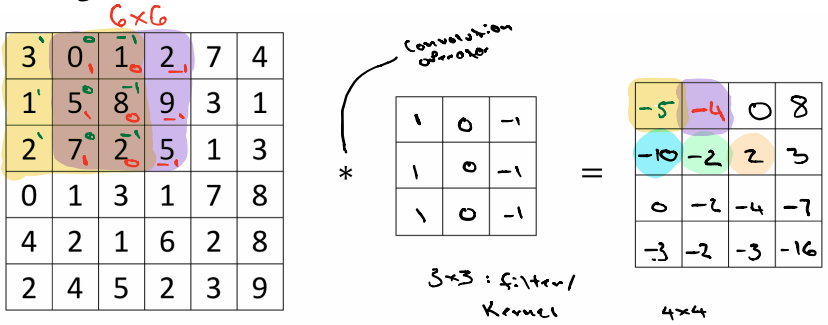
Course 4, Week 1: Foundations of Convolutional Neural Networks

Computer vision

- Rapid advances in computer vision are enabling brand new applications that weren't possible a few years ago.
  - Examples include image classification, object detection, neural style transfer (repainting a content image in a certain artistic style).
- One of the challenges of computer vision problems is that the inputs can get really big.
  - $64 \times 64 \times 3 = 12288$ (3 ~ color channels) ~ input feature size for example
  - Larger images: $1000 \times 1000 \times 3 = 3$ million input features
    - If we utilize 1000 hidden units in our first hidden layer, our weight matrix $W^{[1]} = (1000, 3\text{million})$ in a standard fully connected network
      - The matrix would have 3 billion parameters.

        This is computationally expensive and it's unlikely that we'll find enough data to train our network.
- To scale up, we must use the convolution operation.

Edge Detection Example



i. take the element-wise product of the matrix and the filter overlayed on top of it: $3\times1 + 1\times1 + 2\times1 + 0\times0 + 5\times0 + 7\times0 + 1\times-1 + 8\times-1 + 2\times-1 = \boxed{-5}$

ii. shift the overlayed square one step to the right and repeat the process: $0\times1 + 1\times0 + 2\times-1 + 5\times1 + 8\times0 + 9\times-1 + 7\times1 + 2\times0 + 5\times-1 = -4$

... and you continue as such ...

... after shifting to the right, we begin shifting down

3	0	1	2	7	4
1^{1}	5^{0}	8^{-1}	9	3	1
2^{1}	7^{0}	2^{-1}	5	1	3
0^{1}	1^{0}	3^{-1}	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

3	0	1	2	7	4
1	5^{1}	8^{0}	9^{-1}	3	1
2	7^{1}	2^{0}	5^{-1}	1	3
0	1^{1}	3^{0}	1^{-1}	7	8
4	2	1	6	2	8
2	4	5	2	3	9

3	0	1	2	7	4
1	5	8^{1}	9^{0}	3^{-1}	1
2	7	2^{1}	5^{0}	1^{-1}	3
0	1	3^{1}	1^{0}	7^{-1}	8
4	2	1	6	2	8
2	4	5	2	3	9

... and we continue until we have covered all the elements of the matrix ...

* By convolving in this way, we obtain a vertical edge detecton

• why does this work as a vertical edge detector?

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

6x6

*

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

3x3

=

4x4

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

* the white region in the middle corresponds to the detected vertical edge.

- the line in the resulting image is thick because we are working with a 6×6 image of (low resolution). If our image were 1000×1000 instead, our filter would do a more granular job in detecting the vertical edges of the image.

More edge detection

- positive edge: light to dark vs. negative edge: dark to light

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

*

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

=

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

- light to dark matrix resultant

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	10	10	10

*

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

=

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

- dark to light values.

- horizontal edge detector:

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

- Sobel filter (vertical edge detector):

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

↳ this filter has a little more weight on the central pixel making the processing a bit more robust

* in the deep learning era, we've learned that hand picking nine numbers might not be as useful as just learning them. We can treat these nine numbers as parameters and use backpropagation such that we can convolve it with our image and obtain a good edge detector. (can better learn the statistics of our dataset compared to something hand coded.

