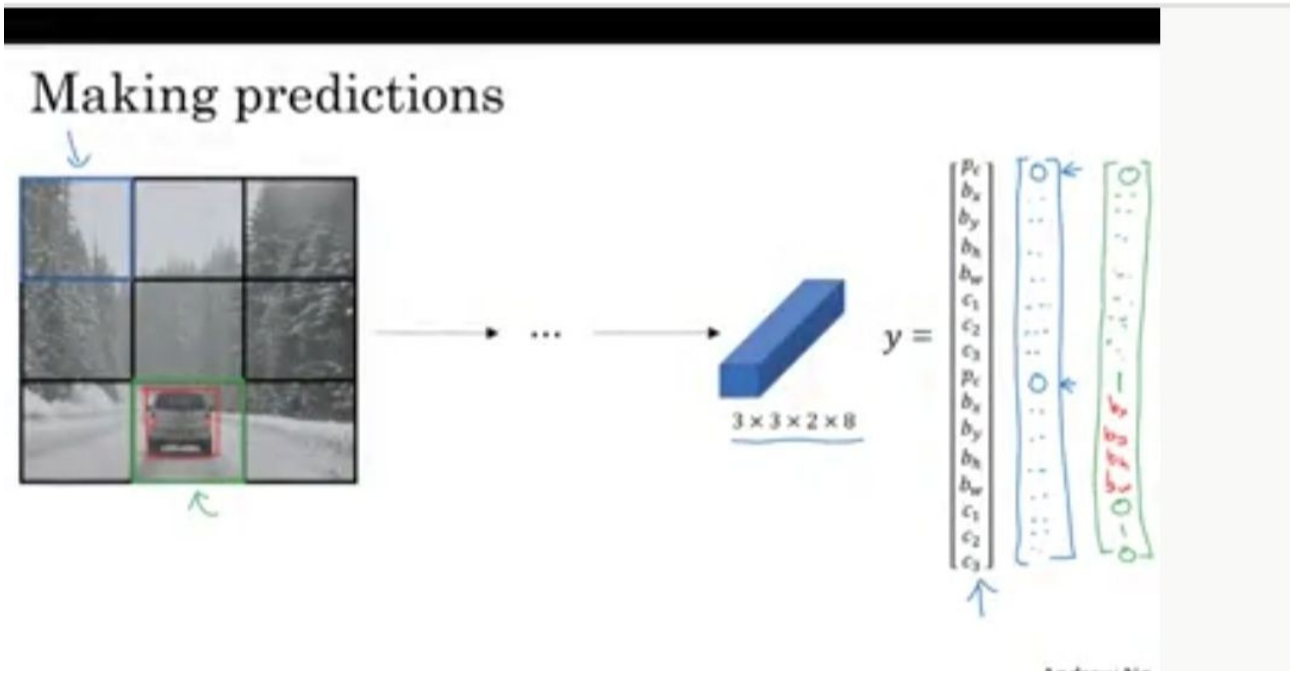
YOLO Algorithm



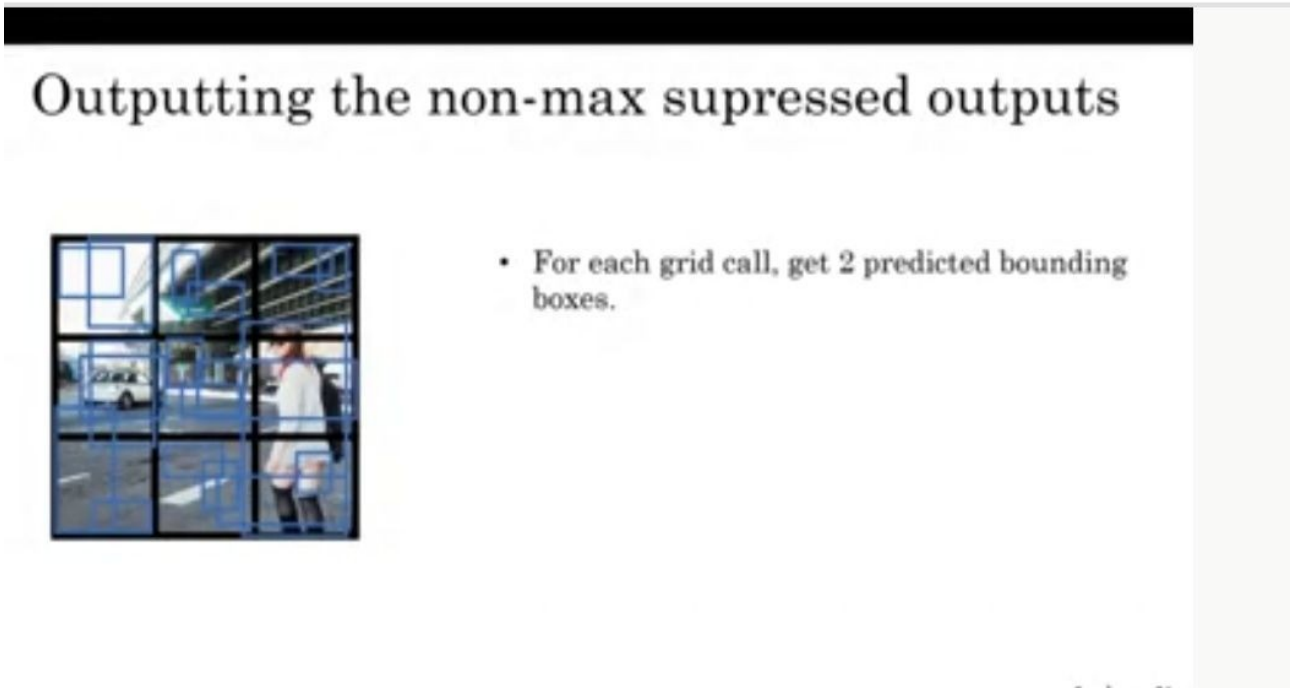
So we go through this and for each of our nine grid positions each of our three by three grid positions , we would come with a vector like this , a 16 dimensional vector and so that's why the final output volume is going to be 3 by 3 by 16. So that's training and we train ConvNet that inputs may be 100 by 100 by 3 and our convnet would then finally output a volume of 3 by 3 by 16 or 3 by 3 by 2 by 8. Now let's look on how our algorithms can make predictions.

