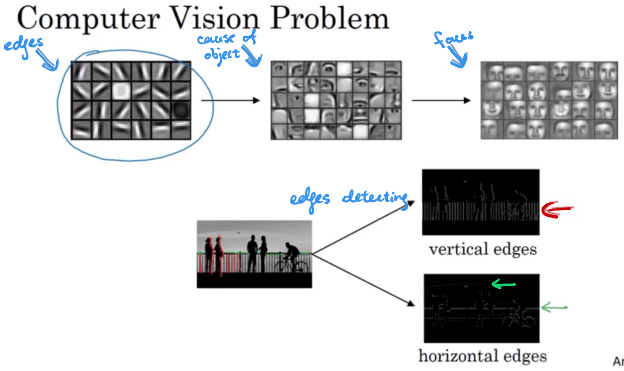
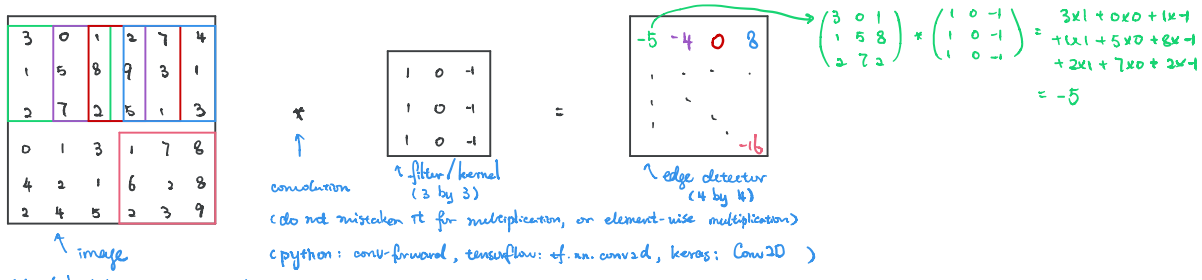
Convolutional Operations

The convolution operation is one of the fundamental building blocks of a convolutional neural network. To see how convolutional operation works, let's use an example of edge detection.

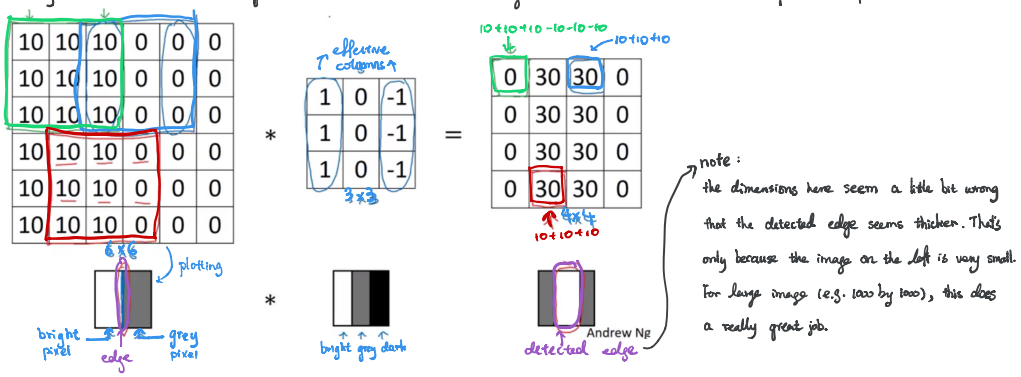
We have talked about how early layers of a NN might edges, then couse of objects later, and finally complete objects.



Let's see how we can detect vertical edges in an image by using a simple example of a 6 by 6 grayscale image.

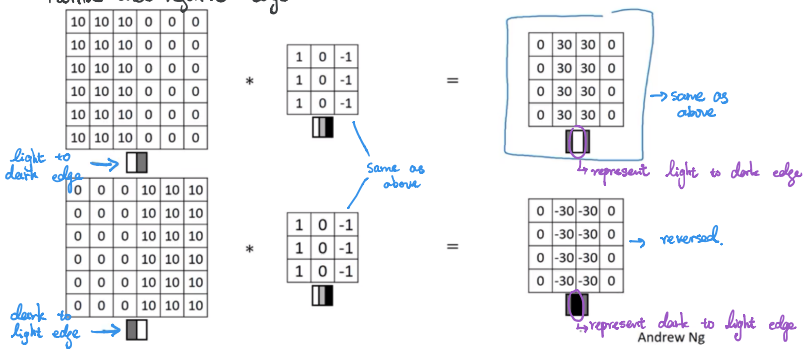


So why does the matrix on the right turn out to be vertical edge detector? Let's use an even simpler example:



There are more edges detectors that the above one, including positive and negative edges (i.e. difference between light to dark versus dark to light edge transitions), as well as algorithm-learned edge detector instead of the hand-coded one like we did.