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

Instead of learning for horizontal and vertical edges, we might be able to learn edges that are 45°, 70° or the orientation that's best suited for our problem.

Padding (modification to the convolution operator)

- when we applied the 3×3 ($f \times f$) filter to our 6×6 ($n \times n$) image (through convolution) our output image shrunk to 4×4 ($n-f+1 \times n-f+1$).
  - the downside to this technique is that we can only convolve our image a few times before our image starts getting very small (shrinking down to 1x1) (we don't want our image to shrink every time we do an edge detection operation.)
  - the second downside is that the edge pixels (in the image) are only used for one operation, while the inner pixels overlap more (so they're used for more operations in the output)

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

- we are throwing away a lot of information near the edges of the picture.

- to solve both these problems, we can pad the image before applying the convolutional operation. (imagine adding a border of one pixel to the image)
  - if the amount of padding at the border = p, then the output after the convolution operation will be $(n + 2p - f + 1) \times (n + 2p - f + 1)$

- how much should we pad our images? (valid and same convolutions)
  - valid: no padding. $n \times n * f \times f \longrightarrow n - f + 1 \times n - f + 1$
  - same: pad so that the output size is the same as the input size.
    - $n + 2p - f + 1 \times n + 2p - f + 1$
      - for $n + 2p - f + 1 = n \longrightarrow p = \frac{f-1}{2}$
        - f which is the filter rows or columns is usually odd as an odd # in computer vision to find a whole number p (symmetric). when we have an odd 3x3 or 5x5, there's an absolute central position to the filter (central pixel)

Strided Convolutions

- strided convolutions happen when we jump to the right or up by more than one step as we convolve the

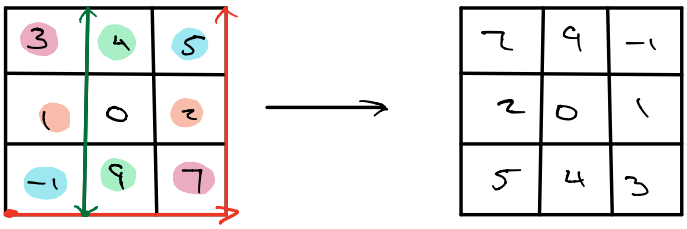
filter with the image. Stride is denoted by S.

- When adding stride, the output is denoted as,

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

  - where, n is the image's pixels, p is the # of padding pixels at an edge, f is the filter's pixels and s is the size of the stride.
  - what if the fraction is not an integer?
    - we floor the output such that if our input image (image + padding) has regions where the filter lies outside, these specific computations aren't performed.

- Convolution vs Cross Correlation:
  - actual convolution (double mirror):

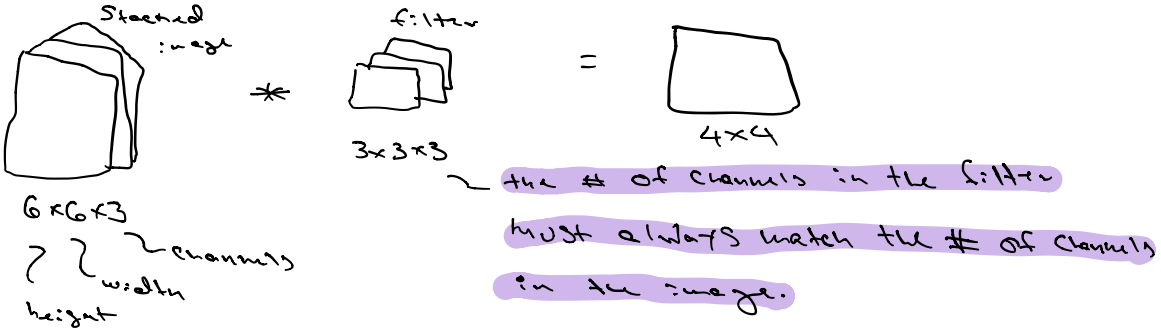


  - In actual convolution for signal processing, we double mirror the filter matrix before overlaying it into the image matrix.
    - this enables the conservation of associativity such that, $(A * B) * C = A * (B * C)$
    - this property is not important in deep learning, therefore, we don't do this operation and just directly overlay