Given an image , our neural network will output this 3 by 3 by 2 by 8 volume where for each of the nine grid cells we get a vector like that. So for the grid cell here on the upper left , if there's no object there , hopefully our neural network will output zero or some other values , our neural network can't output a question marks , can't output don't care , it will output some other numbers but these numbers will basically be ignored because the neural network is telling us that there's no object there. In contrast for the box that the neural network thinks that there is an object it'll output a set of numbers that corresponds to specifying a pretty accurate bounding box for the car. so that's is how the neural network make predictions.

← YOLO Algorithm



Finally we run through non-max suppression , Here's how we run non - max suppression , if we're using two anchor boxes , then for each of the grid cells we get two predicted bounding boxes , some of them will have a very probability , very low Pc but you still get two predicted bounding boxes for each of the nine grid cells.



So let's say those are the bounding boxes we get and notice that some of the bounding boxes can go outside the height and width of the grid cell that they came from. Next , we then get rid of the low probability predictions. so get rid of the ones that even the neural network says , probably there is no object there, so get rid of those and then finally if we have three classes we're trying to detect, we are trying to detect pedestrians , cars and motorcycles. so now what we do is we independently run non max suppression for the objects that were predicted to come from that class. so we use non - max suppression for the predictions of the pedestrian class , run non max suppression for the car class and non max suppression for the motorcycle class. and so the output is hopefully that we will have detected all the cars and all the pedestrians in this image so that's YOLO object detection algorithm.

YOLO Algorithm

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.

Regional Proposals

- R-CNN
- Fast R-CNN
- Faster R-CNN
-

Image Segmentation

- What is image segmentation
- What is the name semantic and segmentation refers to ?
- How is it differ from the object detection
- What are some of the advantages
- How Image segmentation works ?
- What is the disadvantage of normal CNN's to do image segmentation ?
- Why do i need up sampling ?
- What is Transpose
- Why do i need to use it
- What is the advantage of it
- How transpose works
- What is this U-Net Architecture?
- Where does the idea of U-Net originate ?
- Why do i need to use it ?
- What makes this architecture special for semantic segmentation?
- How U-Net architecture works
- How do we label semantic segmentation
- Is there are different types of Image Segmentation like semantic or Instance segmentation ?

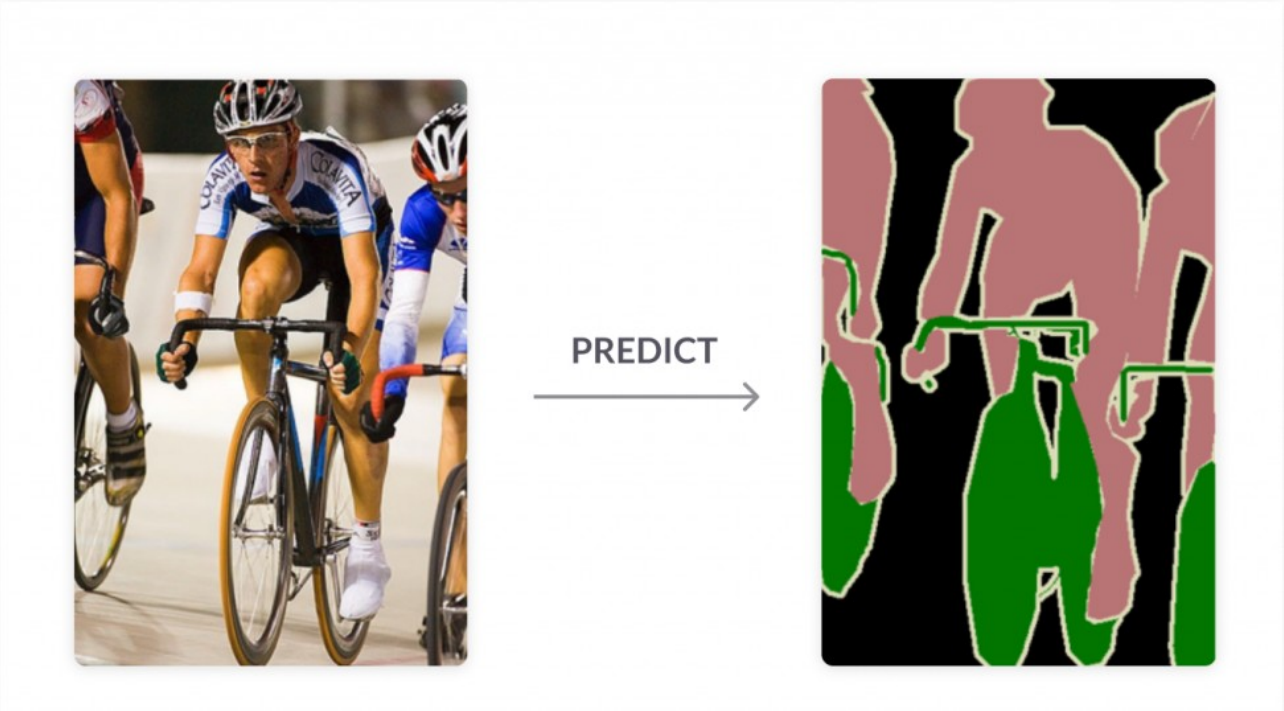
Image Segmentation , Semantic Segmentation and Instance Segmentation

What is Image Segmentation :- Image segmentation is a critical process in computer vision , it involves dividing a visual input in to segments to simplify image analysis , image segmentation is all about identifying parts of the image and understanding what object they belong to . It's all about segmenting out a certain object from our input image. The goal of image segmentation is to grouping similar regions or segments of an image under their respective class labels.

