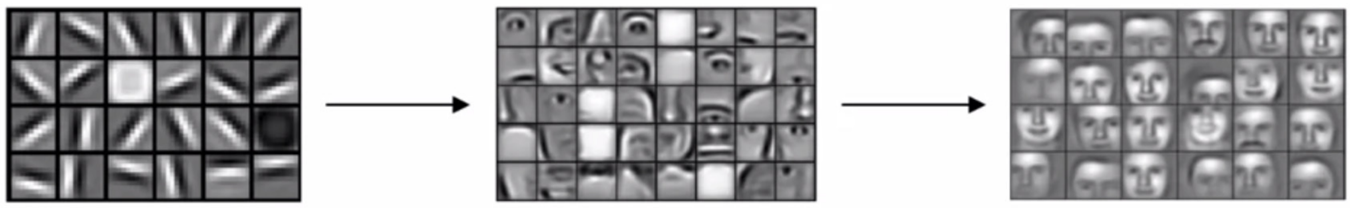
# Convlutional Operation



## Vertical Edge Detection

"convolution"

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

6×6

*

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

3×3

Filter / kernel

=

| | | | |
|---|---|---|---|
| -5 | -4 | 0 | 8 |
| -10 | | | |
| | | | |
| | | | |

4×4

Python : conv-forward
tensorflow : tf.nn.conv2d
Keras : Conv 2D

### How it works:

vertical edge detected

| | | | | | |
|---|---|---|---|---|---|
| 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 |

6×6

*

bright pixels on left

doesn't matter what is in the middle

dark pixels on right

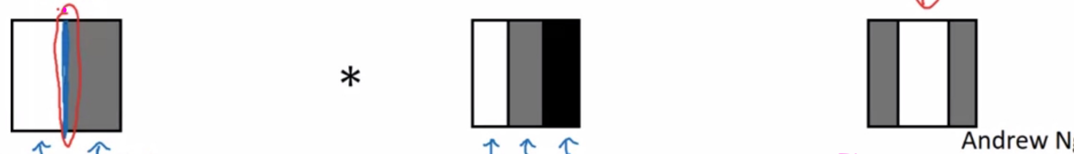
| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

3×3

=

negative if values flipped
* can take abs value

| | | | |
|---|---|---|---|
| 0 | 30 | 30 | 0 |
| 0 | 30 | 30 | 0 |
| 0 | 30 | 30 | 0 |
| 0 | 30 | 30 | 0 |

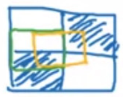



edge detected

# Horizontal Edge Detection

horizontal filter / kernel

$$
\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 30 & 10 & -10 & -30 \\ 30 & 10 & -10 & -30 \\ 0 & 0 & 0 & 0 \end{bmatrix}
$$



$6 \times 6$     $3 \times 3$     $4 \times 4$

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

Other filter options:

→ $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & 1 \end{bmatrix}$ **Sobel filter** : puts emphasis on the central row / pixel

→ $\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$ **Scharr filter** : puts emphasis on the central row / pixel

* flip to get horizontal edge detection

→ $\begin{bmatrix} W_1 & W_2 & W_3 \\ W_4 & W_5 & W_6 \\ W_7 & W_8 & W_9 \end{bmatrix}$ Learn these as parameters such that this gives us a good edge detector

- Learning is even more robust
- helps learn these features its trying to detect
- can learn edges at angles too

# Padding

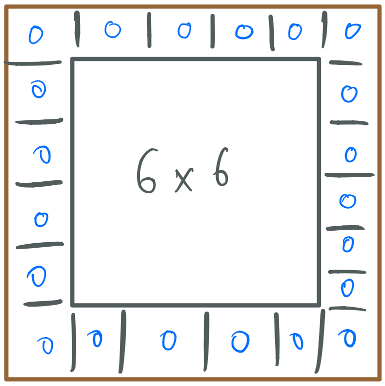
$6 \times 6$ mat $\ast$ $3 \times 3$ mat $= 4 \times 4$ mat

convolutional

problems

→ shrinking output

→ throwing away info from edges




$\ast$ $3 \times 3$ mat $= 6 \times 6$ mat

$$(n+2p-f+1) \times (n+2p-f+1)$$

$8 \times 8$ mat padded

- pad with zeros
- $p =$ padding amount $= 1$ (in the example)