- Positive and negative edge:



- Horizontal edge:

vertical

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

* flip it 90° →

horizontal

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

* note: appliable to other filter as well, i.e. flip the vertical filter 90 degree to get the horizontal filter.

- Other hand-picked filter

Sobel filter

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

→ more weight to the central pixel

Schaar filter

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

- Algorithm - learned filter

treating these 9 numbers as parameters, and let the algorithm learn them by back-propagation:

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

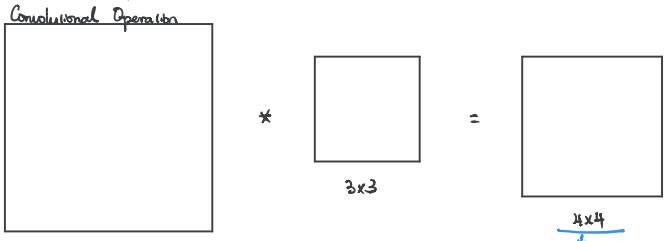
→ other than detecting vertical/horizontal edges, it's also able to edges at 45°, 70° etc.

But before we go about using back propagation to learn these 9 numbers, let's first talk about some other details that are important in building CNNs, including padding and different strides for convolutions.

• Padding

In order to build deep CNN, one modification to the basic convolutional operation is padding.

Basic Convolutional Operation

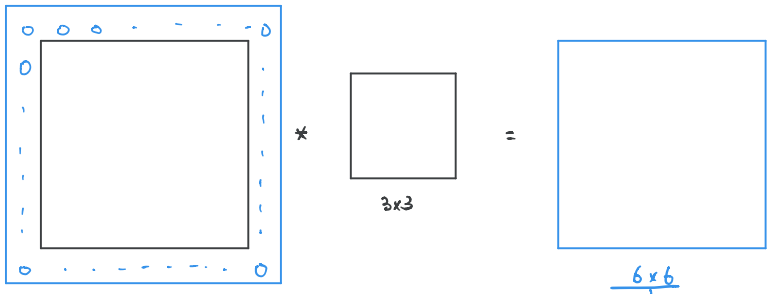


→ the original image has shrinked from 6x6 to 4x4 and after a few convolutions it might shrink to 1 by 1

→ pixels on the corners are used much less in the output, i.e. throwing away a lot of information on the edge of the image

the size of the output is determined by: $n-f+1 \times n-f+1$ (n: original size, f: filter size)

Added 1 pad → padding



padding with 1 broader ($p=1$)

→ preserves the size of original image.

→ not throwing away pixels on the corners

$n+2p-f+1 = 6$

How much to pad ⇒ 2 common choices:

- Valid padding : no padding

  $n \times n$ image $*$ $f \times f$ filter $= n-f+1 \times n-f+1$ output

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

  $n+2p \times n+2p$ image $*$ $f \times f$ filter $= n \times n$ image

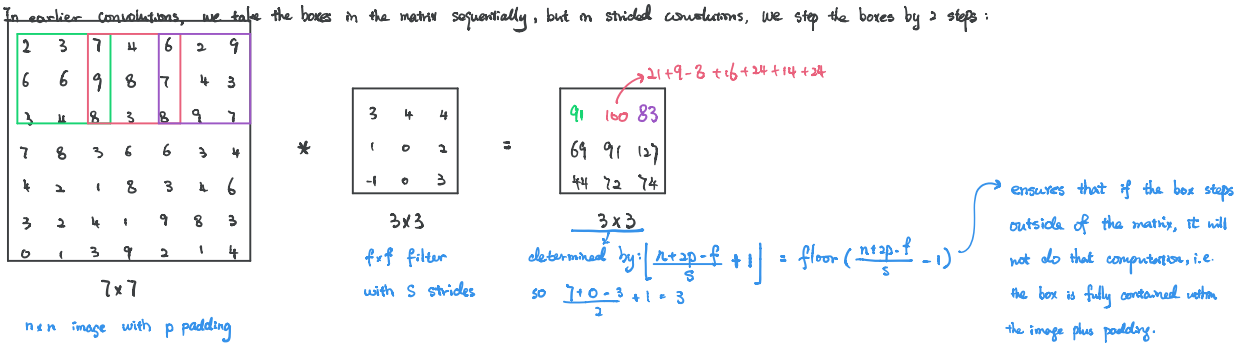
  where p: $n+2p-f+1 = n \Rightarrow p = \frac{f-1}{2}$

  when f is odd, we can make sure that output size is preserved. By convention, f is odd, because if f is even we would have some asymmetric padding to have same padding; second, odd filter has a central position

• Strided Convolutions

Strided convolution is another piece of basic building block of convolutions as used in CNNs.

