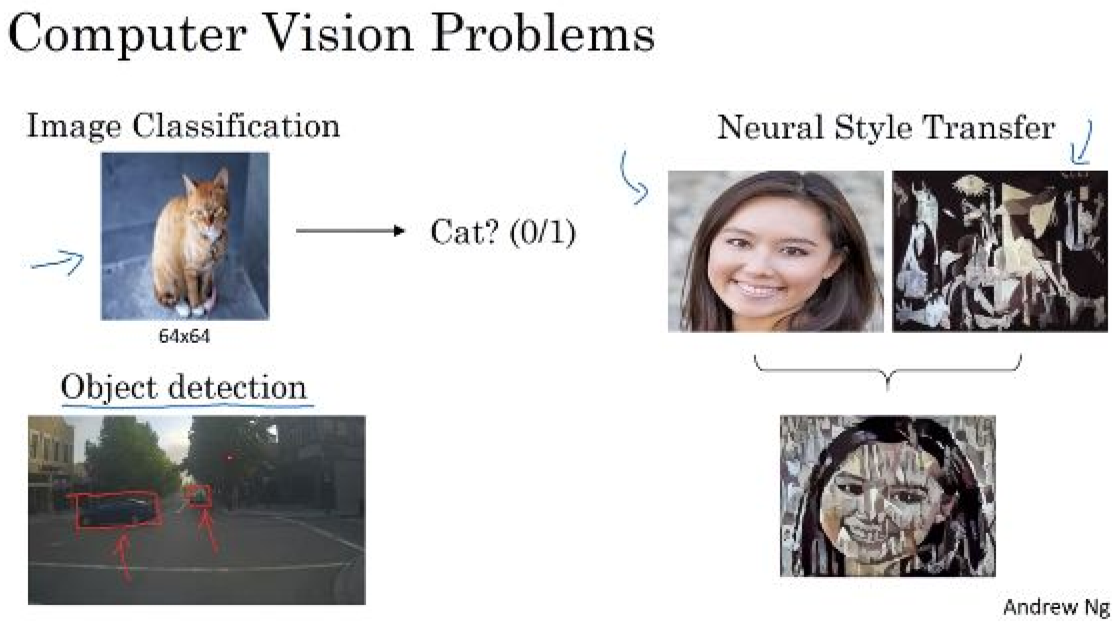
Andrew Ng’s Convolutional Neural Networks

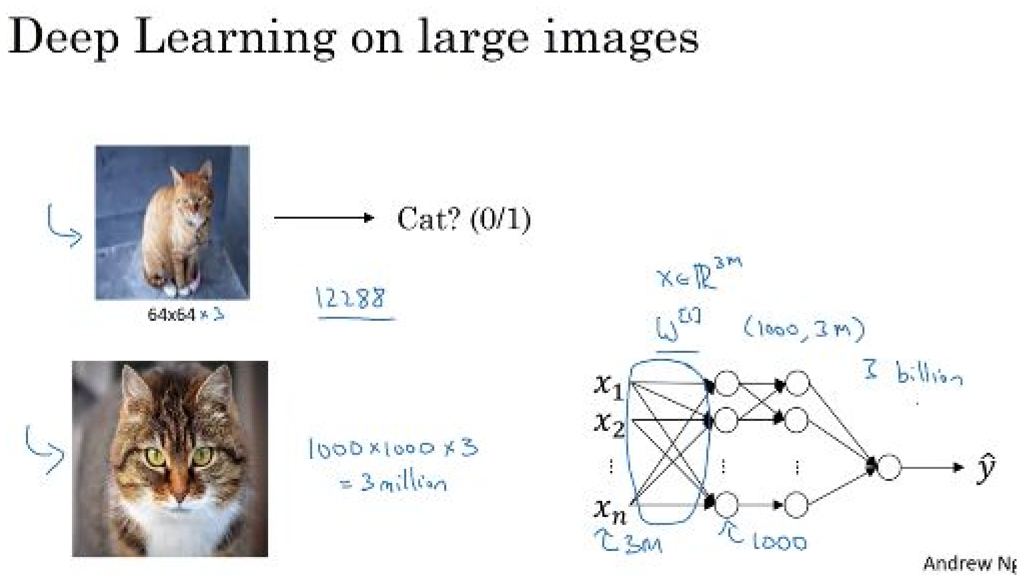
Week 1: Foundations of Convolutional Networks

Computer Vision

- Computer vision is one area that has advanced rapidly thanks to deep learning. Some of its applications are image classification, object detection and neural style transfer (where a neural network can combine 2 images as shown below).

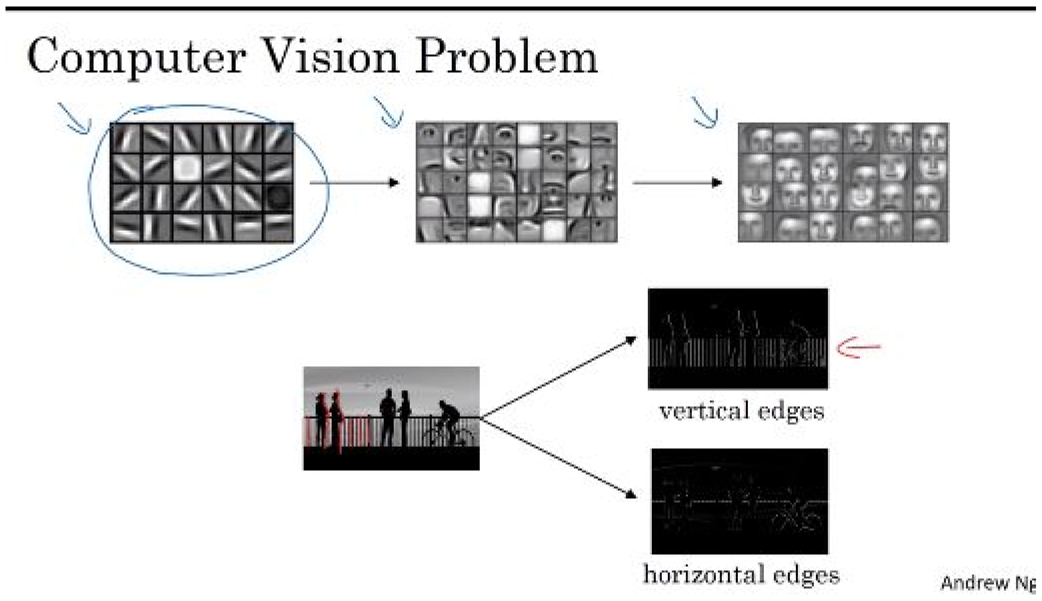


- One of the problems of computer vision is that the inputs can get really big. Below we can see a low res 64x64 image whose size is familiar to us. This size is not too bad but a bigger, higher res image (say 1m by 1m) means our network will have 3 billion parameters which is very large and is not feasible, computationally speaking. However, we don’t want to be stuck doing computer vision with small images, we want to use large images. For that what we do is to implement the convolution operation, which is a fundamental block of convolutional neural networks. We’ll start illustration this concept with edge detection in the next session.



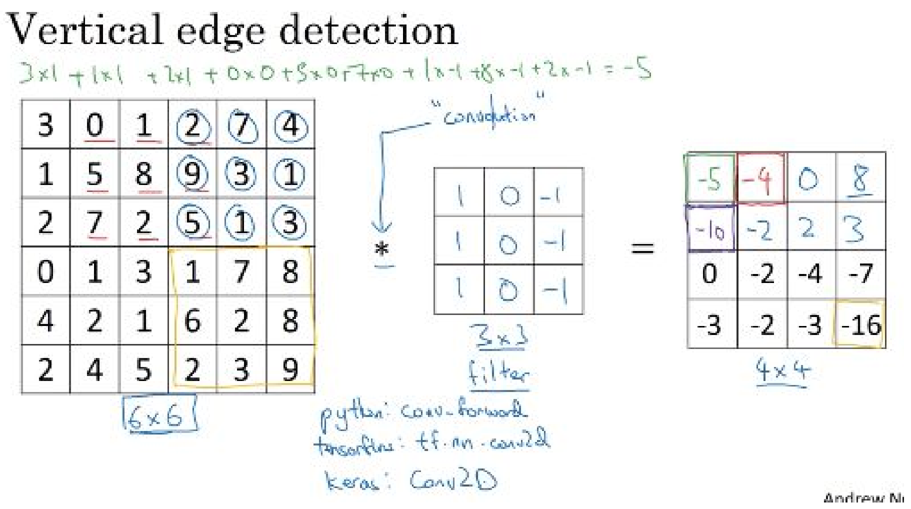
Edge Detection

- We are going to see how convolution works by using edge detection as an example. In previous videos we have seen how a neural network might start by detecting edges and then late layers might detect the complete object. In the image below we can detect vertical edges and horizontal edges.