## Valid & Same convolutions

Valid → no padding applied

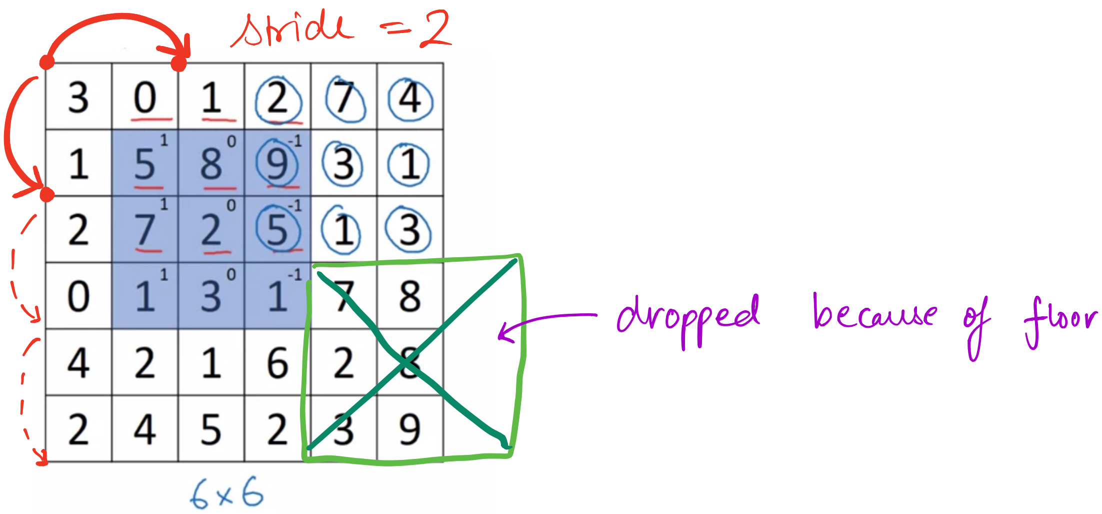
$$n \times n \ast f \times f = (n-f+1) \times (n-f+1)$$