In earlier convolutions, we take the boxes in the matrix sequentially, but in strided convolutions, we step the boxes by 2 steps:



$n \times n$ image with p padding

$f \times f$ filter with S strides

3×3 determined by: $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor = \text{floor}\left(\frac{n+2p-f}{s} - 1\right)$

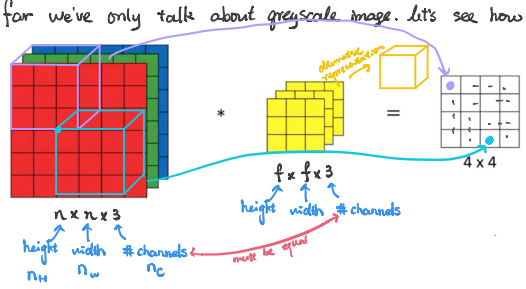
so $\frac{7+0-3}{2} + 1 = 3$

ensures that if the box steps outside of the matrix, it will not do that computation, i.e. the box is fully contained within the image plus padding.

note: so far we call this operation as convolution operations, but really it should be cross-correlation. In convention we call it convolution operation.

• Convolutions over volume

So far we've only talk about greyscale image. Let's see how convolution works on RBG images.



Andrew Ng

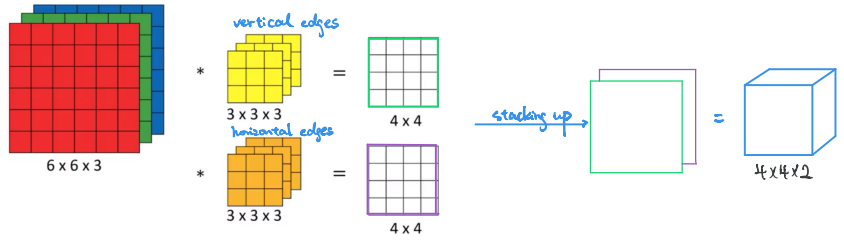
→ convolutions operations of each corresponding channel (i.e. R and 1st channel in filter, G & 2nd channel of filter, B & 3rd channel of filter) and then add them up together

→ detect edges in red colour:

detect edges in any colour:

By stacking up output matrixes, we can detect more features at the same time. For example, if we want to detect vertical edges and horizontal edges at the same time:

Multiple filters



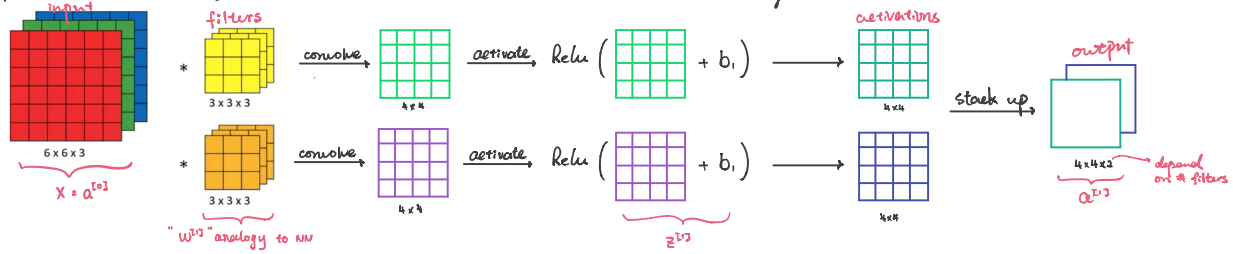
Andrew Ng

→ this example has neither padding nor stride. If it's done with padding or stride, output size is determined by: $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$

→ # channels of output is determined by the # filters used, so it could be up to hundreds of channels depending on how many features we want to detect at that layer

• CNN structure

Suppose we have an RBG image and we want to convolve it with 2 different filters, then one layer of CNN looks like:



Andrew Ng

So in short, x (6×6×3) $\xrightarrow{1 \text{ layer}}$ $a^{[1]}$ (4×4×2)

parameters = (width × height × # channels + 1) × # filters ⇒ makes CNN less prone to overfitting as comparing to the images, # parameters is very small and fixed

(+1 → bias)