Types of Image Segmentation :- Image segmentation tasks can be classified in to three groups based on the amount and type of information they convey

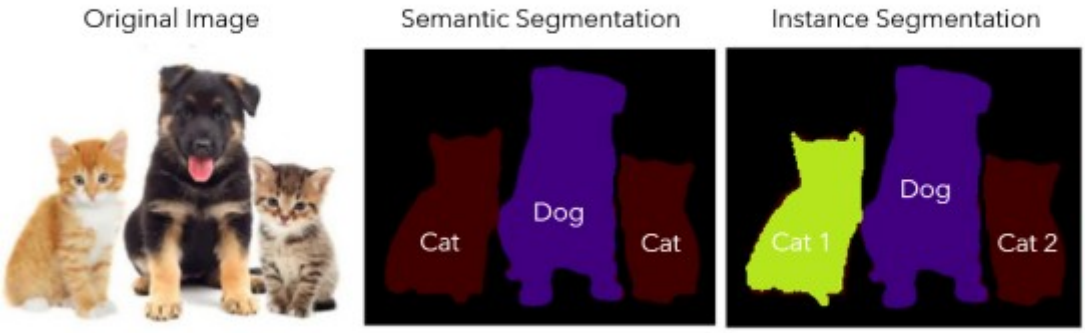
- Semantic Segmentation
- Instance Segmentation
- Panoptic Segmentation

Semantic Segmentation :- classifies all the pixels of an image in to meaningful classes of objects. These classes are “semantically interpretable” and correspond to real world categories. For instance we could isolate all the pixels associated with a cat and color them green. This is also known as dense prediction because it predicts the meaning of each pixel.



Instance Segmentation :- identifies each instance of each object in an image. It differs from semantic segmentation in that it doesn’t categorize every pixel. If there are three cars in an image , semantic segmentation classifies all the cars as one instance , while instance segmentation identifies individual car. semantic segmentation treats multiple object of the same class as a single entity. while instance segmentation produces a segment map of each category as well as each instance of a particular class therefore providing a more meaningful inference on an image.

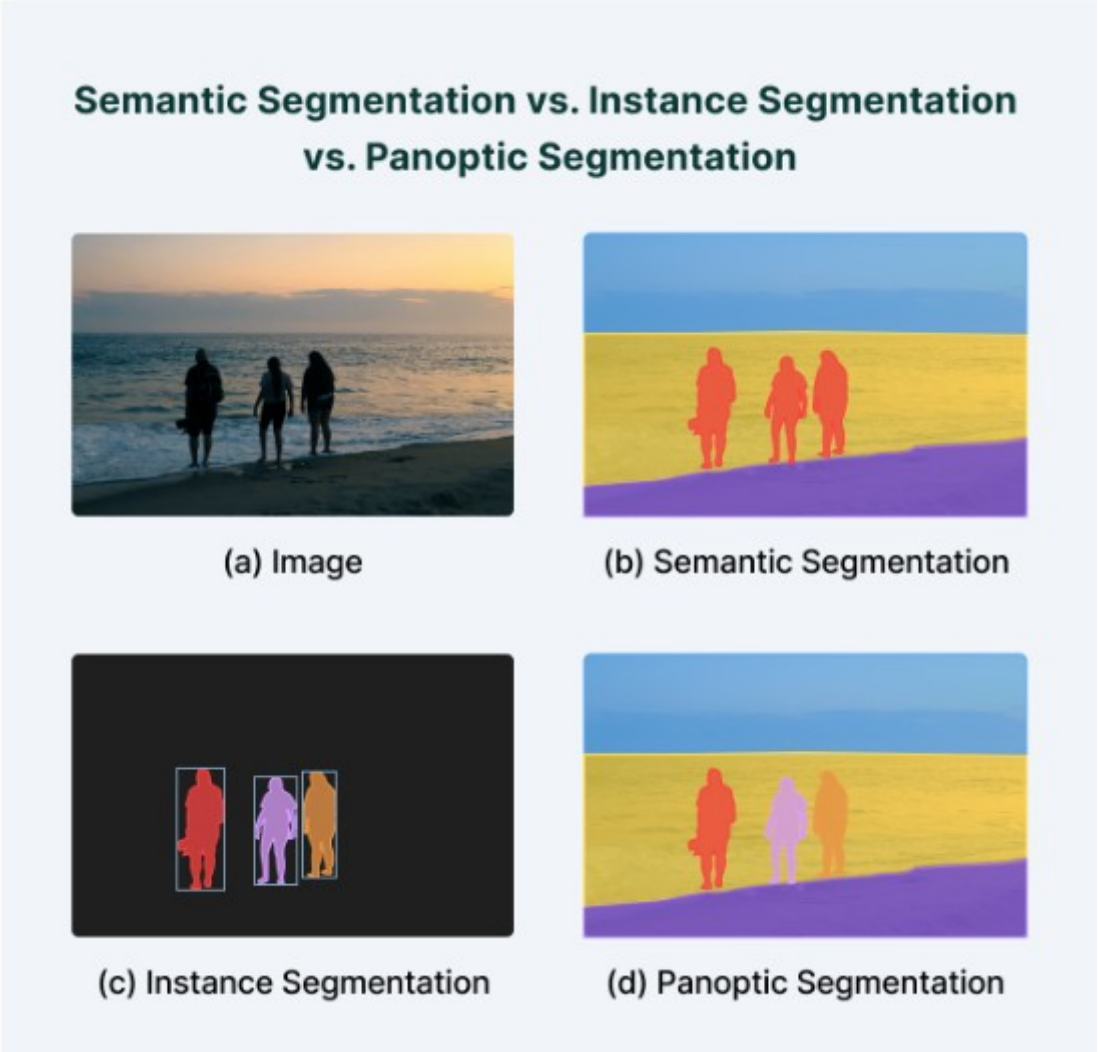
Let's consider an image below with cats and dogs.



Semantic segmentation can mark out the dog and cats pixels however there is no indication of how many dogs and cats are there in the image. But with instance segmentation one can find the bounding boxes of each instance (which in these case pertains to a dog and two cats) as well as the object segmentation maps for each instance , there by knowing the number of instances (cats and a dog) in the image.

Instance segmentation models classify pixels in to categories on the basis of "instances" rather than classes. An instance segmentation algorithm has no idea of the class a classified region belongs to but can segregate overlapping or very similar object regions on the basis of their boundaries. If the same image of a crowd we talked about before is fed to an instance segmentation model , the model would be able to segregate each person from the crows as well as the surrounding object (ideally) but would not be able to predict what each region/object is an instance of.