Same → pad so the output is same as the input size

$$(n+2p-f+1) \times (n+2p-f+1) \ast f \times f = (n+2p-f+1) \times (n+2p-f+1)$$

☆ $f$ is usually odd

# Strided Convolutions

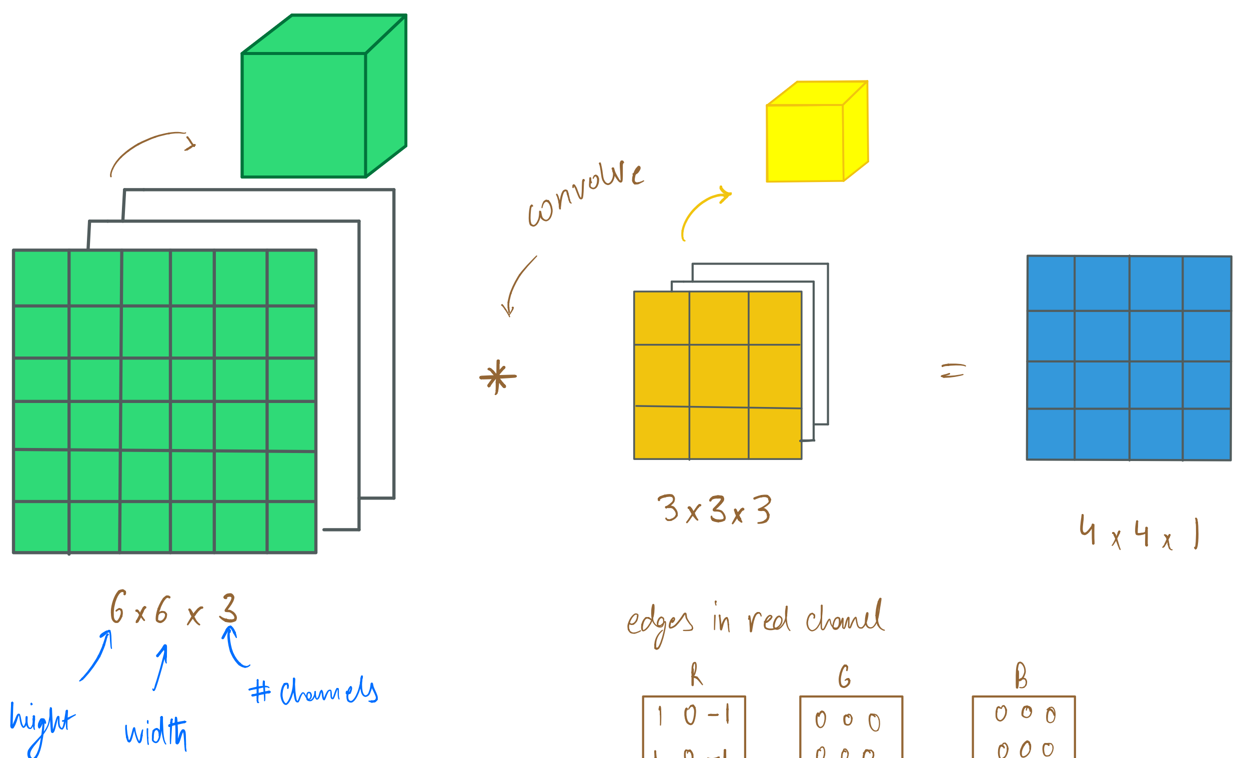



dimensions –