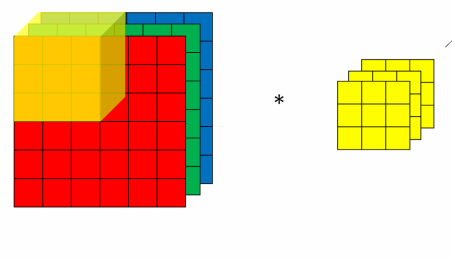
the filter on the image (saving computational cost). Hence, cross-correlation is just called convolution in deep learning applications as a convention.

Convolutions Over Volumes (3D)

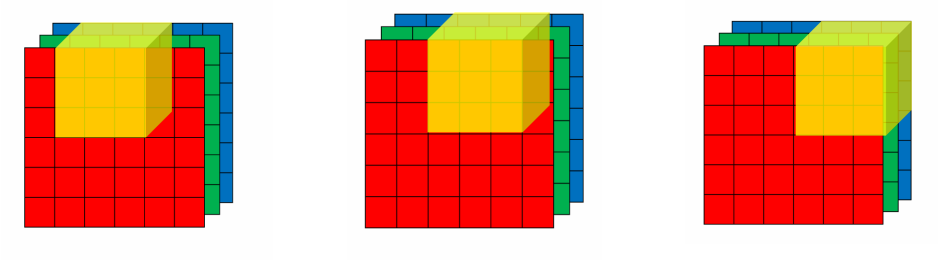
- Detecting convolutions on RGB images ($6 \times 6 \times 3$) ← volume



the # of channels in the filter must always match the # of channels in the image.



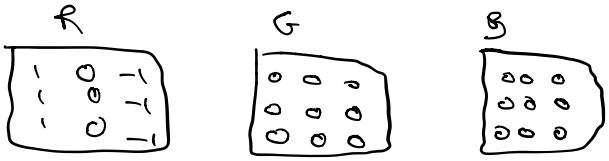
- We overlay the $3 \times 3 \times 3$ filter on the volume (similar to the 2D grayscale case) and shift it to the right and down sequentially.



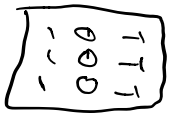
- The $3 \times 3 \times 3$ filter has a total of 27 numbers as we overlay them on top of the volume, we do an element wise multiplication and then sum. We obtain a resultant

4x4 matrix (similar to the 2D case).

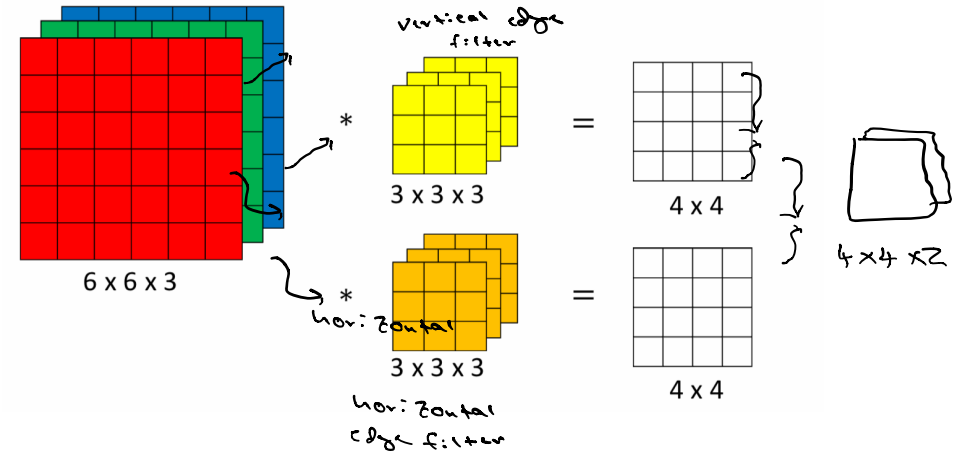
- what so special about analyzing volumes?
  - we can choose to find edges in particular channels, for example to detect vertical edges in the red channel...



we stitch this together to form the 3x3x3 filter.

