Course #4:- Convolutional Neural Networks

Week #1:-

Foundation of Convolutional Neural Networks:-

Computer Vision:- (CV)

- One of the most advanced areas of today, thanks to Deep Learning.
- Deep learning in CV has been very helpful for dicts and lots of applications like detecting pedestrains for autonomous driving.
- Vision based Navigation System.
- Face Detection for many applications.

Some of examples:- that we will be encountering in this topic are:-

- Image classification i.e:- whether it is a cat or not.
- Object detection: i.e. when in the picture a specific object/s is?
- Neural style transfer:-

  Content Image + Style Image

  Repainting of Content image in style of "Style Image".

→ One of the challenges of Computer vision problems is that, inputs can get really big

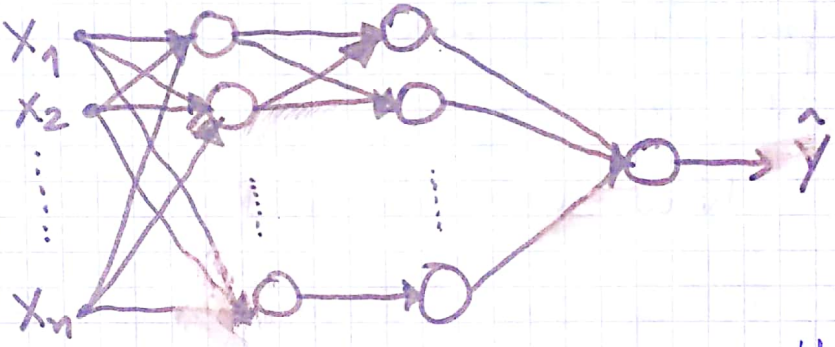
e.g previously we worked with images 64×64 and putting them in vector field 12288×1 feature vector. (Not so bad)

→ But suppose if you work with larger images such as 1000×1000 images

Now dimensions of input feature vector would be :- (1000 × 1000 × 3) × 1

So :- (3 million × 1) feature vector.

Now with Neural network.



Suppose 1st layer has 1000 hidden units then $W^{[1]} \in \mathbb{R}^{(1000, 3M)}$ which is a lot.

with that many parameters, it is difficult to get enough data to prevent your network from over-fitting.

Also, Computational power required will be massive.

→ Now for Computer Vision task you wouldn't want to get stuck with tiny images. For using large images, you implement a **Convolution Operation**. It is the fundamental block of CNNs. Let's do that :-

Edge Detection :-

Earlier we discussed that, how some of the earlier layers might detects features like edges than some later layers might detect cause of objects then further down the network they detect complete objects.

→ Let's see how you can detect edges in an image.

Suppose you have an image with people standing on the bridge against the railing. You pass that through vertical edge detector and horizontal edge detector and see the desired result. Now how do you do that?

Let's take an example of Greyscale image:

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

Let's say you have (6 x 6 x 1) image

e.g: to detect vertical edges you convolve (*) this image with a filter.

* Suppose filter is (3 x 3)

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

=

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

→ "If you are not sure how Convolution works look up online". Pretty basic Python computation

Output of this convolution will be 4x4 image.

→ In Python with tensorflow there is function for this:-

`tf.nn.conv2d`

other framework like **Keras**: Conv2D

Now, why is this doing vertical edge detection?

Let's look at another example:-

Let's use a simple image:-

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

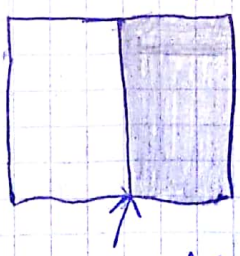
$*$

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

equals to

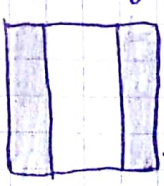
0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

So, in this example then main 6×6 image looks like this



this is a edge in the middle that shows transition of shade.

Kernel or filter



output Image

⇒ bright region in the middle. That corresponds to having detected vertical edge down in the middle of (6×6)

→ One intuition to take away from this example is that, a vertical edge is (3×3) region (since we are using a (3×3) filter where there are bright pixel on the left and dark pixels on the right. Don't care what in the middle.

Let's see how to use all this as one of the basic building blocks of Convolutional Neural Networks.

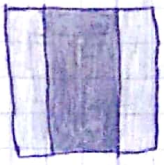
More on Edge Detection:-

→ Let's see the difference between positive and negative edges, that is the difference b/w light to dark and dark to light edge transition and you also will see other types of edge detectors and have an algorithm learn this operation.