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

padding $p$     stride $s$

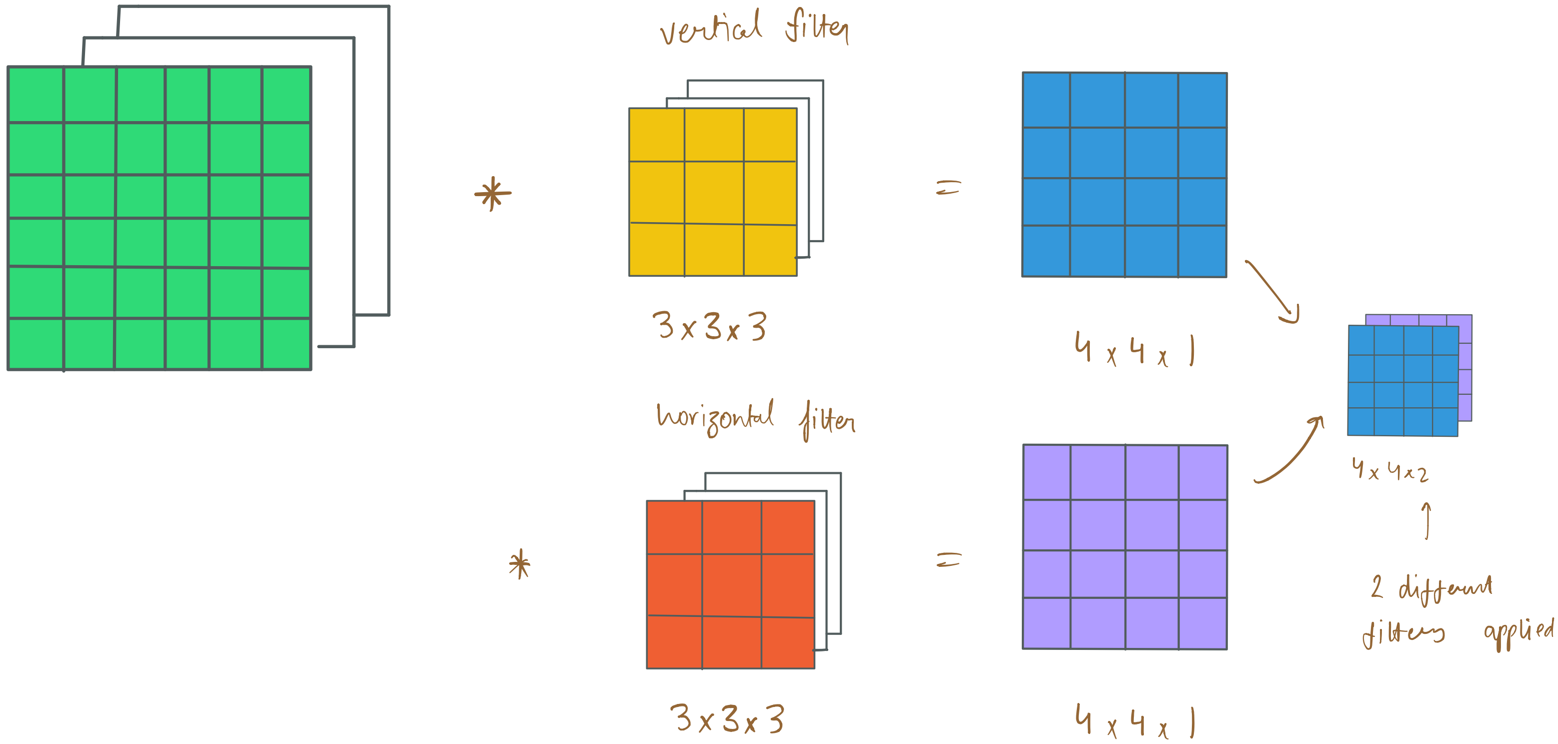
# Convolutions Over Volume




edges in red channel

$$\begin{array}{ccc} R & G & B \\ \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