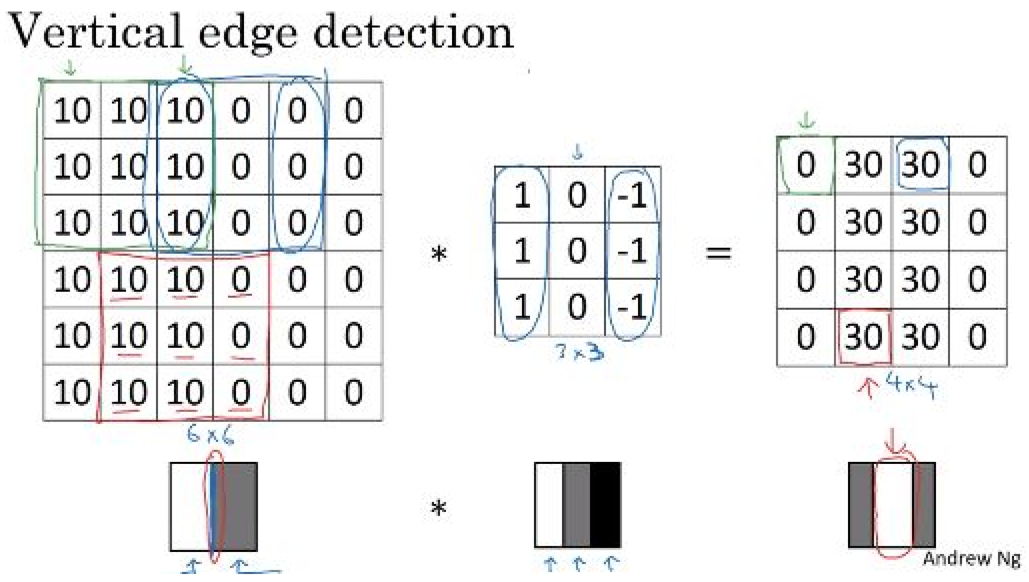
- How do we detect edges? Let's start with vertical edge detection. Below we have a 6x6 matrix that we can think of as a grayscale image. We then have a 3x3 filter matrix and we are going to do the convolution operation with these matrices. Convolution works as follows:

- We take the filter matrix and super-impose it on the top left corner of our 6 x 6 matrix and we then do the math (3x1 + 1x1+2x1+0x0+5x0+7x0+1x-1+8x-1+2x-1=-5). We then move our filter one column to the right and do the same operations again
- Once we are done with the first row we move our matrix one row down and we do the same operation until we are done with the matrix and we should get a 4x4 result matrix as the one below.



The convolution operation is available in all deep learning frameworks. In python it's called $conv\,forward$, in tensorflow it's called $tf.nn.conv2d$ and in keras is called $Conv2D$.

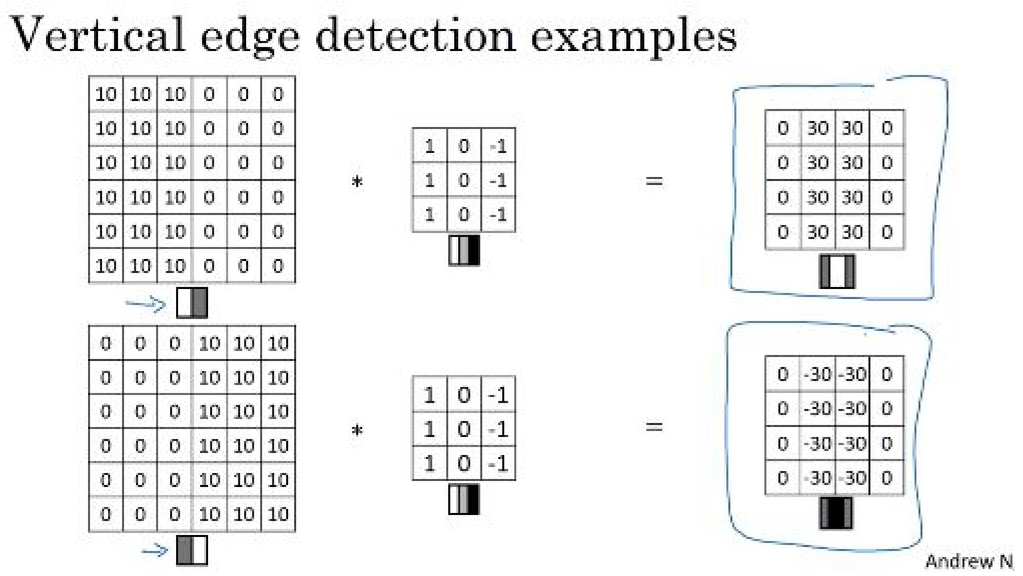
- How does this detect a vertical edge? Let's illustrate it with the example below. In this example we have the 6x6 image that is graphically represented as below, it has a white portion on the left and a gray portion on the right so there is a vertical edge in the middle.



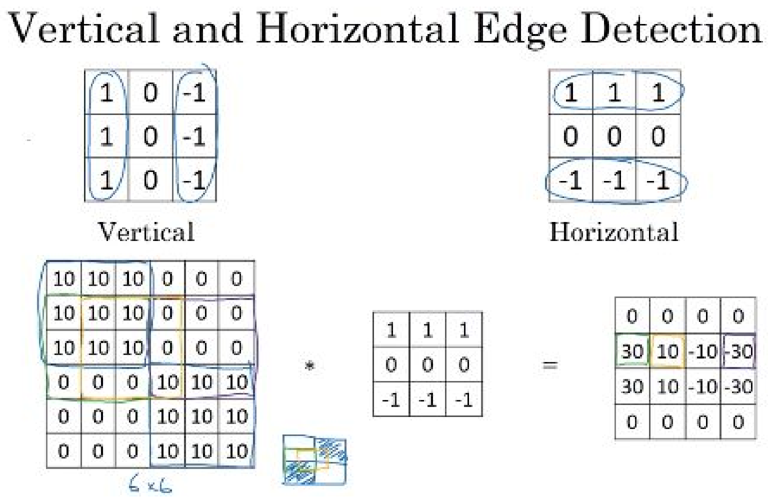
We take our 3x3 filter and do the convolution operation and we end up with the output matrix on the right, which can be graphically represented as shown above. This is how the image is telling us there is a vertical edge right in the middle of the left image. Our output image might look like it has the wrong dimensions but that is only because we are working with a 4x4 matrix, in practice (with big images) convolution does a very good job of detecting edges. An intuition to take away is that a vertical edge is a 3 x 3 region with bright pixels on the left and dark pixels on the right and we don’t care what’s in the middle.

More Edge Detection

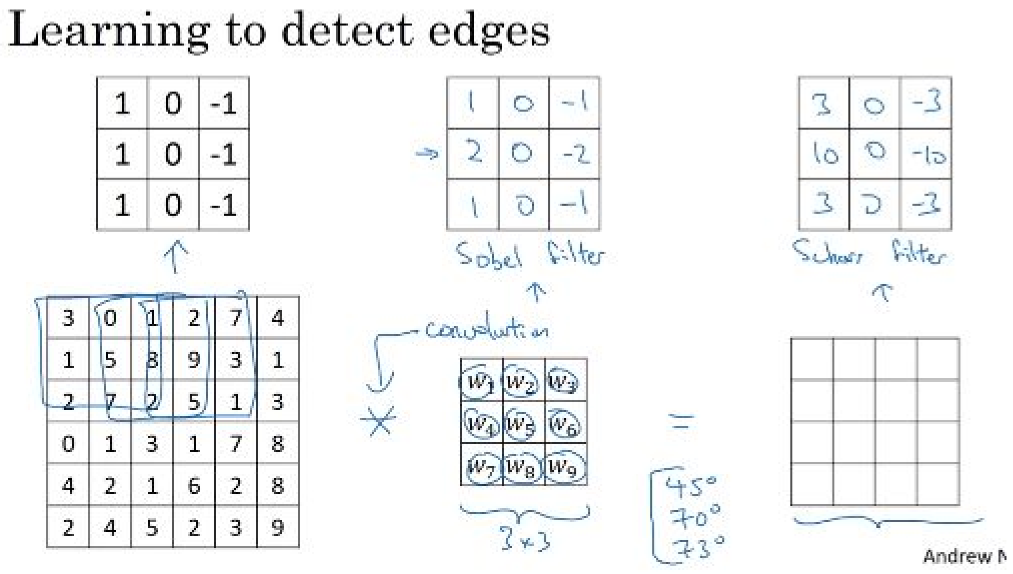
- Below is the example from the last section where we detect a light to dark edge transition. What happens if we flip the input matrix and do the convolution operation again? We end up with the 2nd row output. In this case we have a dark to light edge transition. So we have a filter that can detect both a light to dark transition and a dark to light transition.



- We have so far been using a vertical filter. We can also use a horizontal filter like the one below. If we apply the horizontal filter to the image on the left we end up with an output matrix like the one shown below. In summary, different filters allow us to find vertical and horizontal edges.



- It turns out that we have different vertical filters. There is the filter we have been using, others are the *Sobel* filter and the *Scharr* filter so we can use those filters too. What is more interesting is that instead of using a 'predefined' filter we can treat the filter numbers as parameters that can be learned by the neural network. This gives us the ability to come up with a filter more appropriate for the problem we are trying to solve. The parameters are learned in the back prop process of the neural network.