→ Let's consider the last example that we saw: what if instead of having all 1s to the left we have them at the right. Keeping the Convolutional Kernel same, we are gonna have following output

0	-30	-30	0
0	-30	-30	0
0	-30	-30	0
0	-30	-30	0

Visualization



-30s show that, this is a dark to light rather than light to dark transition.

→ You can take the absolute of the matrix doesn't matter but this particular filter does make a difference between light to dark and dark to light edges.

Let's see some more examples:-

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

This (3×3) filter allow you do detect vertical edges but it shouldn't surprise you this this filter allow horizontal edge detection

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

→ As we discussed that, vertical edge is (3×3) pixel area where pixels are bright on the left and dark on the right. Same this:-

Horizontal Edge would be a 3×3 pixel area where upper part has brighter pixels and lower part has darker pixels.

Let's see a Complex example :-

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

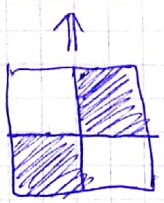
*

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

=

0	0	0	0
30	10	-10	-30
30	10	-10	-30
0	0	0	0

▼ -30 represents marked in ▼ on the original image which is brighter on bottom and darker on top. Negative Edge



* 30 corresponds to marked area on the original image where indeed there are bright pixels on the top and lighter on the bottom. Positive Edge

→ There are different set of edge detectors that you can use, e.g.: instead of using something that we used earlier we can use something like this.

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

Sobel filter.

→ It puts a little more weight to the center which makes it much more robust.

CV experts use other filter's settings too like

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

Scharr filter.

→ This is just for vertical edges, other settings also occur like 90°, 60° or horizontal edges.

→ It turns out that, may be you don't need to hand pick these numbers, may be you can just learn them.

→ Like treat these 9 numbers as parameters which you can learn using back propagation.

Goal is to learn the parameters (9 matrix elements) and when you convolve it with image, you get a good edge detector.

→ In this way your algorithm can learn some optimal setting which not only learns vertical and horizontal edges but other degree of edges too.

Padding :-

One modification you need for your convolution for training CNN is **Padding**.

Let's see how that works.

→ what we saw is that, when convolving (6x6) with (3x3) it gives (4x4) image.

That's because number of possible positions for your (3x3) filter is (4x4).

Now in mathematical terms it can be written as that, if you convolve (n x n) image with (f x f) kernel then you wind up with

$$(n-f+1) \times (n-f+1)$$ Convolved image.

→ So one, downside to this is that, your image shrinks on every step.

→ Secondly, pixels in the corners are only used up for one time, whereas pixels in the middle are used in lot of computations of resultant image.

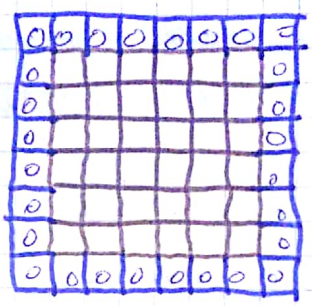
→ As pixels in the corner are used in less computations So, you are throwing away a lot of informations in computing resultant image.

→ Now in order to solve both of these problems, we use padding of additional pixels around the border of the images.

→ Now instead of having $(n \times n)$ image, you'll have $(n+2) \times (n+2)$ image and you'll get back your original size.

→ By Convention you pad image with zeros.

If P = padding = 1.



then output is :-

$$(n+2p-f+1) - (n+2p-f+1).$$

→ You can also pad it with 2 pixels around the edges. Now p will be 2

In terms of how much to pad, there are two choices. **Valid & Same Convolution.**

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

No padding

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

$$(n \times n) * (f \times f) \rightarrow \underbrace{(n+2p-f+1)}_{n} \times \underbrace{(n+2p-f+1)}_{n}$$

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

→ So when f is odd, you can choose the padding size as follows to get the output image as the same size.