# Multiple Filters



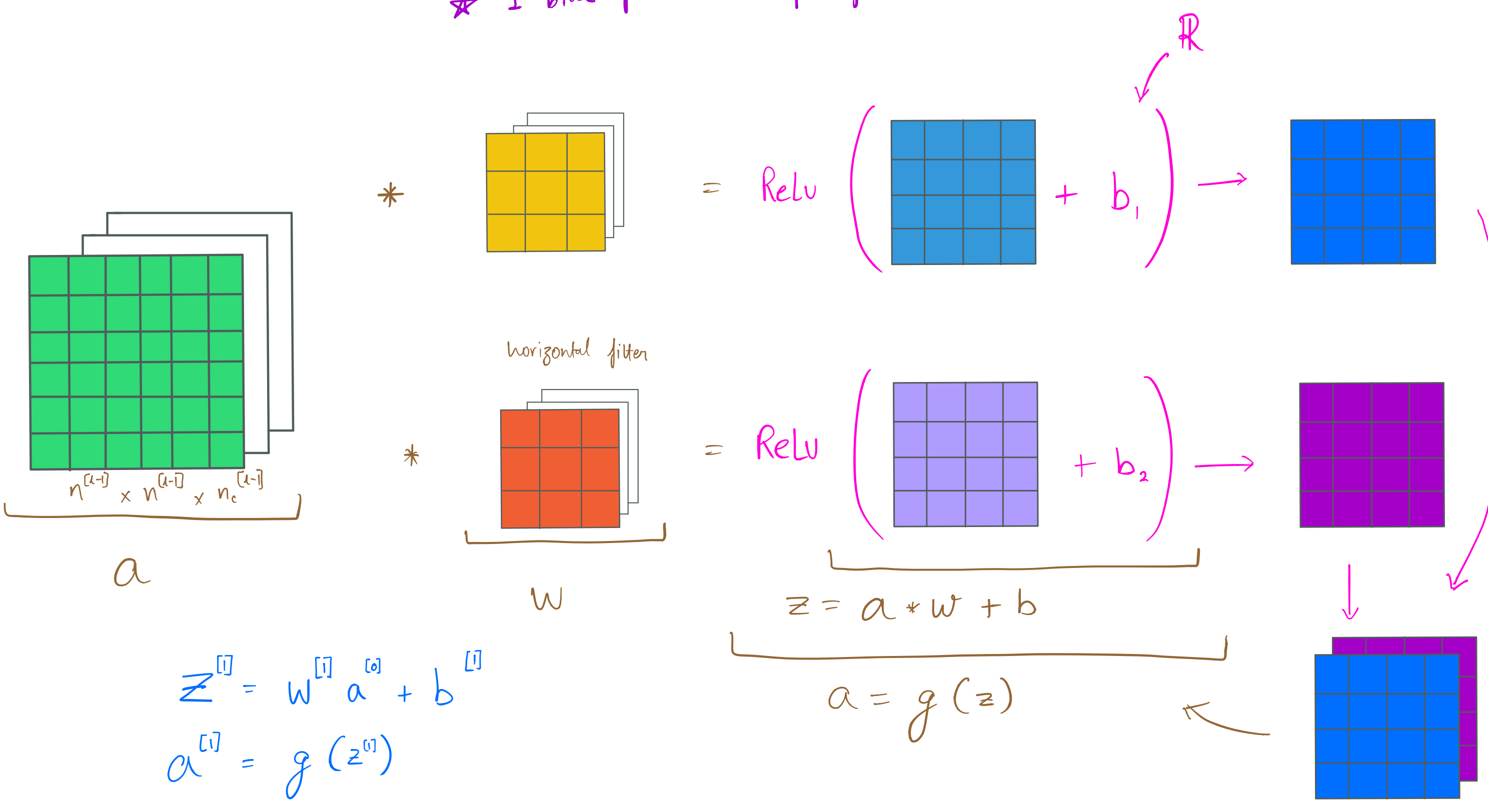

dimension:

$$\underbrace{n \times n \times n_c}_{} * f \times f \times n_c \rightarrow \underbrace{n-f+1}_{4} \times \underbrace{n-f+1}_{4} \times n_c$$

$4 \times 4$

$n_c$ ← # filters depth

☆ 1 bias parameter per filter



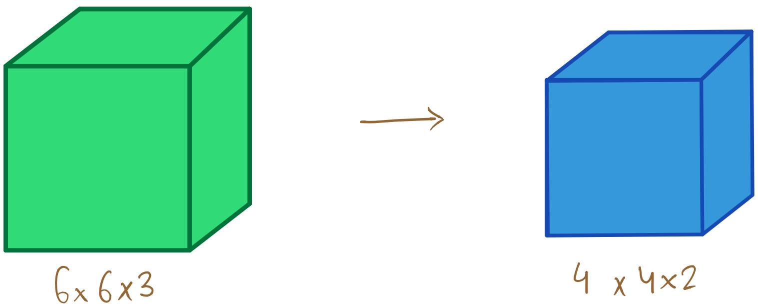

$$z^{[l]} = w^{[l]} a^{[0]} + b^{[l]}$$

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

derivatives – $dA \mathrel{+}= \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw}$

$$dW_c \mathrel{+}= \sum_{h=0}^{n_H} \sum_{h=0}^{n_W} a_{slice} \times dZ_{hw}$$

$$db = \sum \sum dZ_{hw}$$




$a^{[0]} \longrightarrow a^{[1]}$

## Notation

If layer 1 is a convolution layer –

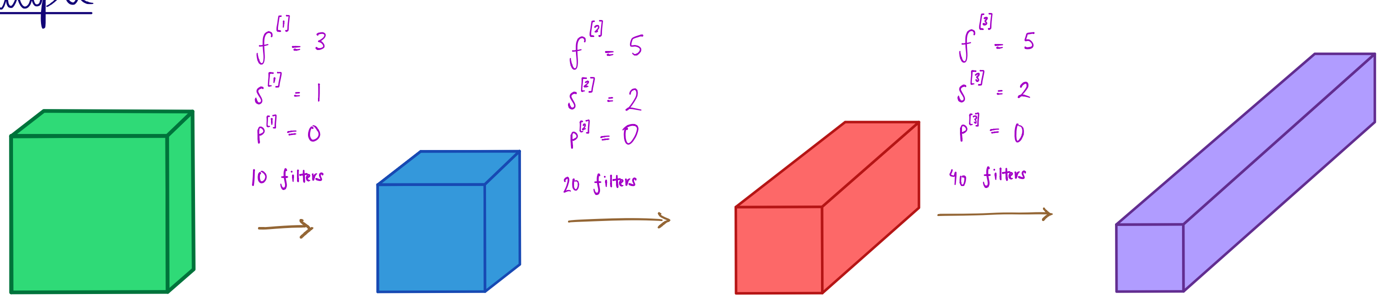
- $f^{[l]}$ = filter size
- $p^{[l]}$ = padding
- $s^{[l]}$ = stride
- input – $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$
- output – $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- $n^{[l]} = \left\lfloor \dfrac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$
- filter – $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