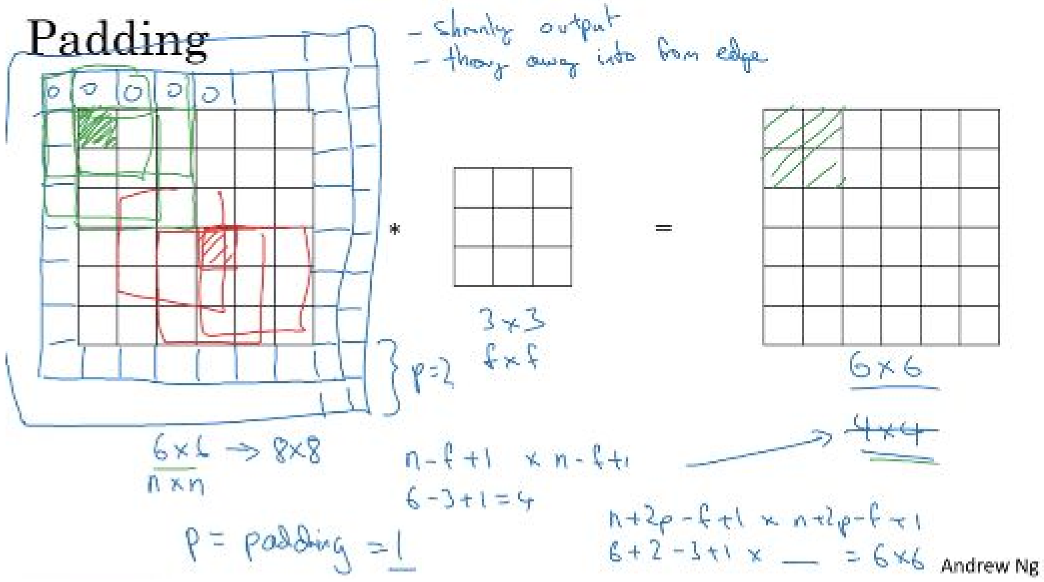


Padding

- There are a couple of drawbacks to the convolution operation we have seen so far. They are
  - Shrinking output: this the fact that we start with a matrix and the resulting matrix is smaller in size. In the example below it goes from 6x6 to 4x4. This is a problem because if we do this operation multiple times (say, one time per neural network layer) we are going to run out of data. In general the matrix dimensions of the

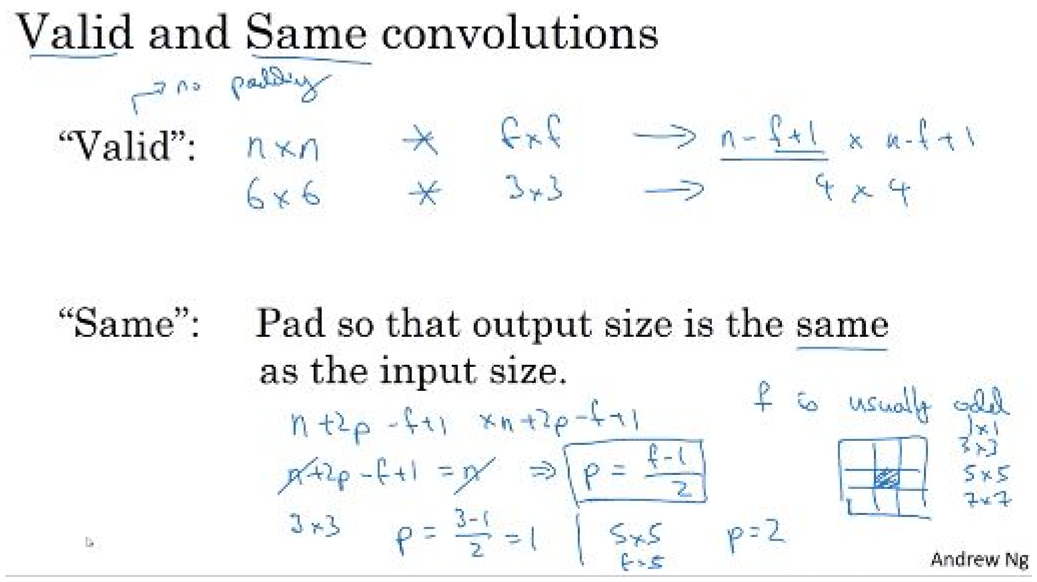
input matrix are nxn, the filter matrix has dimensions fxf and the resulting matrix has dimensions (n-f+1, n-f+1)

  - Throw away info from the edge: we can illustrate this concept with the top right pixel, that pixel in our normal convolutional operation is used only once while other pixels are used multiple times. We are then not using as much info as we have available.



The way around this is to introduce padding. Padding is the act of adding another set of pixels (by convention they have a value of 0). In the example above we added one pixel so p=1, adding that set of pixels means that our padded 6x6 matrix will output a 6x6 matrix. With padding the dimensions of the output matrix are $(n+2p-f+1, n+2p-f+1)$.

- How much to add? There are a couple of options: valid which means no padding or same which adds enough padding so that the output size is the same as the input size.



Valid and Same convolutions

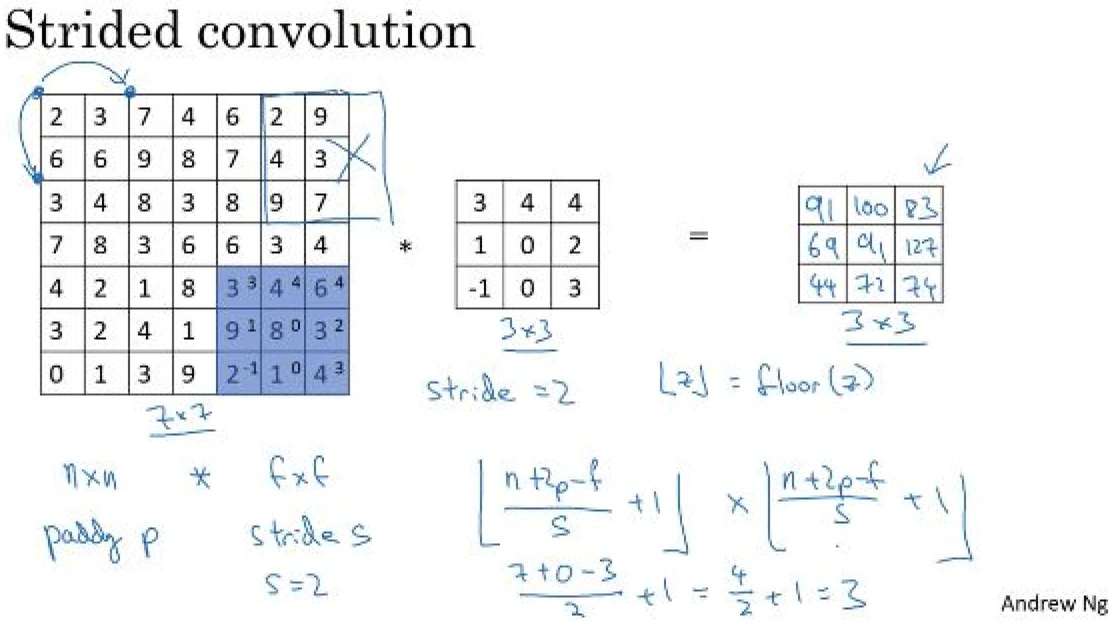
"Valid": $n \times n \ast f \times f \rightarrow n-f+1 \times n-f+1$
$6\times6 \ast 3\times3 \rightarrow 4\times4$

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

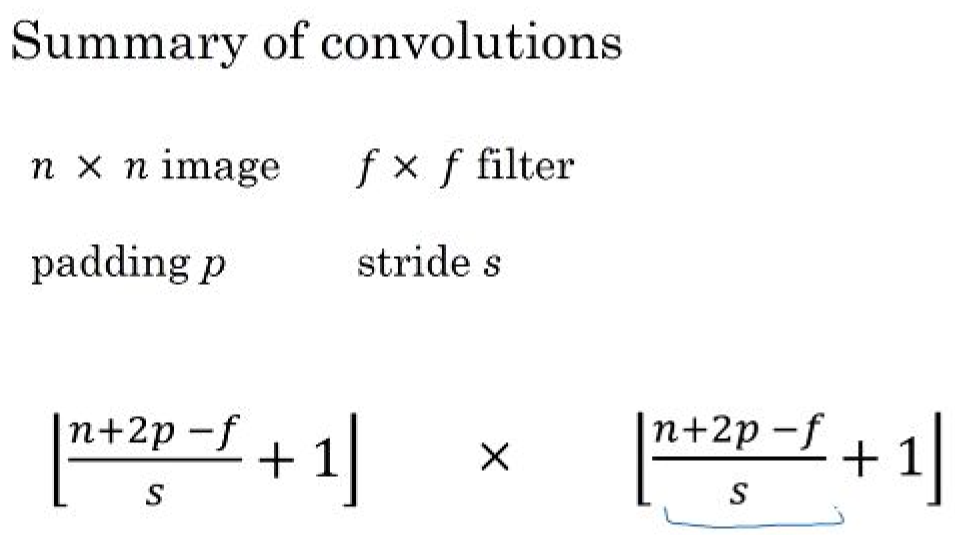
Professor Ng recommends we use odd number filters, it's kind of the convention in computer vision problems. There are 2 reasons for this choice: one is that it gives us a natural padding region (meaning we add the same padding everywhere) and the other one is that it gives us a central pixel so we can reference it regarding the position of the filter.

Strided Convolutions

- Strided convolutions is another foundational block for convolutional neural networks. Let's see how they work using the example below. In this example we have a 7x7 input matrix that we convolute with a 3x3 filter and we introduce a stride of 2. This means that the steps our filter takes 'through' the input matrix are done in multiples of 2 (before our steps were of length = 1). Given this, the new dimensions of our output matrix are $((n+2p-f)/s\ +\ 1),(n+2p-f)/2\ +\ 1)$ where s=stride. What do we do if the dimension calculation does not yield an int? We round down. This is called the flow of Z. One more detail is that if our stride causes us to have part of our filter to be outside the input matrix then we don't do that calculation.



- Summarizing strided convolutions, if we have an input matrix of (n,n) dimensions, a filter of (f,f) dimensions, padding p and stride s, the dimensions of the output matrix are $((n+2p-f)/s\ +\ 1),(n+2p-f)/2\ +\ 1)$.