Panoptic Segmentation :- it is the most recently developed segmentation task , can be expressed as a combination of semantic segmentation and instance segmentation where each instance of an object in the image is segregated and the object's identity is predicted.



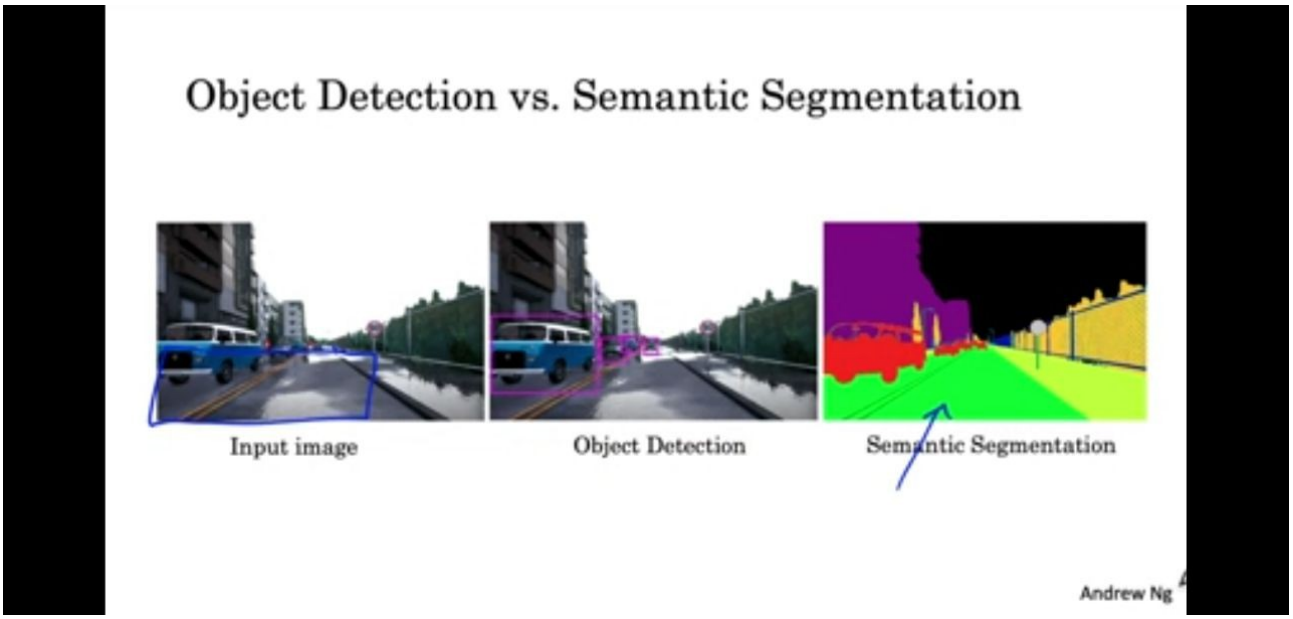
What is semantic segmentation ?

The goal of semantic segmentation is to draw a careful outline around the object that is detected so we know exactly which pixels belong to the object and which pixels don't.

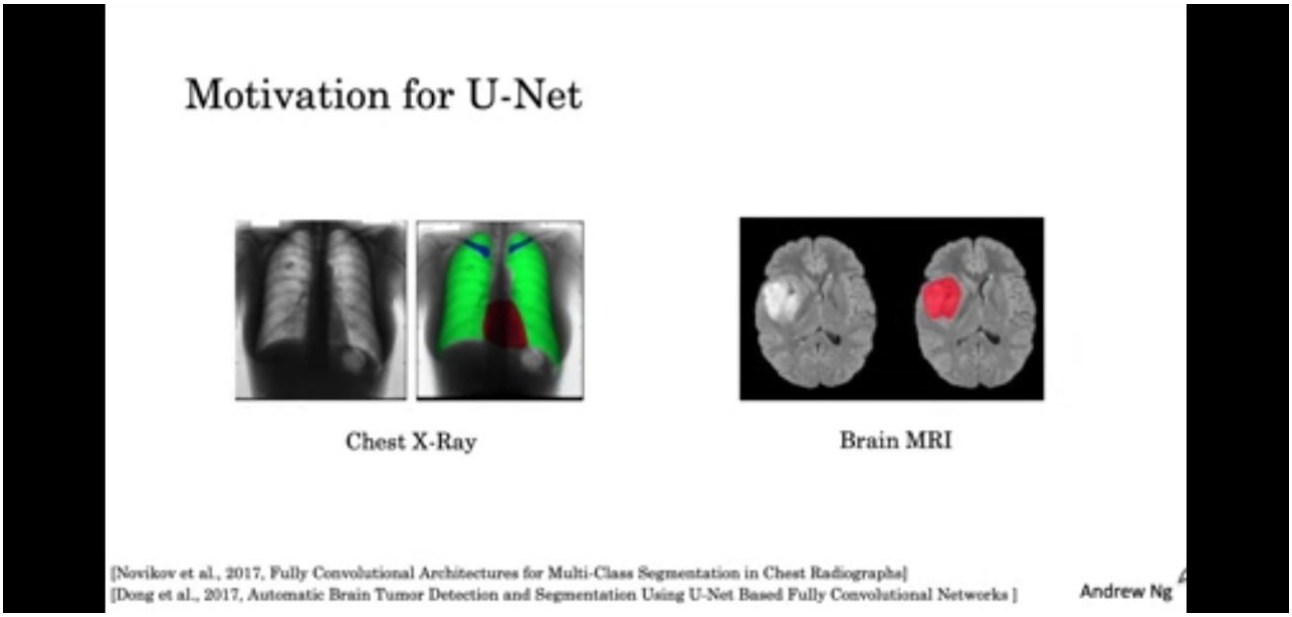
object detection vs semantic segmentation

Let's say we are building a self driving car and this an input image like this down below , and we like detect the position of the other cars , if we use an object detection algorithm , the goal will may be to draw a bounding boxes around the vehicles. This might be good enough for self driving cars (because semantic segmentation might not be that much advantageous for self driving cars) but if we want our learning algorithm to figure out what is every single

pixel in this image , then we may use a semantic segmentation algorithm. Which is rather than detecting the road and trying to draw a bounding box around the road , but with semantic segmentation the algorithm attempts to label every single pixel as “is this drivable road or not ”. semantic segmentation treats multiple object of the same class as a single entity.