- activations – $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- weights – $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$

  depth from last layer ⤴ ⤶ # of filters in layer $l$

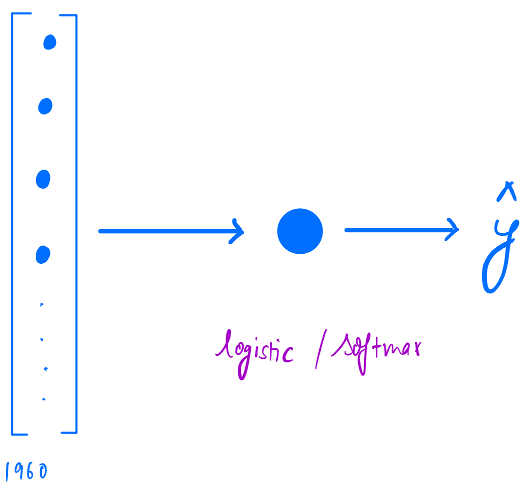
- bias – $n_c^{[l]} \rightarrow 1, 1, 1, n_c^{[l]}$

Example




| $X / a^{[0]}$ | $a^{[1]}$ | $a^{[2]}$ | $a^{[3]}$ |
|---|---|---|---|
| $39 \times 39 \times 3$ | $37 \times 37 \times 10$ | $17 \times 17 \times 20$ | $7 \times 7 \times 40$ |
| $n_H^{[0]} = n_W^{[0]} = 39$ | $n_H^{[1]} = n_W^{[1]} = 37$ | $n_H^{[2]} = n_W^{[2]} = 17$ | $n_H^{[3]} = n_W^{[3]} = 7$ |
| $n_c^{[0]} = 3$ | $n_c^{[1]} = 10$ | $n_c^{[2]} = 20$ | $n_c^{[3]} = 40$ |

vectorizing $a^{[3]}$ –




Types of layers in a convolutional network –

→ convolution (CONV)

→ pooling (POOL)

→ fully connected (FC)

# Pooling Layers

## Max Pooling

hyperparameters

$f = 2$

$s = 2$

| | | | |
|---|---|---|---|
| 1 | 3 | 2 | 1 |
| 2 | 9 | 1 | 1 |
| 1 | 3 | 2 | 3 |
| 5 | 6 | 1 | 2 |

$\longrightarrow$

| | |
|---|---|
| 9 | 2 |
| 6 | 3 |

getting the max from each color

- Some set of features or activations from previous layer, then max pooling detects a particular feature and preserves it due to the high number