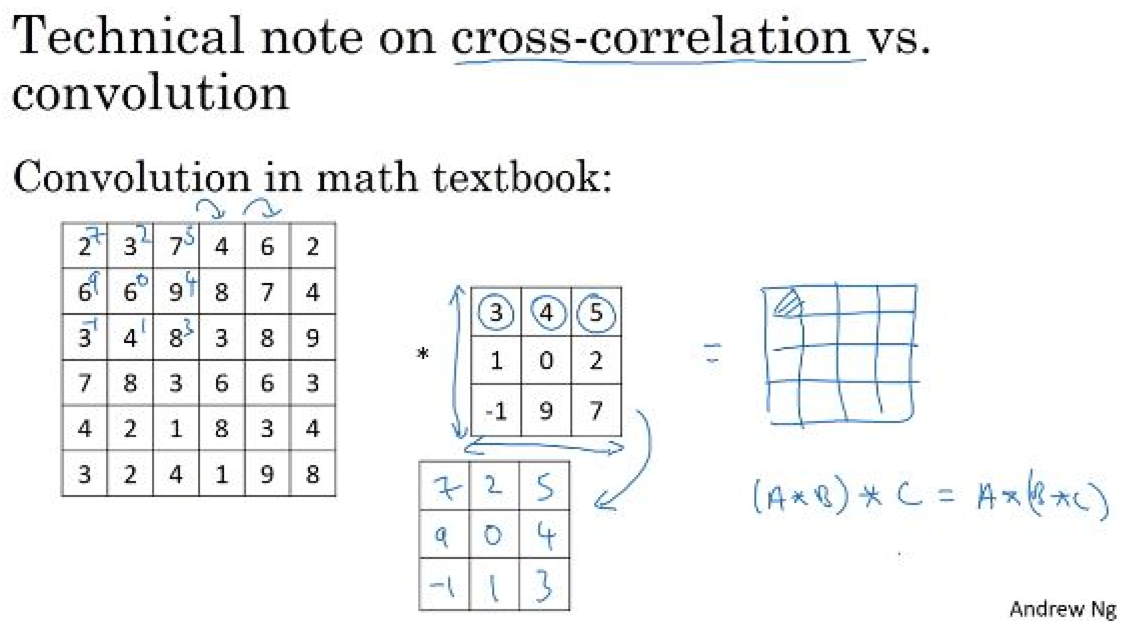
Summary of convolutions

$n \times n$ image $\quad f \times f$ filter

padding $p \quad$ stride s

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

- We have one technical note before moving on. Mathematicians call our convolution operation cross-correlation. A math textbook will define convolution as our same operation but with one extra step: the filter will be flipped as shown below. Everything else is the same. Deep Learning does not bother with this step and we will still call our operation convolution.



Technical note on cross-correlation vs. convolution

Convolution in math textbook:

$$\begin{bmatrix} 2^{7} & 3^{2} & 7^{5} & 4 & 6 & 2 \\ 6^{9} & 6^{0} & 9^{4} & 8 & 7 & 4 \\ 3^{-1} & 4^{1} & 8^{3} & 3 & 8 & 9 \\ 7 & 8 & 3 & 6 & 6 & 3 \\ 4 & 2 & 1 & 8 & 3 & 4 \\ 3 & 2 & 4 & 1 & 9 & 8 \end{bmatrix} * \begin{bmatrix} \textcircled{3} & \textcircled{4} & \textcircled{5} \\ 1 & 0 & 2 \\ -1 & 9 & 7 \end{bmatrix} =$$

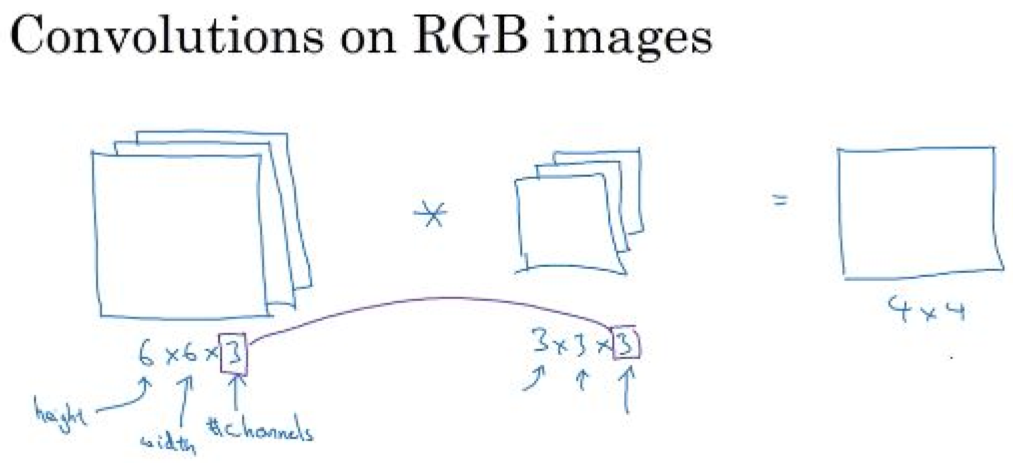
$$\begin{bmatrix} 7 & 2 & 5 \\ 9 & 0 & 4 \\ -1 & 1 & 3 \end{bmatrix}$$

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

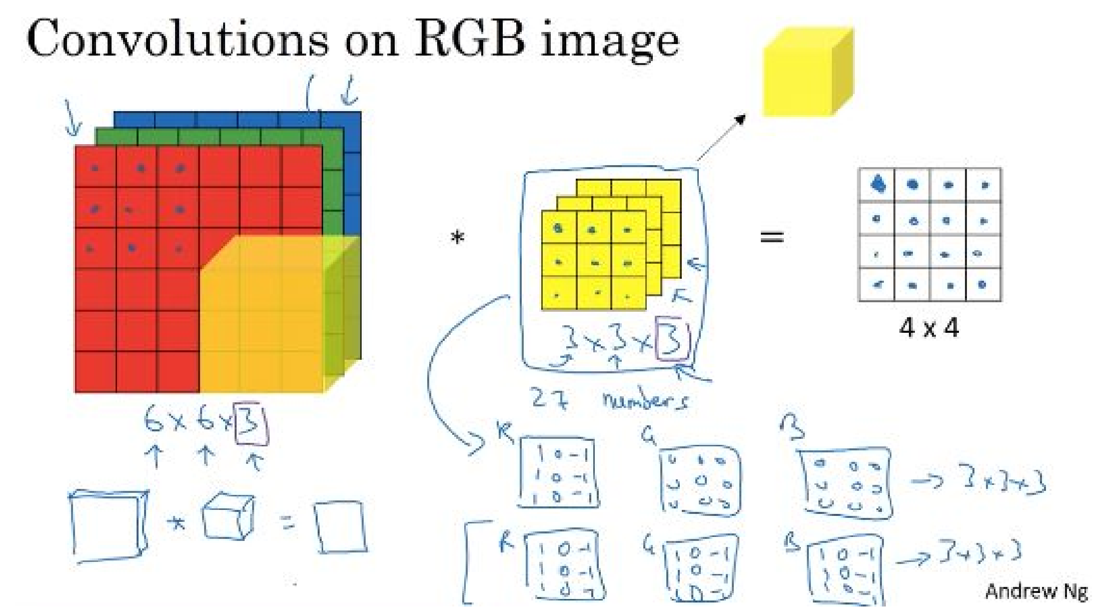
Andrew Ng

Convolutions over Volume

- Now that we have seen how convolution works on 2D images, let's see how it works on 3d images. Let's say we have as input a RGB image of dimensions 6x6x3; the first number is the height, the second number is the width and the 3rd number is the number of channels. Our corresponding filter will also have the *same* number of channels (it is a requirement). In our example below our filter has dimensions (3, 3, 3) and the output image is of dimensions (4, 4).

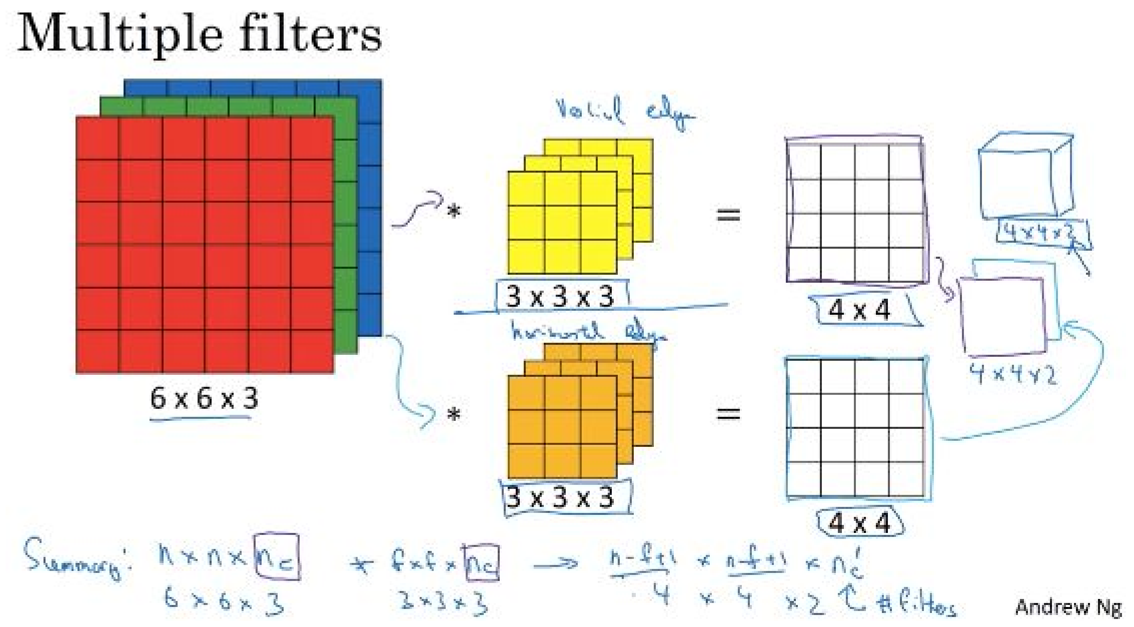


- Let’s see how the convolution operation works on 3d images: What we first do is take our 3x3x3 filter and create a cube out of it. We then place that cube on the top left corner of our input image and what happens is the first 9 numbers are for the red matrix, the 2nd 3 numbers are for the green image and the last 9 numbers are for the blue matrix. Doing the math for those 27 numbers gives us the first value of our output matrix. We then move our cube one column to the right and do the same operation, then we move it one more column to the right and so on. We also move it one row below when we go to the end of the first column. This is how we fill our output matrix.