So when $f=3$ $\quad p = \frac{3-1}{2} = 1$

and when $f=5$ $\quad p = \frac{5-1}{2} \Rightarrow \boxed{p=2}$ for output size to be same as i/p's

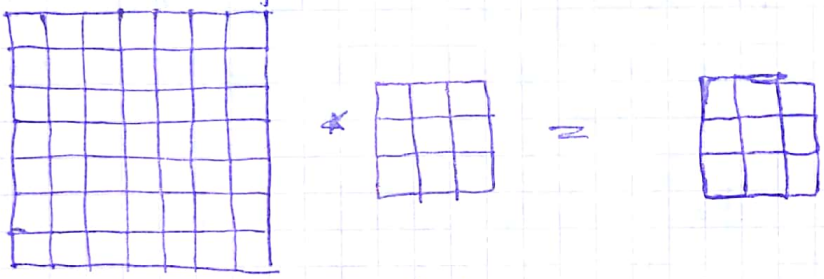
→ By Convention in CV, f is usually odd.

Strided Convolutions:-

- Another piece of basic building blocks when training CNNs

Example:-

- Suppose you have (7×7) image to be convolved with (3×3) filter.



Convolve with Stride = 2 ⇒ $S=2$

what that means? Padding = $p=0$

- Place kernel onto image do usual computation and upon stepping through the columns step 2 (stride = 2) instead of 1. Do the same computations then. When stepping through rows, follow the same stepping scheme and do computations.

In this example:-

$(7\times7) * (3\times3) \rightarrow (3\times3)$ output. → floor the o/p

$(n\times n) * (f\times f)$ these dimensions are governed by the following formula:-

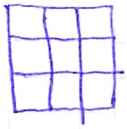
$$(n\times n) * (f\times f) \Rightarrow \left[\frac{n+2p-f}{s}+1\right] \times \left[\frac{n+2p-f}{s}+1\right]$$

$(S=2)$

→ Now the convention here is that, your kernel must lie exactly on the image + padding (if you have any). If one column or two of the kernel hangs outside the image you don't do that part of Computation.

→ Technical note on cross-Correlation Vs. Convolution :-

Convolution in math textbooks :-

if we have (n×n) image and it is to be convolved with  (3×3) kernel, then it is defined as that, this (3×3) kernel is flipped horizontally and vertically then it is convolved like the way we have done so far.

→ The process we have been doing so far is Cross-Correlation. But in DL literatures it is called as Convolution. We don't both with flipping operations.

Using this flipping operations, Convolution enjoys associativity :

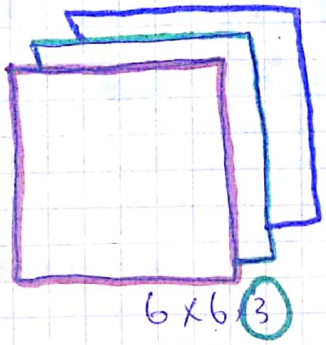
$$(A * B) * C = A * (B * C)$$

But it doesn't really matter in Deep learning.

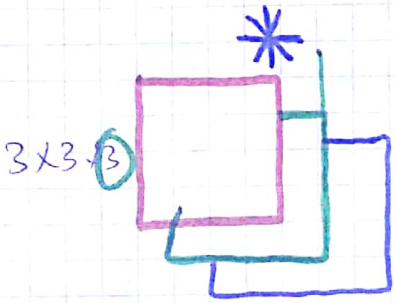
Convolution over Volume:-

Let's see how you implement Convolution on 3d-volumes.

→ Let's say instead of grayscale image, you want to implement Convolution on RGB image.

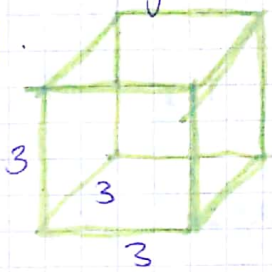


Now in order to detect features, you convolve it with not only 3x3 filter as you have done previously but with (3x3x3) 3d filter.



Not necessarily an image (RGB).

It can also be represented as



- Place this cube on the RGB image:-
  3d kernel has 3 layers so it will fit on to layers, do the usual computations for convolution, and compute the output.

Then slide through col. and rows and do computations.

What does this allow you to do?

You set parameters of kernel like

R
$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

G
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

B
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Detects edges only in red channel.

OR you can set the same settings in all 3 layers to detect edges in (RGB) channels.

In Convention, the number of layers of image and kernel should be the same.

In this case output would be 2d (4×4).

Crucial Idea:-

What if you don't want to detect just vertical edges, what if you want to detect vertical, horizontal and other orientations' edges too?

In other words, what if we want to use multiple filters.

Suppose you get (4×4) image from vertical edge detector and (4×4) image from horizontal edge detector. Now you have two images, what you do is stack them together and you'll have $(4 \times 4 \times 2)$ output layers.