- alternatively, we can set all the filters to the  values to find vertical edges in any channel.

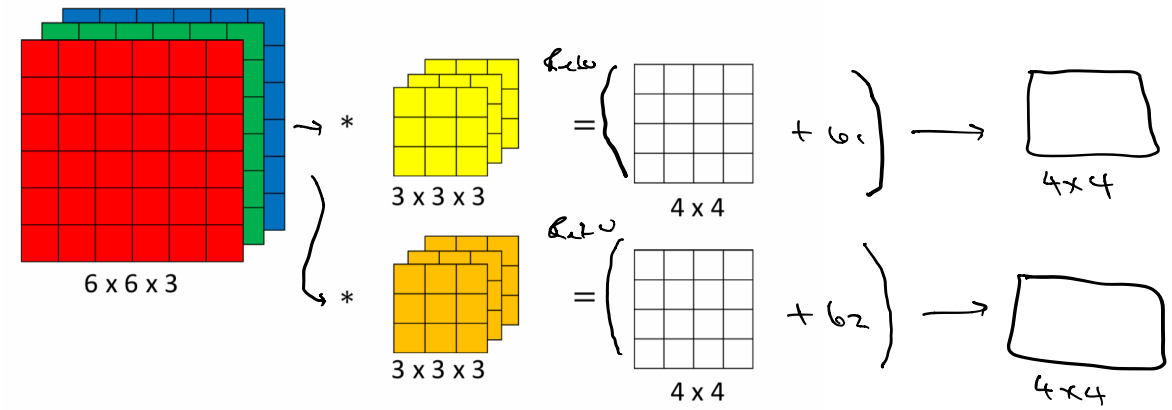
- how do we combine multiple filters to detect both horizontal and vertical (or other types) of edges at the same time?



* the # of depth in the output is equal to the # of features being extracted from the volume (in this case

we applied two filters and stacked them together, hence, our depth = 2).

One layer Implementation of a convolutional neural network



- to make the output a convolutional network layer, we add a bias (through Python broadcasting) and apply a non-linear activation function for each of the outputs.
  - After this step we stack both outputs, to make a 4x4x2 volume.
  - the filters acts similar to the weight matrix in a normal neural network scenario, $z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$
  - we transform the 6x6x3 into a 4x4x2 after passing one layer of the convolutional neural network.
  - Because the # of parameters is only dependent on the size of the filters, the # of filters and the bias term, the # of parameters that must be learned are fixed for an image of any input size.
    - 10 filters that are 3x3x3 (+1 from the bias)

have a total of 280 parameters that must be learned regardless of the size of the input image (resolution).

└ this feature of the convnet makes it less prone to overfitting (the small training set)

- Summary of notation (Convolutional neural network layer)
  - $f^{[l]}$ = filter size
  - $p^{[l]}$ = padding (valid vs same convolution)
  - $s^{[l]}$ = Stride

└ input = $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ ← activations from previous layer
(# of channels)
└ if $l=1$, it's the size of the input image

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

└ the output width & height are given by,

$$n^{[l]} = \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \quad \text{(and we floor this \#)}$$

* this formula can be used for either the height or the width.

* $n_c^{[l]}$ in the output is only dependent on the # of filters that we process in the layer of the neural network.