We could have different filters. We can have a filter that only detects edges in the red channel or a filter that detects edges on all channels.

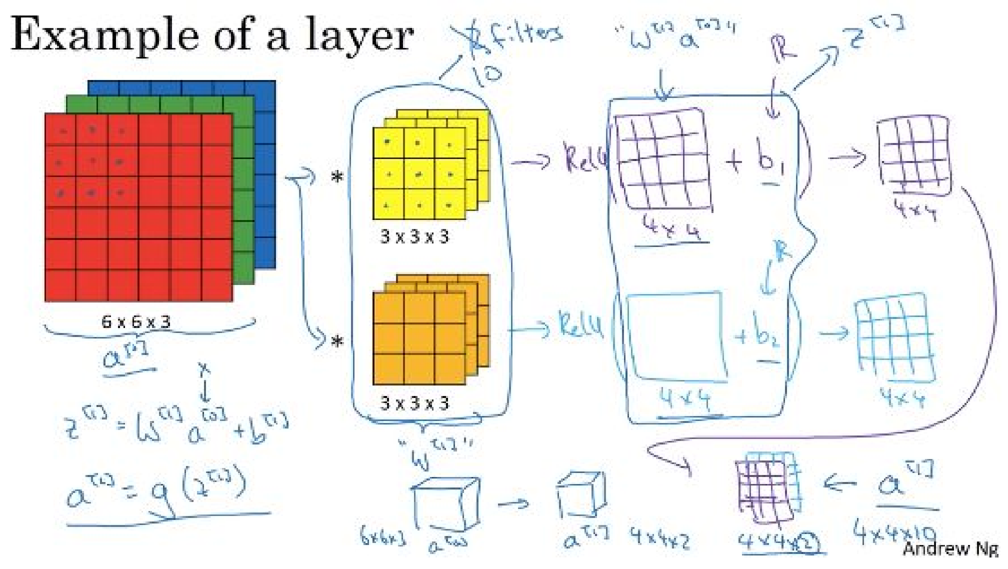
- How do we use multiple filters at the same time? We can have one filter that detects vertical edges and one filter that detects horizontal edges as shown in the example below. So basically we are doing the convolution operation twice, once per each filter and, as a result, we get 2 output matrices that can be stacked as shown below.



The general matrix dimensions are then: for output matrix $(n,\ n,\ n_c)$, for a filter: $(f,\ f,\ n_c)$ and for the output matrix we will have $(n - f + 1, n - f + 1,\ n'_c)$ where $n'_c =$ *number of filters.*

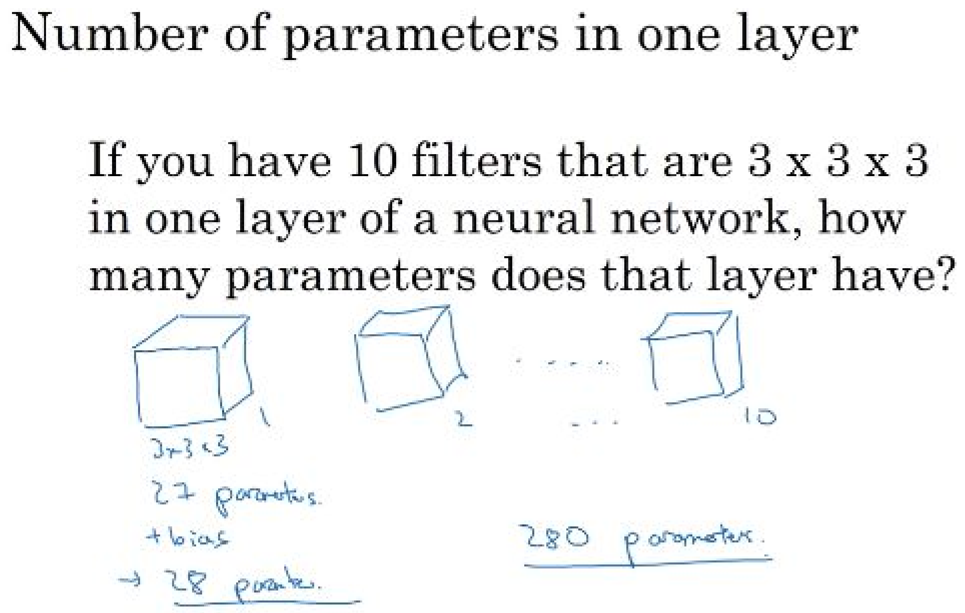
One Layer of a Convolutional Network

- Now we are ready to go over one layer of a convolutional neural network. We start with the example from the previous session. In that example we had an input matrix of dimensions (6,6,3), 2 filters of (3,3,3) and the output is 2 matrices of dimensions (4,4,2)



Remembering forward prop for a neural network we have $z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$ and then we do the activation function to get the input for the next layer: $a^{[1]} = g(z^{[1]})$. For convolutional networks, the input matrix of dimensions 6x6x3 is $a^{[0]}$, the filters are $w^{[1]}$ and the resulting matrix is $w^{[1]}a^{[0]}$. We then a bias number (the same for all) and the result is $z^{[1]}$. Adding all our output matrices gives us $a^{[1]}$.

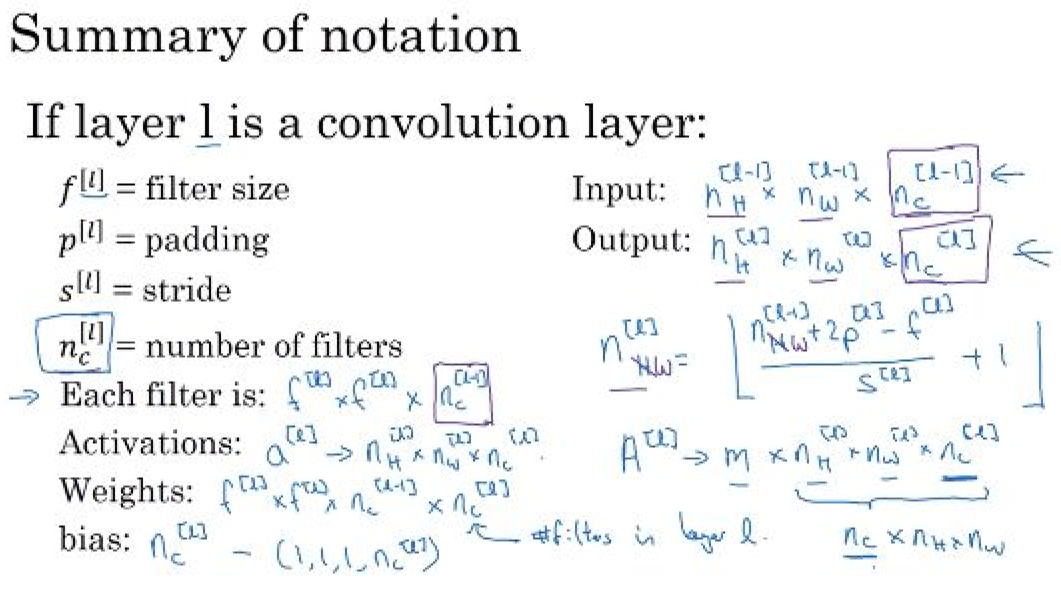
- Let's go through an exercise to check our understanding. If we have 10 filters of 3x3x3 we then have 27 parameters per filter + bias = 28 * 10 filters = 280 parameters. Notice that no matter how big the input images are our number of parameters remains fixed and this is one property of convolutional nets that make them less prone to overfitting



Number of parameters in one layer

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

- To wrap up, let's summarize the notation for one layer of a convolutional network. We have filter size $f^{[l]}$, padding $p^{[l]}$, stride $s^{[l]}$ and number of filters $n_c^{[l]}$.



Summary of notation

If layer $\underline{l}$ is a convolution layer:

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

$p^{[l]}$ = padding

$s^{[l]}$ = stride

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

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

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

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ ← #filters in layer l.