→ no parameters, only hyperparameters ⇒ fixed computation

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 |

$5 \times 5$

$\longrightarrow$

| | | |
|---|---|---|
| 9 | 9 | 5 |
| 9 | 9 | 5 |
| 8 | 6 | 9 |

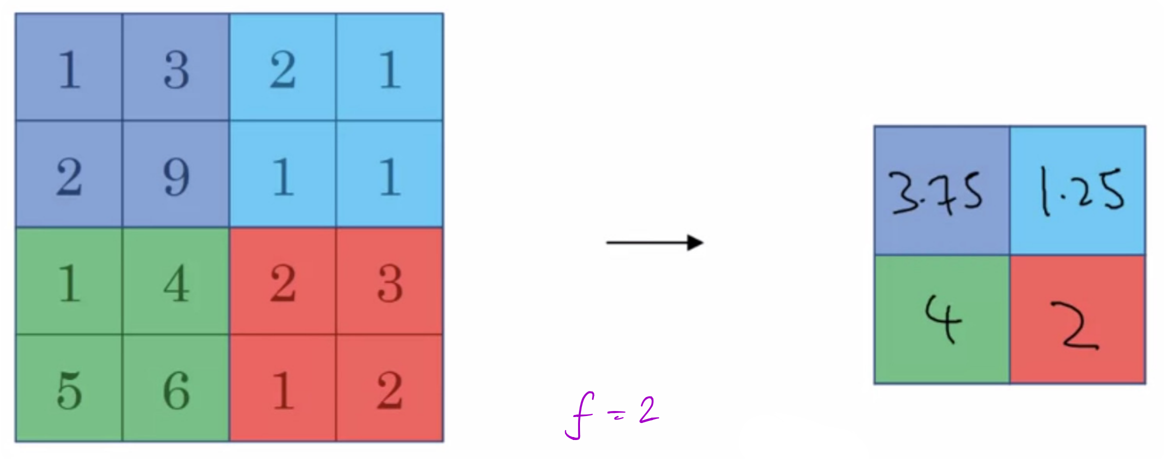
$3 \times 3$

$f = 3$

$s = 1$

$$\frac{n + 2p - f}{s} + 1 = 3$$

# Average Pooling



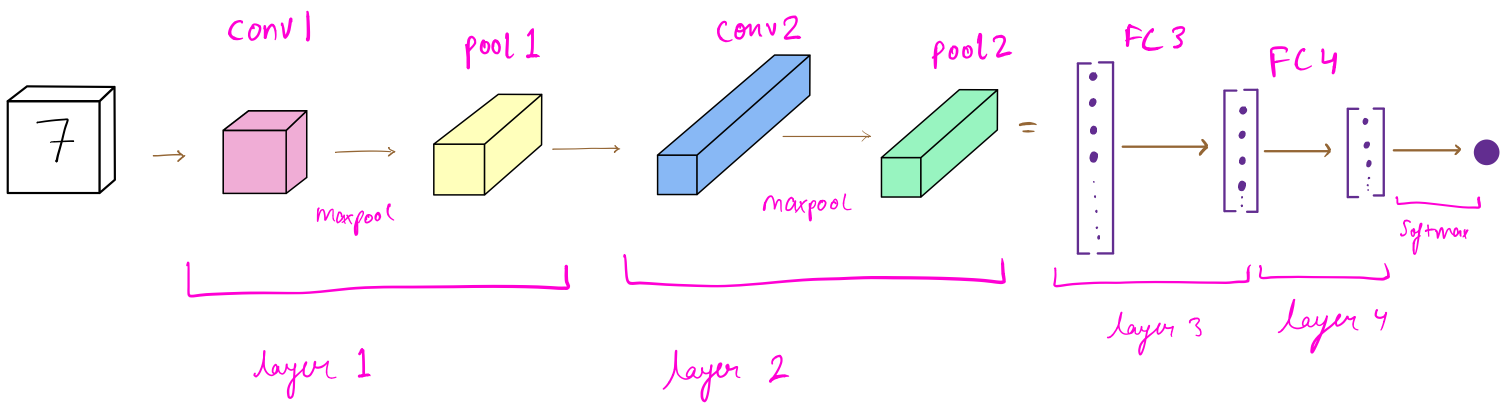

$f = 2$

$s = 2$

$7 \times 7 \times 1000 \longrightarrow 1 \times 1 \times 1000$

→ helps make the size of the matrix smaller by taking the averages

# Example of a full Convolutional Network




| | Activation shape | Activation Size | # parameters |
|---|---|---|---|
| Input: | (32,32,3) | 3,072 | 0 |
| CONV1 (f=5, s=1) | (28,28,8) | 6,272 | 608 |
| POOL1 | (14,14,8) | 1,568 | 0 |
| CONV2 (f=5, s=1) | (10,10,16) | 1,600 | 3216 |
| POOL2 | (5,5,16) | 400 | 0 |
| FC3 | (120,1) | 120 | 48120 |
| FC4 | (84,1) | 84 | 10164 |
| Softmax | (10,1) | 10 | 850 |

# Why Convolutions

CONV nets have comparatively way less parameters

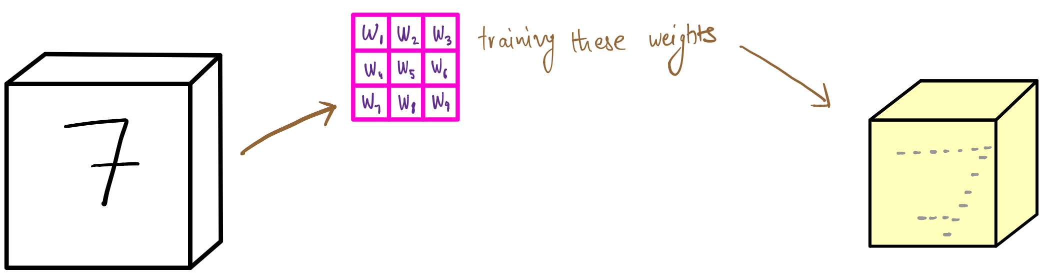
↪ Parameter Sharing - A feature detector like vertical / horizontal edge detectors are useful in all parts of the image

↪ Sparsity of connections - in each layer, each output value depends only on a small number of inputs