What are the application semantic segmentation ?



In medical imaging given a chest x-ray we may want to diagnose some one has a certain condition but what even more helpful to doctors is if we can

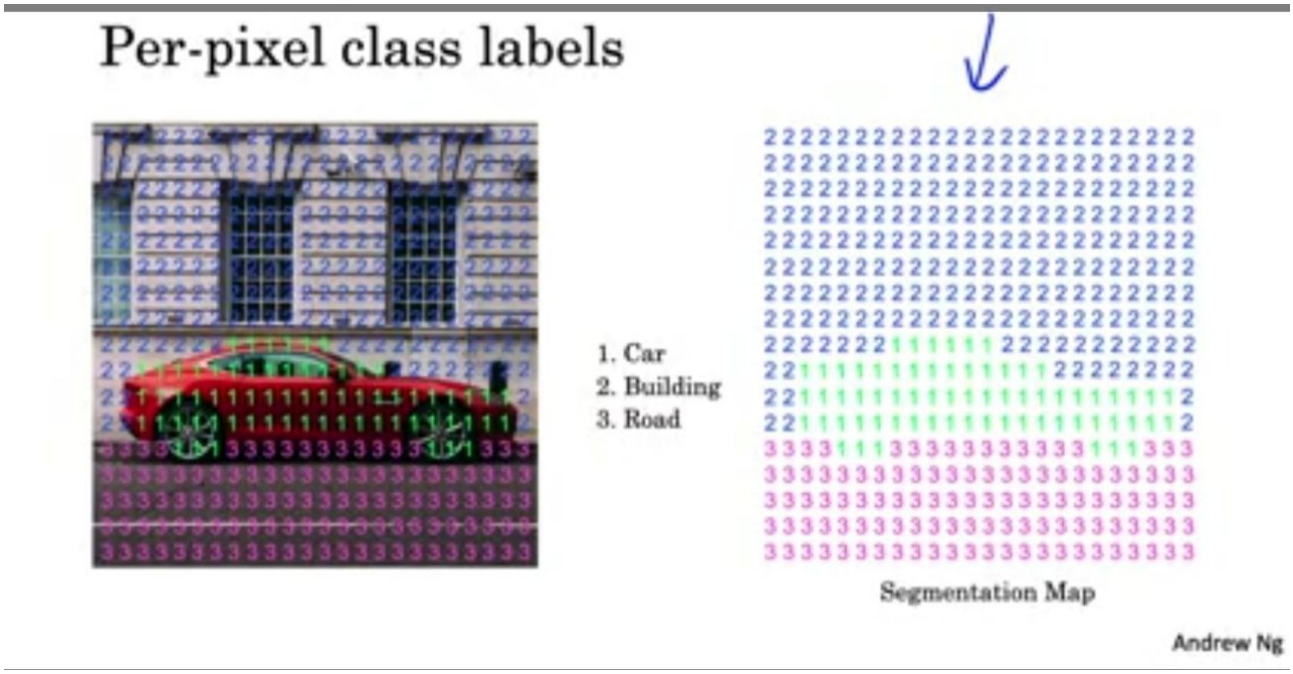
segment out in the image exactly which pixels corresponds to certain parts of the patients anatomy. In the image on the left the lungs , the hearts are segmented out with different colors , this segmentation would make it easier to spot irregularities and diagnose disease and also help surgeons planning out surgerious. In the image on the left a brain's MRI scan is used for brain tumor detection , manually segmenting out tumor from MRI is very time consuming and lubories but the learning algorithm can segment out the tumor automatically, this saves a radiologist a lot of time and this is also a good input for surgical planning. The architecture used to generate this result is an algorithm called U-Net.

Let's dig in what semantic segmentation actually does , for simplicity let's use the example of segmenting out a car from some background , let's say for now the only thing we care about is segmenting out the car in this image. In this case we may decide to have two class labels , one for a car and zero for not car.

In this case the job the segmentation algorithm of the U-Net algorithm will be to output either 1 or 0 , for every single image on the pixel , where each pixel should be labeled 1 if it's part of a car and labeled 0 if it's not part of the car.

Alternatively if we want to segment this image a little bit more finely, we may decide that we want to label the car as 1 and may be we want to know

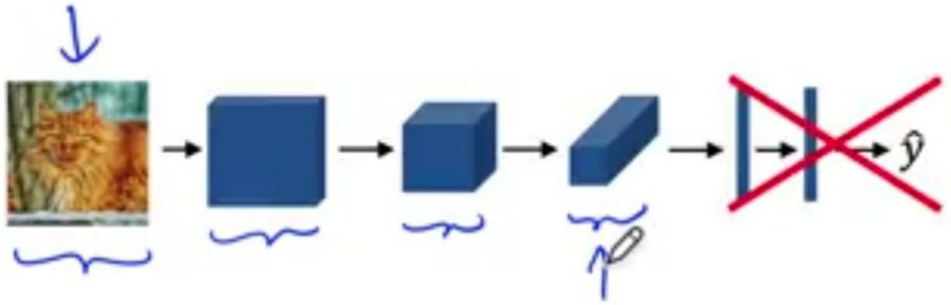
where the buildings are in which case we would have a second class , class 2 the building and then finally the ground or the road class three , in which case the job of the learning algorithm is to label every pixel as follows instead. and taking per pixel label and shifting it to the right , the pixels will be the output that we will like to train U-Net to be able to give.



Instead of just giving a class label or may be a class label and a coordinates needed to specify a bounding box , the neural network U-Net in this case has to generate a whole matrix of labels.

So what is the right architecture to do semantic segmentation ? So let's start with object recognition neural network architecture that we are familiar with and let's figure out on how to modify this in order to make this neural network output a whole matrix of class labels. Here is a familiar convolutional neural network architecture where we input an image which is fed forward in to multiple layer in order to generate multiple class labels y^ , in order to change this architecture to a semantic segmentation architecture , let's get rid of the last few layers and one key step of semantic segmentation is that where us the dimensions of the image have been generally getting smaller and smaller as we get left to right and now needs to get bigger so we can gradually blow back up to a full size image which is a size we want for the output.