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

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

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

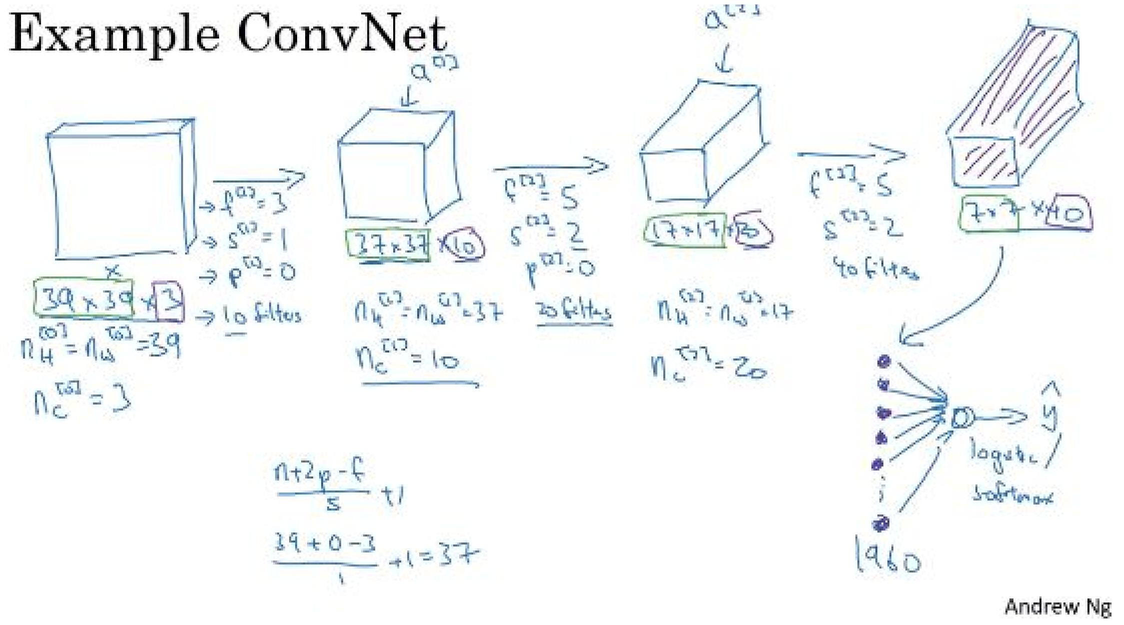
$$\underline{n_{H/W}^{[l]}} = \left\lfloor \frac{n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

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

The input matrix has dimensions $n_H^{[l-1]}\ x\ n_W^{[l-1]}\ x\ n_c^{[l-1]}$. The output matrix has dimensions $n_H^{[l]}\ x\ n_W^{[l]}\ x\ n_c^{[l]}$. Before we said the output height and width were $(n+2p-f)/s$. For this case, we modify the previous equation to $n_H^{[l-1]} = ((n_H^{[l-1]} + 2p^{[l]} - f^{[l]})/s^{[l]})\ +\ 1$. Each filter is of dimensions $f^{[l]}x\,f^{[l]}\ x\ n_c^{[l-1]}$. The activation $a^{[l]}$ is then of dimensions $n_H^{[l]}\ x\ n_W^{[l]}\ x\ n_c^{[l]}$ and the vectorized version is $A^{[l]} = m\ x\ n_H^{[l]}\ x\ n_W^{[l]}\ x\ n_c^{[l]}$ .The weight dimensions are $f^{[l]}x\,f^{[l]}\ x\ n_c^{[l-1]}\ x\ n_c^{[l]}$. Finally the bias has dimensions $n_c^{[l]}$.

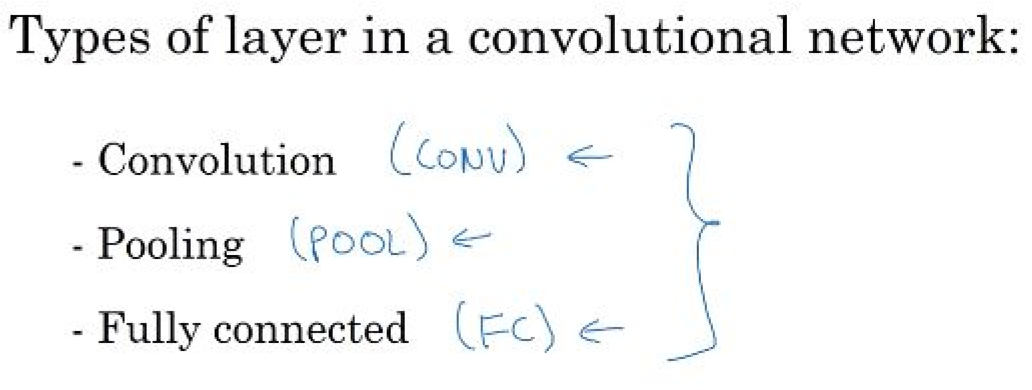
Simple Convolutional Network Example

- Let's take a look at an example of a convolutional network. We start with our cat classifier problem so our inputs are cat images. We then have an initial image of 39x39x3 (so $n_H^{[0]} = n^{[0]} = 39\ \&\ n_c^{[0]} = 3$) and let's say we have $f^{[1]} = 3,\ s^{[1]} = 1,\ p^{[1]} = 0$ and 10 filters. Our $a^{[0]}$ output is of dimensions 37x37x10. How did we get those dimensions? We applied the formula $((n+2p-f)/s)+1 = (39+0-3)/1\ +1 = 37$. 10 is the number of filters. That was the first layer and that becomes the input for the second layer. For the second layer let's say we have $f^{[2]} = 5,\ s^{[2]} = 2,\ p^{[2]} = 0$ and 20 filters. Following the same process, our $a^{[1]}$ output is of dimensions 17x17x20; in this case $n_H^{[0]} =$ $n^{[0]} = 17\ \&\ n_c^{[0]} = 20$. Finally, let's do one more layer. The last layers has parameters $f^{[2]} = 5,\ s^{[2]} = 2$ and 40 filters so our $a^{[2]}$ output is of dimensions 7x7x40. The last step is to unroll our output in one vector, in this case the vector is of length 1960 (49x40) and apply an activation, say logistic or softmax.



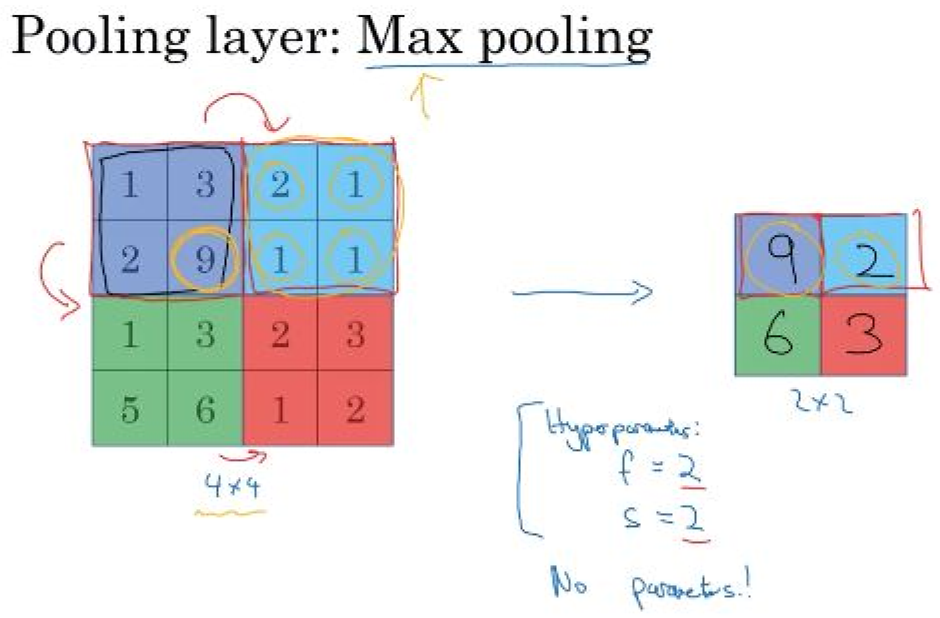
Notice how the size of the images decrease and the number of channels increase. This is common in convolutional networks.

- There are 3 types of convolutional network layers. They are convolutional (CONV and the kind we use in the previous example), Pooling (POOL) and Fully Connected (FC). Fortunately, pooling and fully connected are simpler to define than convolutional.


Types of layer in a convolutional network:
- Convolution (CONV)
- Pooling (POOL)
- Fully connected (FC)

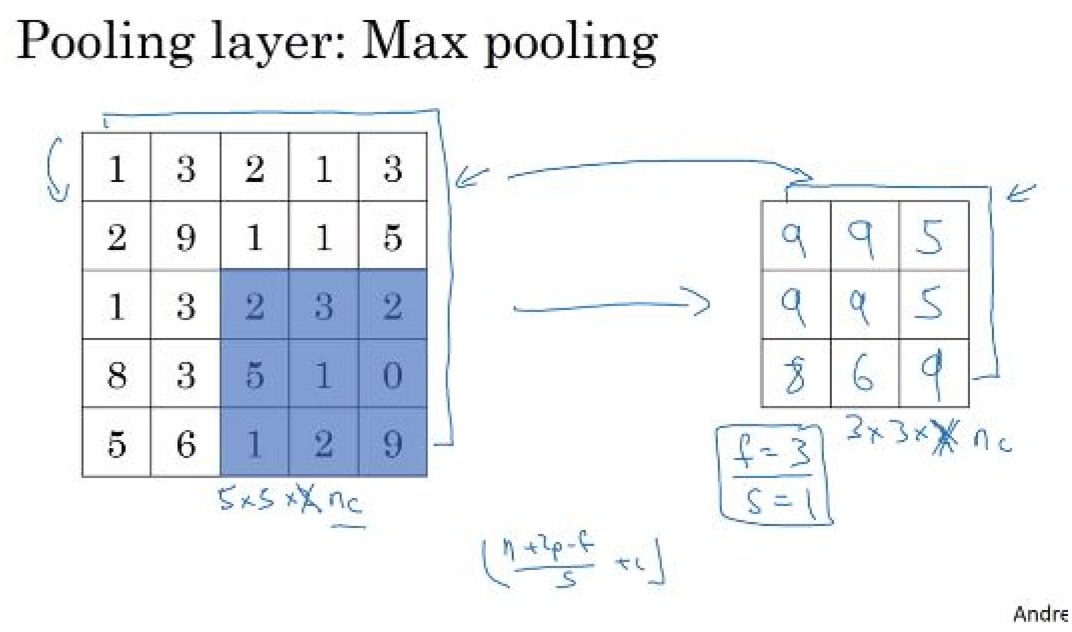
Pooling Layers

- Let's see what pooling does through an example. In the example below we have a 4x4 image and we are using something called max pooling. What we do is we divide our input image in 4 regions of 2x2 and we take the highest number of each quadrant as our output for the quadrant. In our example the quadrant has values 9, 2, 6 & 3. So the hyperparameters are s=2 and f=2.