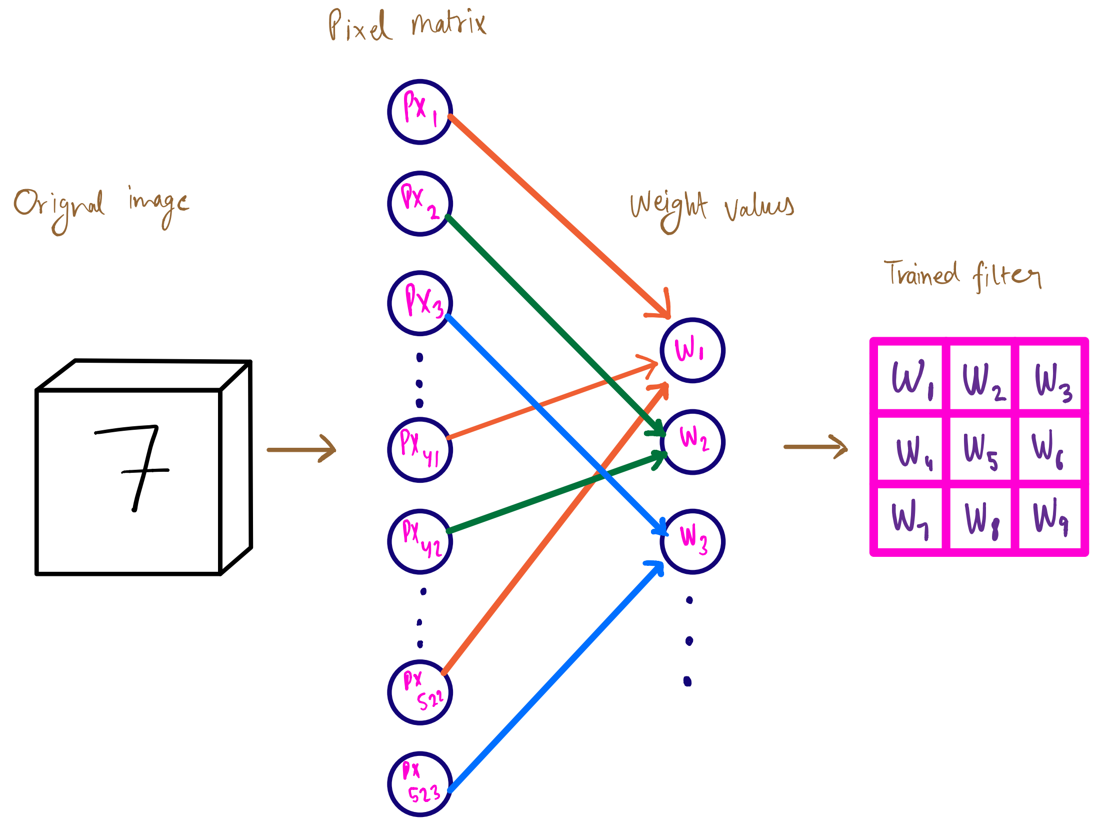
As we take a picture suppose RGB, and we apply a filter/kernel to it, we are able to make that picture go through a filter like edge detections and leave out other features of the image. The image we get is particular to that filter. As we apply more filters/kernels to the image, we can stack these output images and we get images that have certain filters applied to them and these images tend to be smaller in width and height as compared to the actual images.

We can view this as a neural network if we try to stack all the pixels of the original picture and connect it to the weights which are the values of the pixels in the filter. The concept of convolutional neural networks is that we connect a lot of these pixels values / input units to the weights / hidden units but not all of them to each one of the hidden units. The missing connections or the less connections help us to end up with an easier network with less parameters than a traditional neural network where all input units are connected to every one of the hidden units.

As use this filter to get another image that we feed again to another convolutional network, this helps us separate or individually feed these features to further layers which eventually help us identify what the image is. We train the parameters, i.e the pixel values of the filter, to make new filters that help us detect features important in identifying the image. As we apply the filter we get a part or a feature of the image important for our classification.




What pooling does is it helps us to kind of average out and make the size of the height and width smaller to prevent having a large number of parameters. It is like a control check on the network and helps us keep the number of parameters in control.


Orignal image
7
Pixel matrix
Px1
Px2
Px3
Px41
Px42
Px522
Px523
Weight values
W1
W2
W3
Trained filter
W1 W2 W3
W4 W5 W6
W7 W8 W9