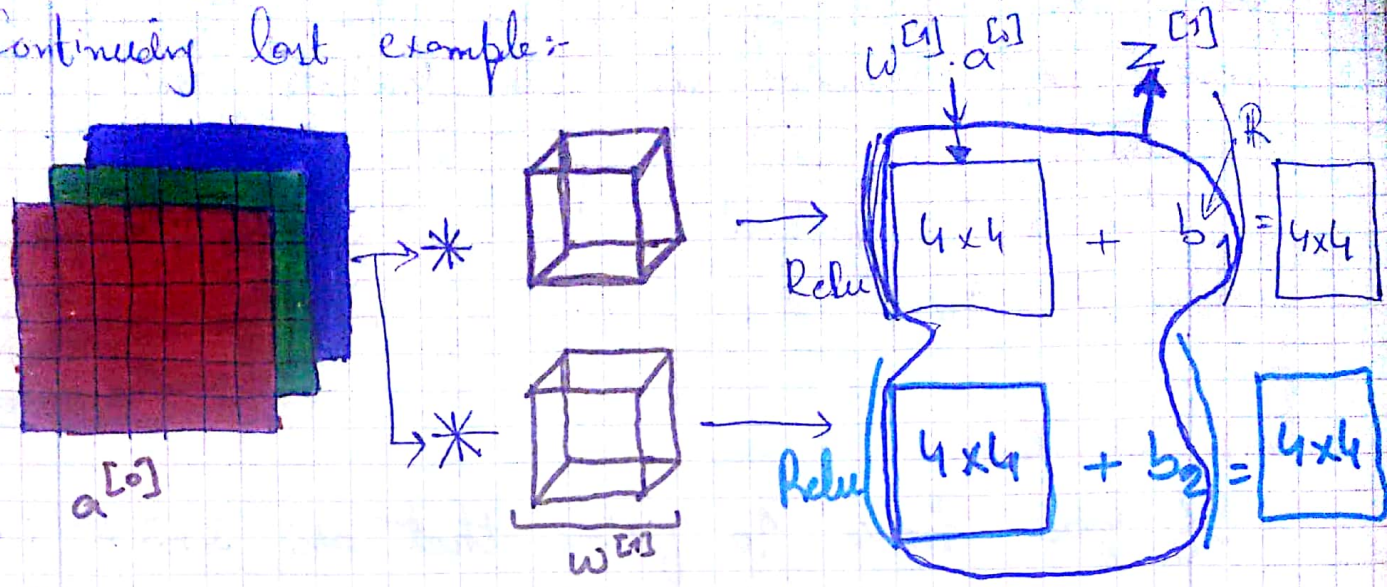
Summary:- $(n \times n \times n_c) * (f \times f \times n_c)$

$\Downarrow$

$(n-f+1) \times (n-f+1) \times n_c'$

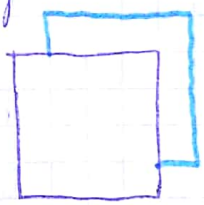
One Layer of Convolutional Network:-

Continuing last example:-



final thing to turn this into convolutional neural network layer is that, for each of output we are going to add a bias. Then apply non-linearity like Relu.

Now stack them again like we did before.



$\in \mathbb{R}^{(4\times4\times2)}$

- This is one layer of CNN.

For mapping it back to standard neural network's forward propagation step, remember we did something like this:

$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$$

$$a^{[1]} = g(z^{[1]})$$

So the RGB image is $a^{[0]}$

- filters play the role of W's.

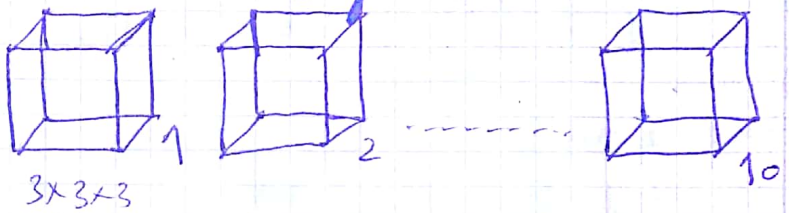
So $\left[(6\times6\times3) * (3\times3\times3)\right] + b_1 \rightarrow$ Relu $\Rightarrow$ one layer of CNN

Now, In this example we assumed that, we have two filter, what if we had 10 filters. Then instead of wounding up with (4×4×2) we would've wounded up with (4×4×10) dimensional output volume Because then there'll be 10 maps of 'Relu(wx+b)'

Exercise:-

If you have 10 filters that are 3×3×3 in one layer of neural network, how many parameters does that layer have?