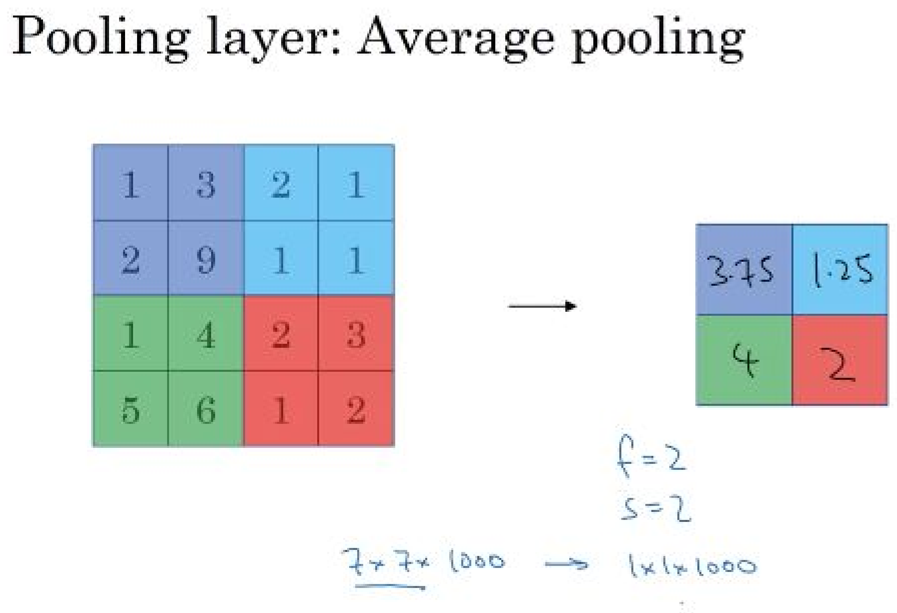


The intuition about pooling is that it preserves a feature that it has seen (the max value represents the feature).

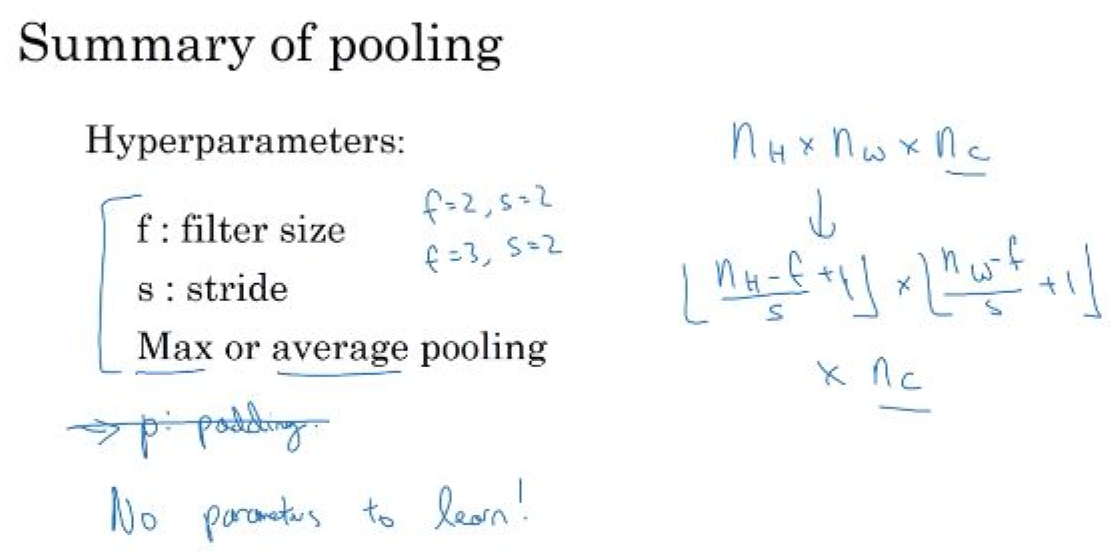
- Let's look at another example. In this case we have a 5x5 input image, a size of a filter =3 and stride = 1 so we then have an output of 3x3. In this example the resulting image has values 9,9,5,9,9,5,8,6 & 9. We can do this across multiple channels, we just do the same operation on each channel.



- There is another type of pooling layer that is not used very frequently so we are going to go over it briefly. It is called average pooling and it takes the average of each quadrant instead of the max value. Below the output image is composed of values 3.75, 1.25, 4 & 2. Average pooling is used deep in the conv net to dramatically reduce our representation.



- Summarizing the hyperparameters of pooling: f= filter size, s= stride and max or average pooling. If we have an input image of $n_H \ x \ n_W \ x \ n_c$ we will get an output of $((n_H - f)/s) \ + 1 \ x \ ((n_W - f)/s) + 1 \ x \ n_c$



Summary of pooling

Hyperparameters:

f : filter size

s : stride

Max or average pooling

$f=2, s=2$

$f=3, s=2$

→ ~~p: padding~~

No parameters to learn!

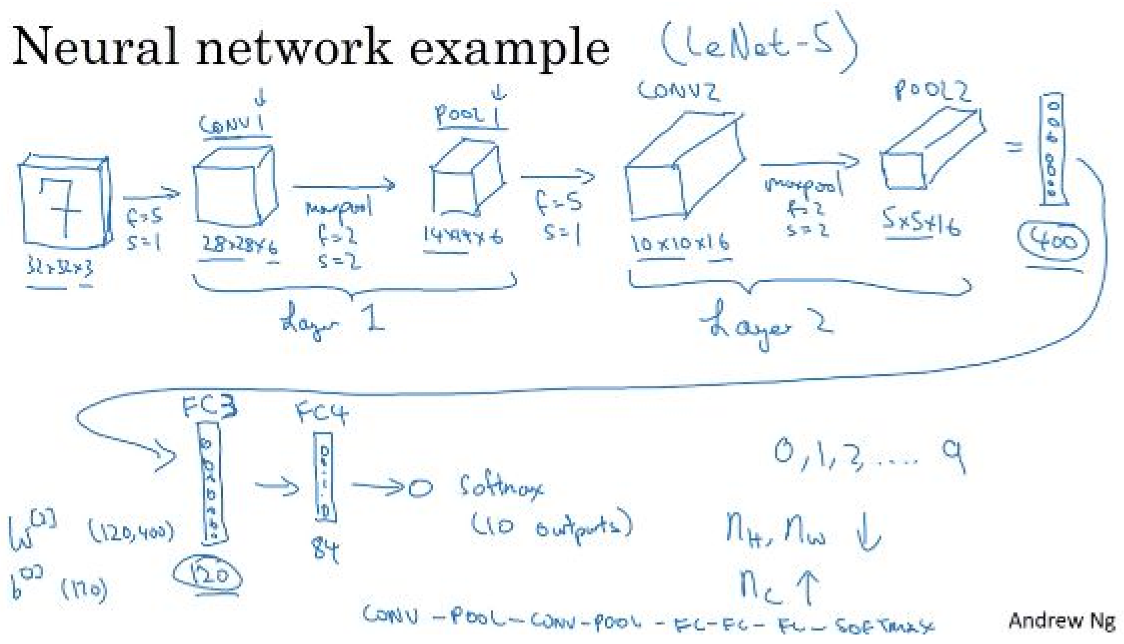
$$n_H \times n_W \times n_C$$

$$\downarrow$$

$$\left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_W - f}{s} + 1 \right\rfloor \times n_C$$

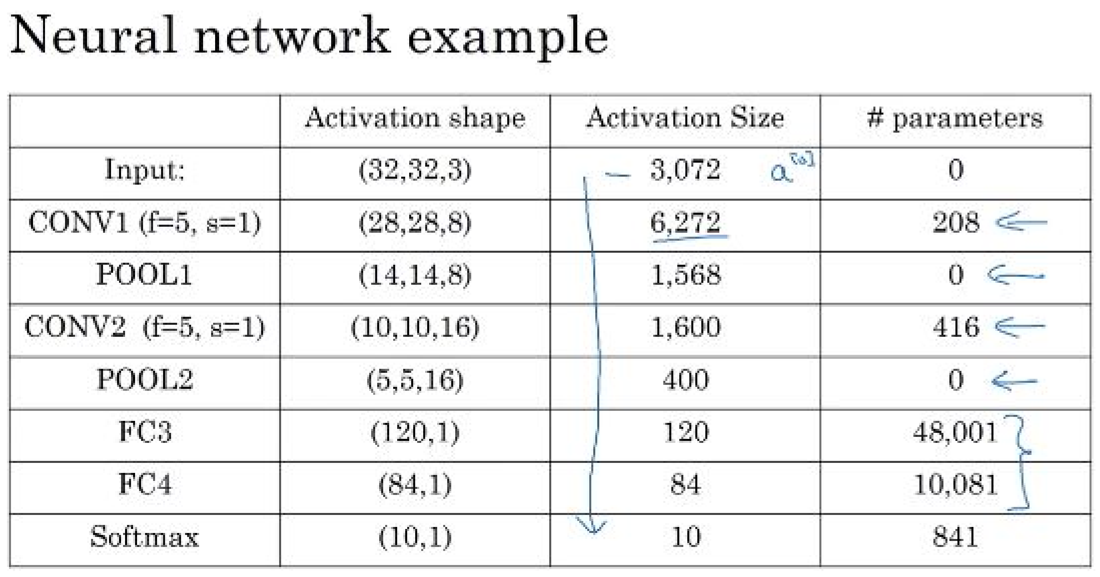
CNN Example

- Now let's look at another neural network example. Let's assume we are trying to recognize if the number of the image is one of 0-10. So we start with
  - a 32x32x3 image and we have a filter f=5 and s=1, this gives us
  - A 28x28x6 output that we call CONV1.
  - We then apply max pooling with f=2 and s=2 and the result is a POOL1 layer with output 14x14x6. We call layer 1 = CONV1 + POOL1 because POOL1 doesn't have parameters (common practice).
  - We then apply convolution with a filter f=5 and stride s=1 and the result is CONV2 with dimensions 10x10x16.
  - To that we apply max pooling with filter f=2 and stride s=2 and the result is POOL2 with dimensions 5x5x16. Layer 2 is then CONV2+POOL2.
  - We then unroll POOL2 into a 400 x 1 vector.
  - This vector is fully connected to a 120x1 vector and this is FC3. Fully connected mean all the points from the input layer are connected to all the points of the output.
  - We then have another fully connected layer, FC4 with output FC4 and dimensions 84x1.
  - Finally, we apply softmax and we get 10 outputs because we need to recognize if the number in the image is between 0-10.