Deep Learning for Semantic Segmentation



So specifically this is what a U-Net architecture looks like , as we go deeper in the U-Net the height and width will go back up where the number of channels will decrease , the U-Net architecture will looks like this until we eventually get the segmentation map of the cat. Now one operation we have not yet covered is what does we used to make the image bigger , to explain how that works we have to know how transpose convolution works.

TRANSPOSE CONVOLUTION

- Why is the name called transpose ?
- Why do i need transpose convolution ?
- Why do i need the up sampling for semantic segmentation ?
- How the transpose convolution works ?

Problem with simple RNN :- The convolution operation reduces the spatial dimension as we go deeper down the network and creates an abstract representation of the input image. This feature of CNN's is very useful for tasks like Image Classification where we just have to predict whether a particular object is present in the input image or not. But this feature might cause problems for tasks like object Localization , segmentation where the spatial dimension of the object in the original image are necessary to predict the output bounding box or segment the object.

Most of the layers like Convolutional layers and pooling layers down sample the height and width of the input or by using padding , keep them unchanged .In many processes like semantic segmentation , object detection which processes classification at a pixel level , it's convenient to not change the size of the output or keep the dimensions of the input and output the same.

To fix these problem various techniques are used such as fully convolutional neural networks where we preserve the input dimensions using "same padding ", this technique solves the problem to a great extent , it also increases the computation cost as now the convolution operation has to be applied to original input dimensions throughout the network.

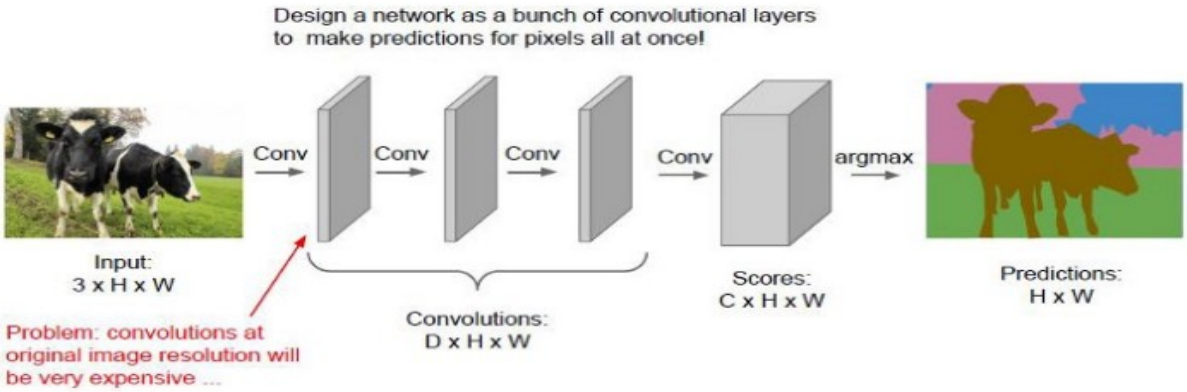


Figure 1. Fully Convolutional Neural Network(source)

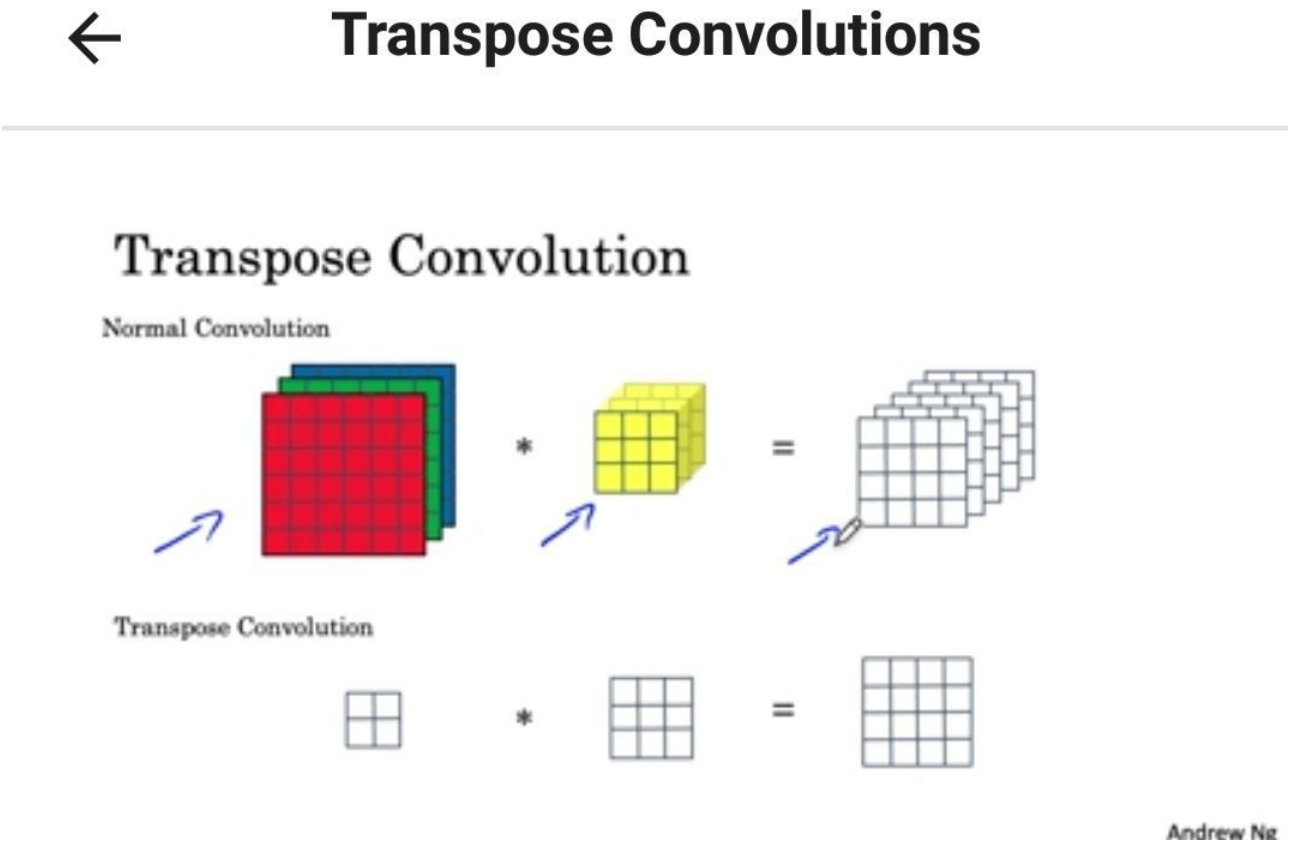
Another approach used for image segmentation is dividing the network in to two parts that is an down sampling network and then an up sampling network. In the down sampling network , simple CNN architectures are used and abstract representations of the input image are produced. In the up sampling network , the abstract image representations are up sampled using various techniques to make their spatial dimensions equal to the input image. This kind of architecture is famously known as the Encoder - Decoder network.