Each filter



3×3×3

27 parameters + 1 bias.

So 27×10 + 10 = 280 parameters.

Notice nice thing about this, despite the size of Image being as high as 1000×1000 or 5000×5000, no. of parameters remain fixed. and in these parameters we may be are computing horizontal, vertical or other dimensions' edges.

This is one nice property of CNNs that make them less prone to over-fitting.

Summary of notation :-

- If layer 'l' is a Convolutional layer :

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

$p^{[l]}$ = Padding. ⇒ It can also be specified by saying whether you're used "valid" convolution (no padding) or "Same Convolution" means appropriate padding.

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

$s^{[l]}$ = Stride

- **Input** $^{[l]}$: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ ∵ [l-1] denotes that it is activation from previous layer.

($n_H^{[l-1]}$ ↙ height of image; $n_W^{[l-1]}$ ↓ Width of image)

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

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

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

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

for vectorized implementation : $\boxed{A^{[l]} = (m) \times (n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]})}$

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

($n_c^{[l]}$ ⇑ no. of filters in layer 'l')

bias: $n_c^{[l]} \Rightarrow (1, 1, 1, n_c^{[l]})$

($n_c^{[l]}$ ⇑ vector of this dimension)

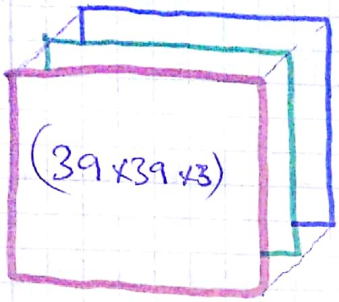
-

That's just one layer, let's see how can we stack them up and form a deep network.

Single Convolution Network Example:-

Concrete example of Deep CNNs :-

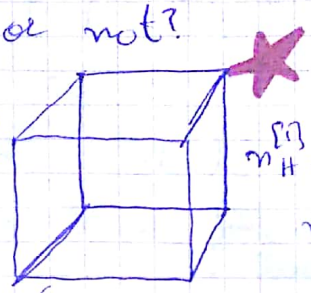
Let's say we are dealing with classification problem i.e. is it a Cat or not?



$n_H^{[0]} = n_W^{[0]} = 39$

$f^{[1]} = 3$
$s^{[1]} = 1$, 10 filters.
$p^{[1]} = 0$

Relu $(W x + b)$ → Activation →



$n_H^{[1]} = n_W^{[1]} = 37$
$n_C^{[1]} = 10$

(37 x 37 x 10)

Let build a ConvNet you could use for this

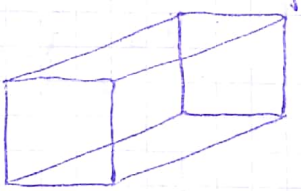
First layer uses a 3x3 filter to detect features

$$\frac{n+2p-f}{s} + 1 = \left(\frac{39-0+3}{1}\right) + 1$$

Now let's say you have another layer and this time you use 5x5 filter

$f^{[2]} = 5$
$s^{[2]} = 2$, 20 filters
$p^{[2]} = 0$

Relu $(W x + b)$

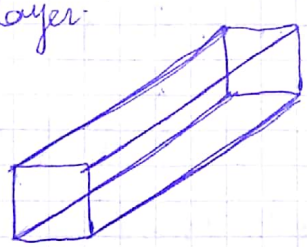


17 x 17 x 20

$$n_H^{[2]} = n_W^{[2]} = \frac{n+2p-f}{s} + 1 = \frac{37+0-5}{2} + 1 = 17$$

Another layer

$f^{[3]} = 5$
$s^{[3]} = 2$
$p = 0$

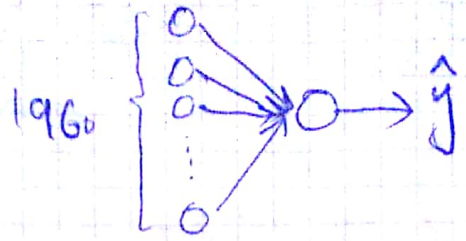


7 x 7 x 40

What you have done is you have taken (39 x 39 x 3) images and computed 7 x 7 x 40 features.