Notice how the size of the image goes down and the number of channels goes up as we go through the conv net. Finally, the sequence of layers CONV-POOL-CONV-POOL-FC-FC-SOFTMAX is a pretty common occurrence in practice.

- Now let's look at how the activation size and the number of parameters go as we progress through the neural network above. Notice how the activation size decreases, pooling layers don't have any parameters, the conv layers don't have many parameters and most parameters are in the fully connected layers.

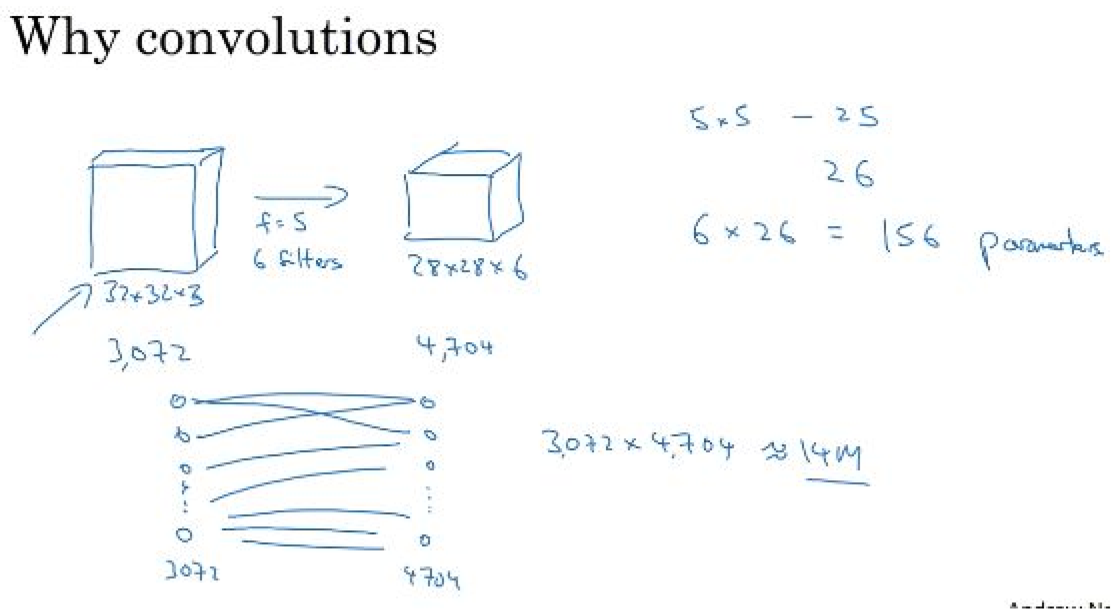


Neural network example

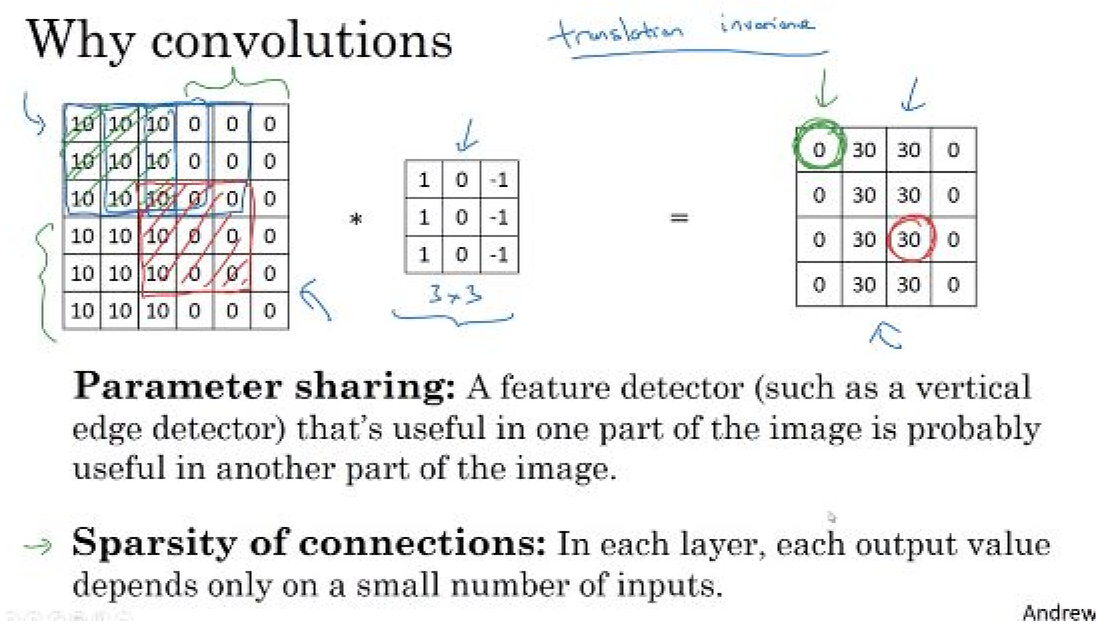
	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208 ←
POOL1	(14,14,8)	1,568	0 ←
CONV2 (f=5, s=1)	(10,10,16)	1,600	416 ←
POOL2	(5,5,16)	400	0 ←
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

Why Convolutions

- Convolutions are useful in neural networks because of two reasons: parameter sharing and sparsity of connections. Let's illustrate them with an example. Below we have an input image of 32x32x3 and we will apply a filter with f=5 and 6 filters which will result in a 28x28x6 image. If we unroll each of these images into a vector we end up with about 14 million parameters; that is a very high number of parameters to train. However, if we use convolution we have the size of the filter (5x5=25 + 1 (for the bias)) x the number of filters (6) and we end up with 156 parameters.



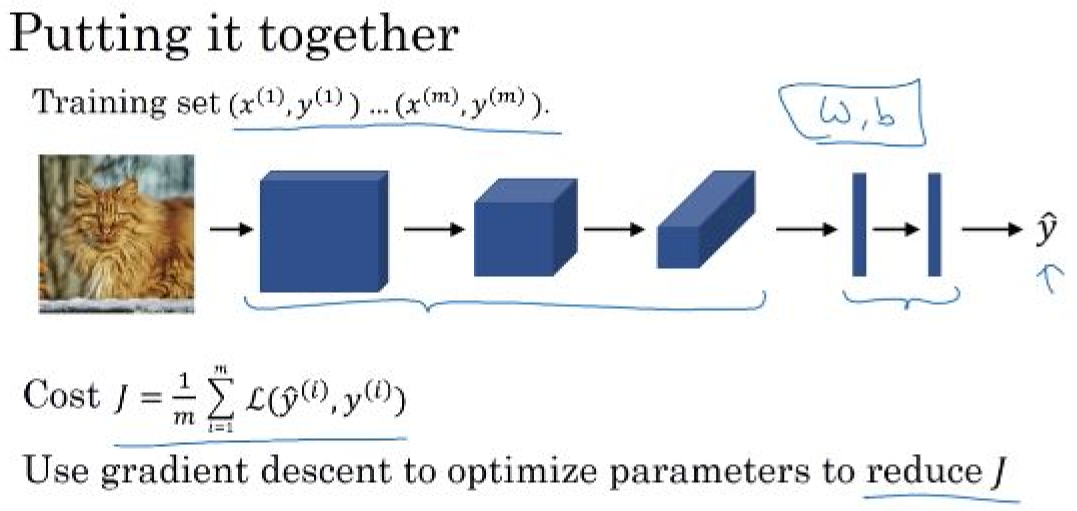
- As mentioned, convolutions are useful because they:
    - Allow us to share parameters: a feature detector is useful in all parts of an image
    - Have sparsity of connections: In each layer, an output value only depends on a small number of inputs



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.

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

- Putting everything together, given a labeled training set $(x^{(1)}, y^{(1)})\ldots(x^{(m)}, y^{(m)})$, we apply our neural network that has various parameters such as w and b. So we can compute a cost function $J = 1/m \sum_{i=1}^{m} L(yhat^{(i)}, y^{(i)})$ and optimize the parameters so we can reduce J.



Putting it together

Training set $\underline{(x^{(1)}, y^{(1)}) \ldots (x^{(m)}, y^{(m)})}$.

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

Use gradient descent to optimize parameters to $\underline{\text{reduce } J}$