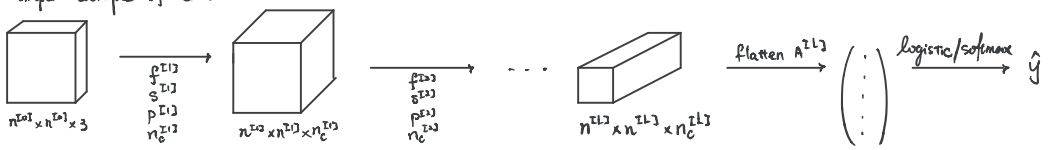
Notation: if layer l is a convolutional layer,

- $f^{[l]}$ = filter size
- $p^{[l]}$ = # padding
- $s^{[l]}$ = # stride
- $n_c^{[l]}$ = # filters
- $n_H^{[l]}$ / $n_W^{[l]}$ = height / width
- for $n_H^{[l]}$ or $n_W^{[l]}$, $n^{[l]} = \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1$

Sizes:

- input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$
- output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- each filter is $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$ → to match input
- activation matrix $A^{[l]}$: $m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ stack of output
- weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ stack of filters
- bias: $n_c^{[l]}$ or $1 \times 1 \times 1 \times n_c^{[l]}$ stack of bias

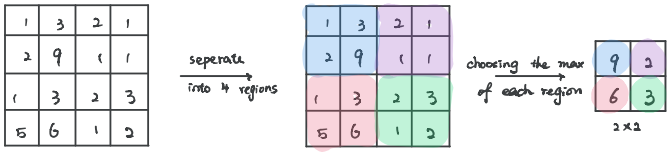
An simple example of CNN:



So far we have only talked about one type of layer in a CNN, which is convolution. There are other types of layer as well: pooling and fully-connected.

- pooling: can reduce size of the representation, to speed up computation as well as make some of the features that detects a bit more robust.

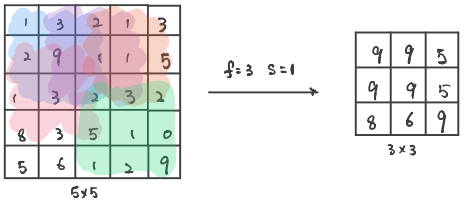
suppose we have a 4×4 input, max pooling is as follow:



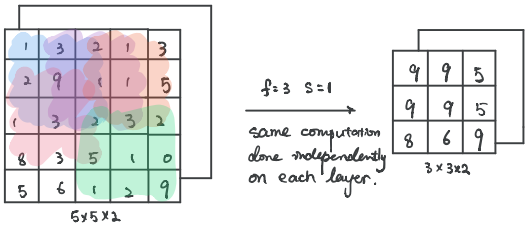
hyperparameters:
$f = 2$ (because output is 2×2)
$s = 2$ (because each region is 2×2)

note:
- there is no parameters for gradient descent to learn because f and s are fixed and
- $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$ also works for max pooling but p is often 0 (no padding)

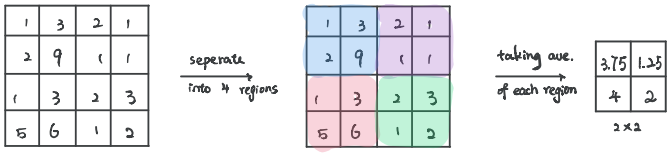
another example,



volumn max pooling,

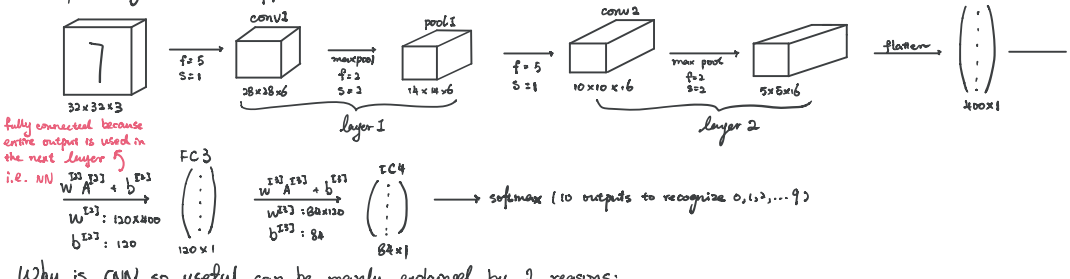


another pooling type known as average pooling is used less often.



- fully-connected

inspired by LeNet-5, suppose we have a CNN:



Why is CNN so useful can be mainly explained by 2 reasons:

- parameter sharing: a feature detecter (e.g. vertical edge detecter $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$) that's useful in one part of image is probably useful in other part of the image ⇒ a detector be used multiple times in one layer ⇒ reduce # parameters used
- sparsity of connections: in each layer, each output value depends only on a small number of input (outputs don't affect each other)