1960

what we can do is take this volume of 1960 and flatten it in a vector



This is a pretty typical example of ConvNet.

A lot of work in designing ConvNets involve tuning these hyperparameters i.e. what the filter size, what's the stride, padding, no. of filters.

→ Size of images remain same for some layers but it decreases as you go deeper into the network. As shown in the above example

→ whereas no. of channels generally increase.

Types of layer in a ConvNet :-

In a typical ConvNet, there are 3 types of layers :-

- Convolutional layer → one we have been using so far.
- Pooling (Pool)
- Fully Connected (FC)

Although it is possible to design pretty efficient ConvNet with just Convolutional layer, but most N.N architectures will also have a few pooling layers and few FC layers.

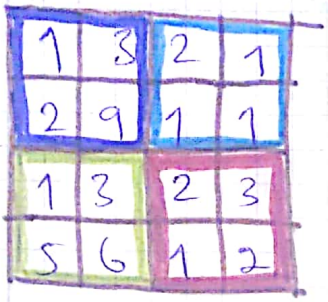
Pooling Layers:-

- Used to speedup the speed of computations.
- Makes detection of features more robust.

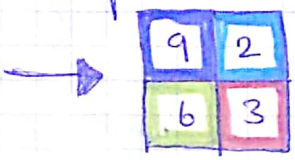
Let's analyze an example.

Pooling layer: Max pooling

Suppose we have a 4x4 input and you want to apply type of pooling called "Max Pooling"



It's output will be a 2x2 output.



way you do that is quite simple:

→ Take your input and divide it into different regions 4 different regions are shaded.

Then output will be corresponding shaded region's Max

→ For computing (2x2) output, we took the max of (2x2) region.

→ This as if you are applying a filter of size 2 and you are taking a stride of 2.

Filter = f = 2
Stride = S = 2
} hyperparameter.

Intuition behind max pooling:

→ If you think of this 4x4 region as some set of features, the activations in some layer of neural network, then a **large number** means that, it's a may be detected may be detected a particular feature. like upper left corner's feature has a particular feature may be an eye or whisker, whereas this feature doesn't really exist in the upper right hand corner

→ So, what the max operation does is that, so long as the features are detected anywhere and one of the quadrants in the (4x4) figure. It then remains preserved in the output of max pooling.

(Not sure if this is the underlying reason behind max pooling)

→ **Interesting property:** It has a set of hyper parameters but no parameters to learn.

Let's go over a different example:-

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

5x5

$f = 3$

$s = 1$

→

9	9	5
9	9	5
8	6	9

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

also work to computing the size of output.

→ What if you have a 3d input, then the output will have same dimension e.g $(5 \times 5 \times 2) \rightarrow (3 \times 3 \times 2)$

Perform max pooling on each channel independently.

There is one other type of pooling which isn't used very often.

Average Pooling: Here instead of taking the Max, you take the average of every filter.

So according to our last example, average pooling output will be.

with $f=2$, $S=2$

Exception:

→ Some times very deep in the neural network, you might use average pooling to collapse your representation from say $7\times7\times1000 \longrightarrow 1\times1\times1000$

But max pooling is used much more.

Summary of Pooling:

Hyperparameters:

- f: filter size.
- S: Stride

$f=2$ ⇓ used setting quite often. ← $S=2$ ⇓ OR $f=3, S=2$.

⇒ has effect of shrinking the height and width by a factor of above 2.

- Max Pooling or average pooling
- P: Padding ⇒ Rarely used.