The Transpose convolution is a key part of the U-Net architecture. The reason it is called Transpose is because it's the transpose(T) the ordinary convolution.

How Transpose Convolution Works ?

We're familiar with the normal convolution in which a typical layer of a new network which input a six by six by three image, convolve that with a set of say three by three by three filters and if we have five of these then we end up with an output that is four by four by five.

A transpose convolution looks a bit different. We might input a two by two activation, convolve that with a three by three filter and end up with an output that is four by four , that is bigger than the original inputs. What is the formula for this?, i think since it is transpose,we will use (n+f-1) * (n+f-1)

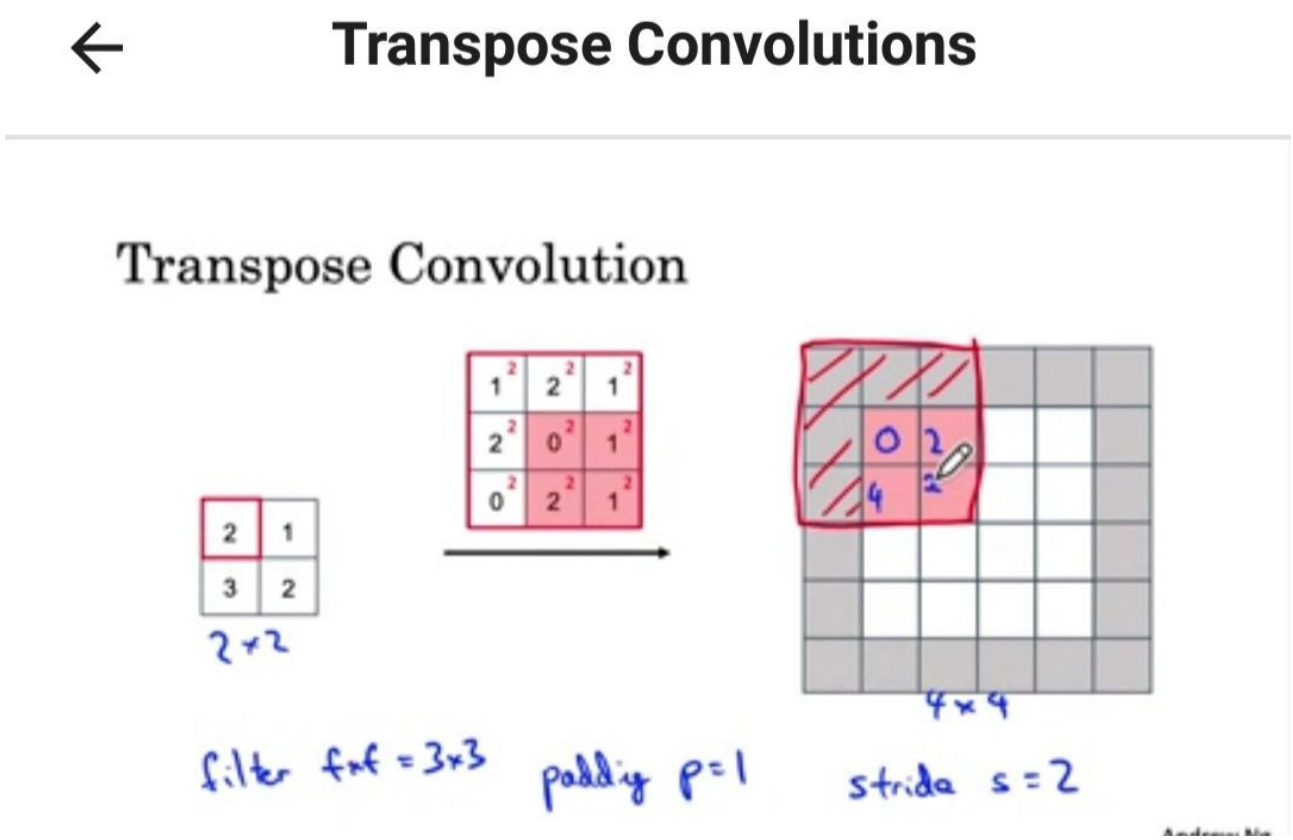


In this example , we're going to take a two by two inputs like they're shown on the left and we want to end up with a four by four outputs but to go from a two by two to four by four , we are going to choose a filter that is three by three.

We're going to use a padding p = 1 in the outputs and a stride s equals to 2, so now let's see how the transpose convolution will work.

In the regular convolution, we would take the filter and place it on top of the inputs and then multiply and sum up. In the transpose convolution , instead of placing the filter on the input we would instead place a filter on the output.