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

└ the activation dimensions are, $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

└ if we batch the examples (for batch gradient descent),

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

└ where m is the index of the training example

└ the weights are given as, $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n^{[l]}$

└ we take into account both the # of filters being

inputted from the previous layer and the # of filters being processed in the current layer

- Why?
  - We will have one real # for each filter being processed within the layer, hence, the dimensions are $n_c^{[l]}$

A Simple Convolution network example

* the last step of a convolutional neural network is to flatten the output of the last layer (i.e. say 7x7x40) into units (1960 units) and then densely connecting those units to a single final output neuron which uses a logistic regression or Softmax to compute the output $\hat{y}$.

- a lot of the work in designing convolutional neural networks is selecting the hyperparameters such as the total size, stride, padding in each layer, how many filters to use, and so on.
  - a suggestion is that as we go deeper in a convnet, we want the height and width to gradually trend down as we go deeper into the network and the # of channels to increase
    - 39 x 39 x 3 → 37x37x10 → 17x17x20 → 7x7x40

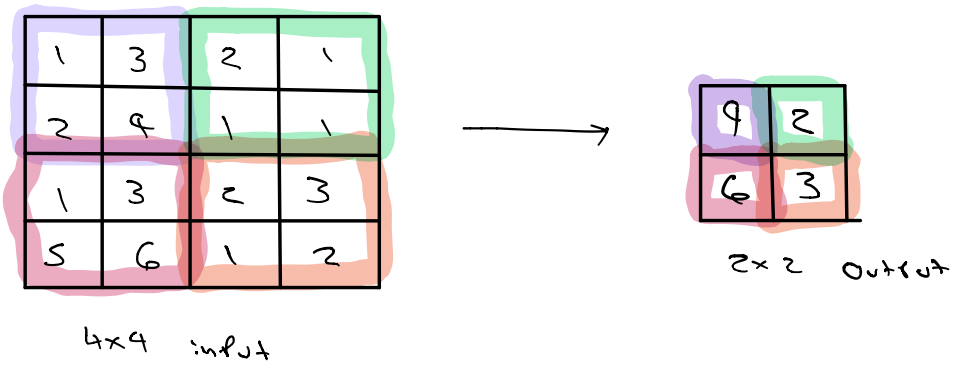
- three types of layers in a convolutional neural network
  - convolution (Conv): what we have been using up to this point.
  - Pooling (POOL)
  - Fully connected (FC): like the standard neural network layer

* it's possible to design a good neural network with only Conv layers. However, most neural network architectures also have pooling layer and fully connected layers.

Pooling Layers

- pooling layers are used to reduce the size of the representation to speed up computation as well as make some features detected a bit more robust

- pooling layer: max pooling



4x4 input

2x2 output

↳ take the 4x4 input, break it into different regions and in the output, each of the four #s will be the max from the corresponding shaded region.

↳ it's like we are applying a filter of $f=2$ with a stride of 2. $(4-2)/2 + 1 = 2$

↳ What's the intuition behind what max pooling is doing?

↳ if we think about the 4x4 as some set of features (the activations in some layer of the network) then

a large # might mean that we've detected a particular feature.

- an interesting property of max pooling is that it has a set of hyperparameters, but it has no parameters to learn.
  - once f and s are set, it's a fixed computation.

- if we have a 3D input, the output of max pooling will have the same # of channels/ depth as the input. The process of taking the max remains (computed on each channel independently).

- another type of pooling which is used less often is average pooling.
  - in average pooling we take the average of each region instead of the max.
  - an exception to when average pooling is used over max pooling is very deep in the network when we want to collapse our representation
    - $7 \times 7 \times 1000 \rightarrow 1 \times 1 \times 1000$

* By using hyperparameters $S=2$ $f=2$, the height and width are roughly shrunk down by a factor of 2. when we do pooling, we almost never use padding.

Why Convolution?

- two main advantages of using conv layers instead of only

using the fully connected layers variant.

i. Parameter sharing: 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.

- if we figure a 3×3 filter to detect edges, we can apply the same 3×3 filter to all the input.
- this applies to both low level features (edges) and high level features (such as faces)

ii. Sparsity of connections: In each layer, each output value depends on a small # of inputs (the size of the overlayed filter / shaded region of the input where we are performing the convolution).