Input:- $n_H \times n_W \times n_C$

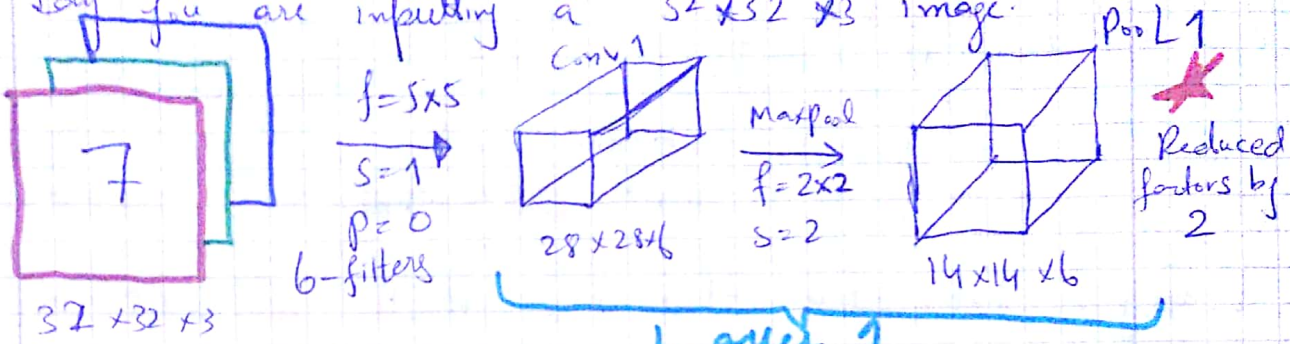
Output:- $\left\lfloor \frac{n_H - f}{S} + 1 \right\rfloor \times \left\lfloor \frac{n_W - f}{S} + 1 \right\rfloor$ assuming no padding.

Convolutional Neural Network Example:-

You know now pretty much all the building blocks for full CNNs.

Neural Network Example: (LeNet-5)

Let's say you are inputting a 32×32×3 image.



May be you are trying to do hand written recognition.

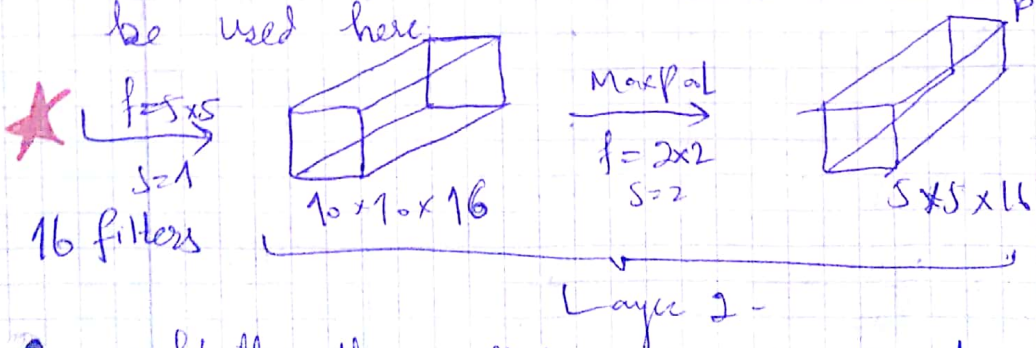
★ In literature there are 2 conventions which are slightly inconsistent what you call a layer.

parameters choices inspired by LeNet-5

→ One Convention:- Conv 1 + Pool 1 = Layer 1

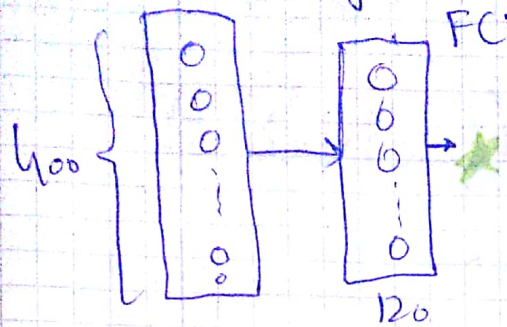
→ 2nd Convention:- Conv 1 is a layer and Pool 1 is a layer.

Usually layers are those which use weights and bias for computation so "One Convention" will be used here.



- flatten the output 5×5×16 ⇒ 400×1.

Take this 400 units and build next layer as having 120 units.



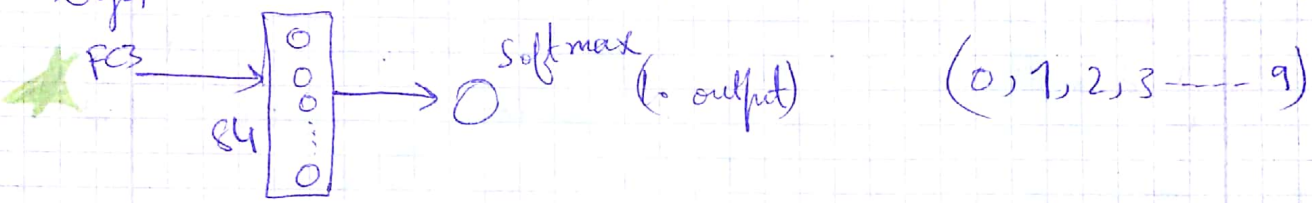
→ This is just like the single neural network you saw in Course 1 and Course 2

$w^{[3]} = (120, 400)$

$b^{[3]} = 120$

This is called fully connected because each units is interconnected with each other.