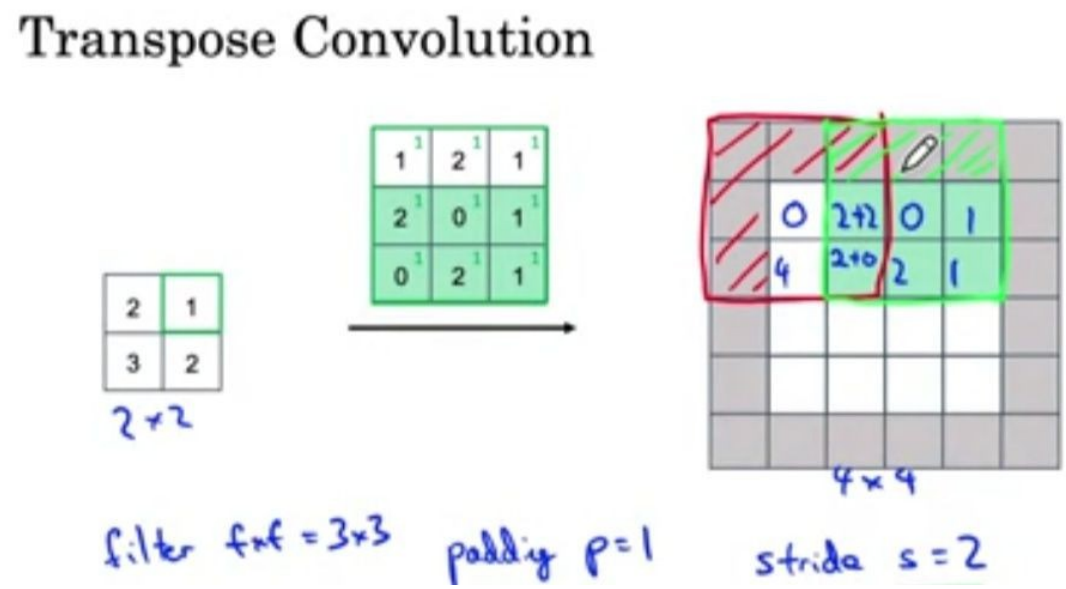
Let's start with the upper left entry of the input, which is two, we are going to take this number two and multiply it by every single value in the filter and we're going to take the output which is three by three and paste it in the output position , now the padding area isn't going to contain any values. what we are going to end up doing is ignore this padding region and just throw in four values in the red highlighted area and specifically , the upper left entry is zero times two , so that's zero. The second entry is one times two , that is two. down in the second row two times two , that's four and the next over here is one times two , that's equal to two.



Once again , we're going to take a one and multiply by one very single elements of the filter because we're using a stride of two , we're now going to shift to box in which we copy the numbers over by two steps. Again , we'll ignore the area which is in the padding and the only area that we need to copy the numbers over is this green shaded area. we may notice that there is some overlap between the places where we copy the red-colored version of the

filter and the green version and we can not simply copy the green value over the red one. where the red and the green boxes overlap , we add two values together. where there's already a two from the first weighted filter , we add to it this first value from the green region which is also two , we end up with two plus two. The next entry , zero time one is zero , then we have one , two plus zero time s one , so two plus zero followed by two , followed by one and again , we shifted two squares over from the red boxes here because it using a stride of two.

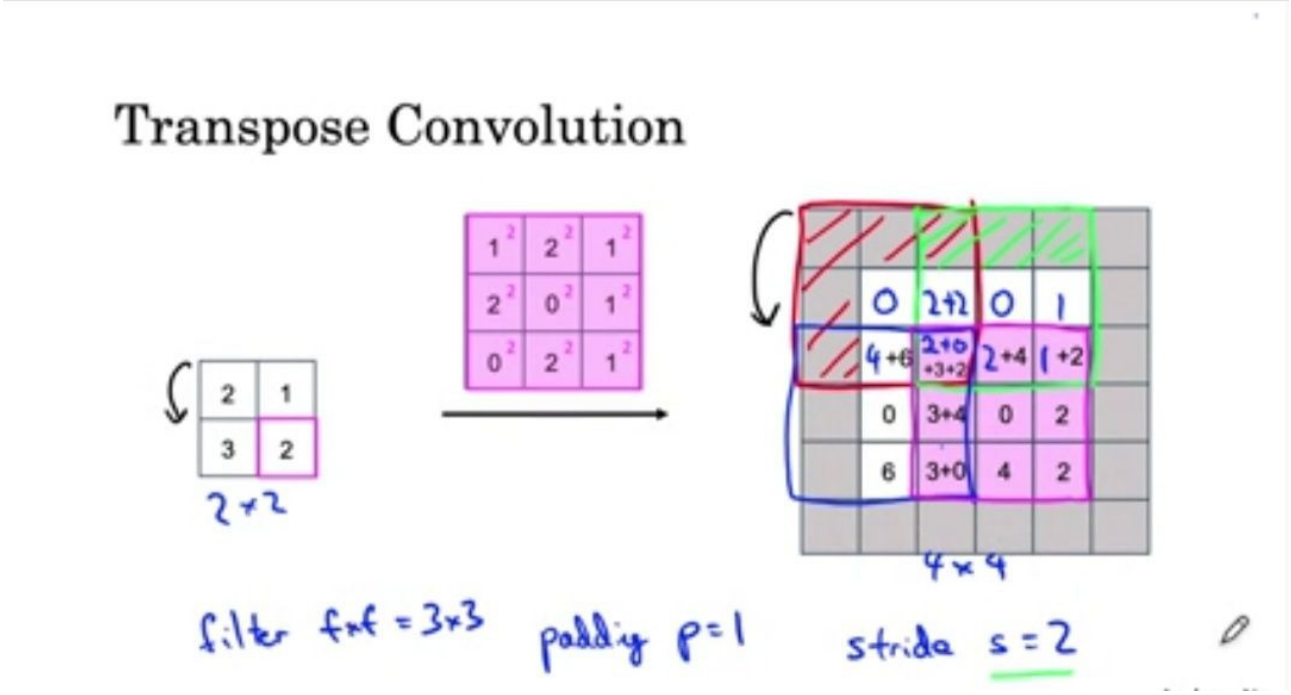
← Transpose Convolutions



Next let's look at the lower left entry of the input , which is three. we'll take the filter , multiply every element by three and we're going to go down by two steps here. we will be filling in our numbers in this three by three squares and we find that the numbers we copying over are two times three , which is zero and so on. now let's go in to the last input element , which is two. we will multiply every elements of the filter by two and add them to this block and we end up with adding one times two which is plus two and so one

for the rest of the elements , the final step is to take the numbers in these four by four matrix of values in the 16 values and add them up.

← Transpose Convolutions



U-Net Architecture Intuition

- ✔ - What is this U-Net Architecture?
- ✔ - Where does the idea of U-Net originate ?
- ✔ - Why do i need to use it ?
- ✔ - What makes this architecture special for semantic segmentation?
- ✔ - How U-Net architecture works