Lastly, let's take these 120 units and add another layer



One guideline is not to invent your own hyperparameter but look through literature and see what others have used. There's a chance that may work well for you.

1 or more ConV → PooL → 1 or more ConV → PooL → FC → FC → FC → Softmax

	Activation shape	Activation Size	# parameters
Input	(32, 32, 3)	3072 $a^{[0]}$	0
ConV1 (f=5, s=1)	(28, 28, 8)	6272	208
PooL 1	14, 14, 8	1568	0
ConV2 (f=5, s=1)	10, 10, 16	1600	416
PooL2	(5, 5, 16)	400	0
FC3	(120, 1)	120	48,001
FC4	(84, 1)	84	10,081
Softmax	10, 1	10	841

- Notice max pooling layers don't have any parameters.
- Conv layers tend to have few parameters.
- Activation size tend to decrease down gradually.

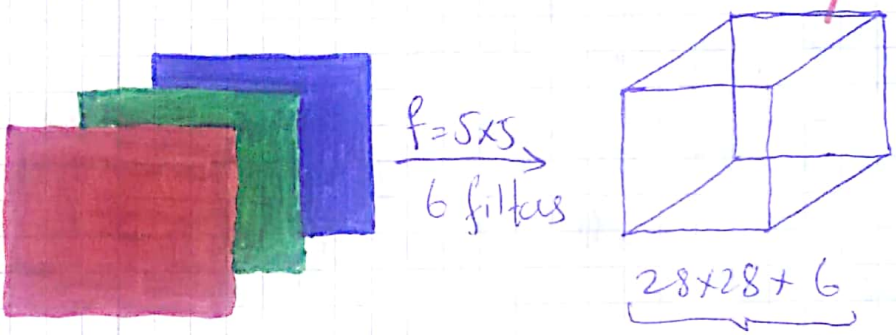
Why Convolutions?

- Why these are useful and how can we put them all togetha and train a CNN?

Two main advantages :-

- Parameters Sharing.
- Sparsity of Connections.

Let's illustrate with an example :-



Now if we were to build fully interconnected N.N with these kind of parameters then on the first layer it would be like this that

$$W^{[1]} \in \mathbb{R}^{3072, 4704} \Rightarrow 14 \text{ million floating point storage space required.}$$

→ So one thing more considering this small image these are lot of parameters to train.

→ Now in this examples parameters are quite small like $5 \times 5 = 25$ for each filter and 1 bias term so $26 \times 6 \Rightarrow 156$ parameters → Quite small.

Why parameters are so small:- Following are the reasons :-

Parameter Shaving:- A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.

This is true not only with low part of the image, like detecting low level features, but also high part detection like detecting eye etc.

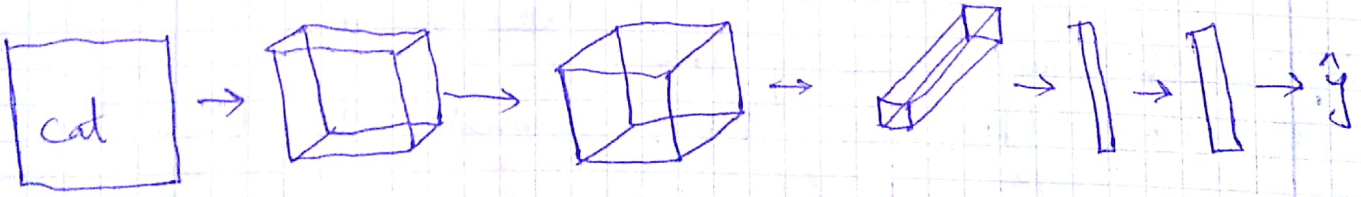
→ You don't need to learn parameter (hyper) separately for different parts of the images.

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

often you hear that CNNs are pretty good in catching translation in Variance e.g. pic. of cat shifted couple pixels to right is still pretty much a cat.

Putting it together:-

$(x^1, y^1), \ldots (x^m, y^m)$



Conv layer and FC have various parameters w and b. It lets you to define a cost function

$$\text{Cost } J = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

Use gradient descent to optimize parameter to reduce Cost J.