

CNN

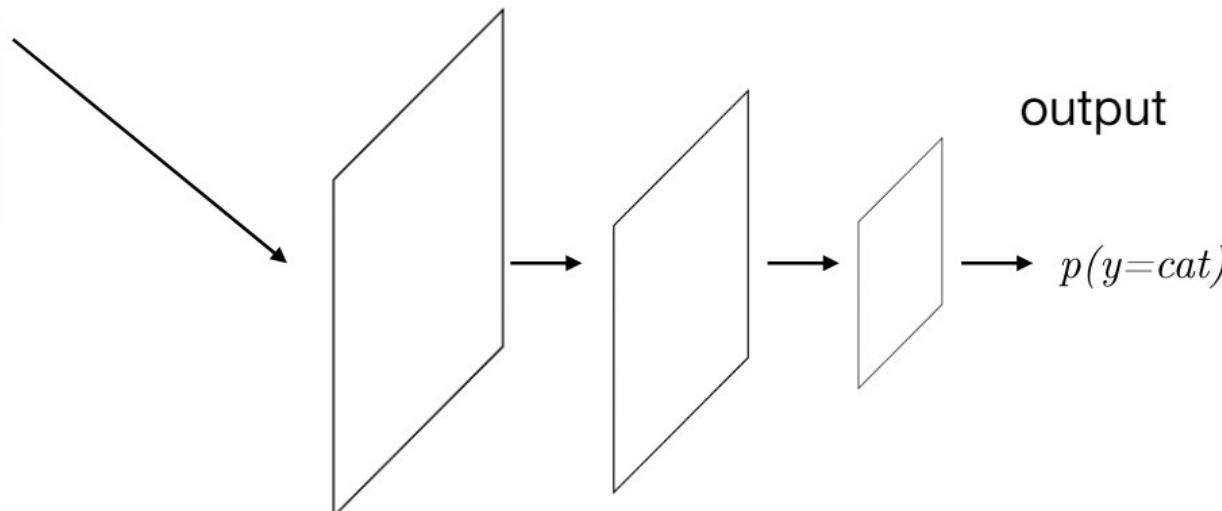
# CNN for image classification



Image Source:  
[twitter.com%2Fcats&psig=AOvVaw30\\_o-PCM-K21DiMAJQimQ4&ust=1553887775741551](https://twitter.com%2Fcats&psig=AOvVaw30_o-PCM-K21DiMAJQimQ4&ust=1553887775741551)



Image Source: <https://www.pinterest.com/pin/244742560974520446>



# Object detection



Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 779-788).

# Object segmentation

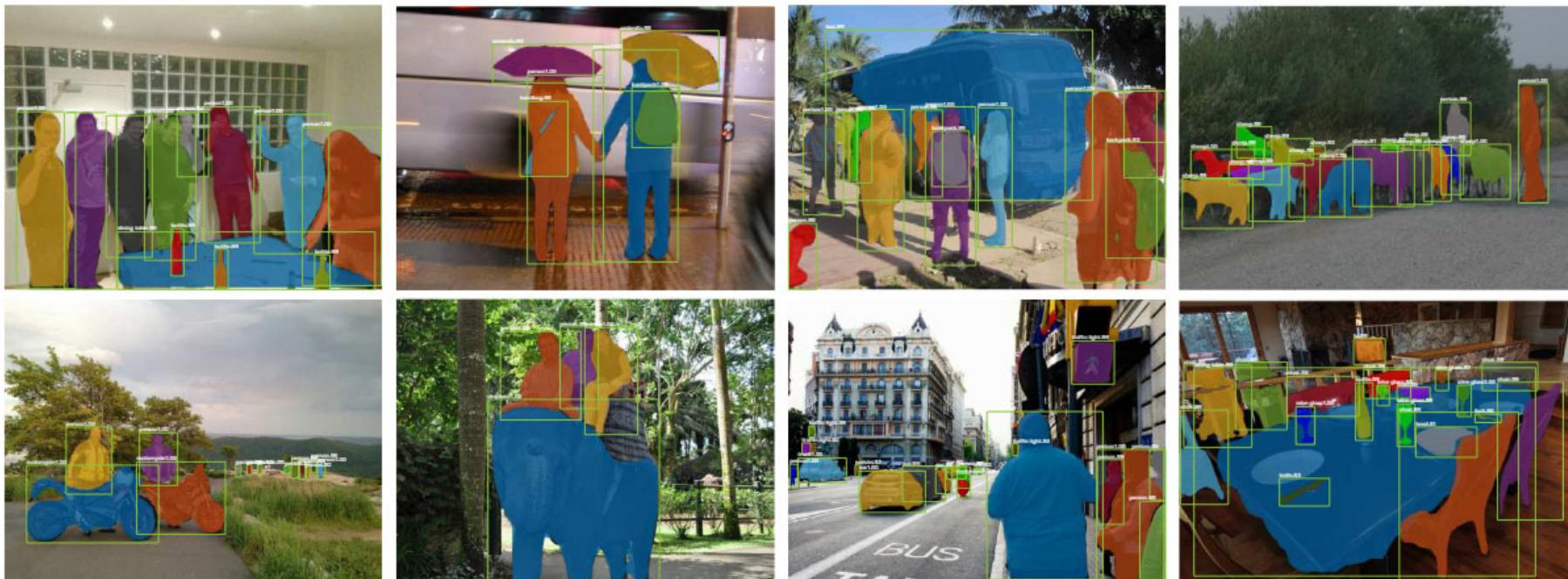
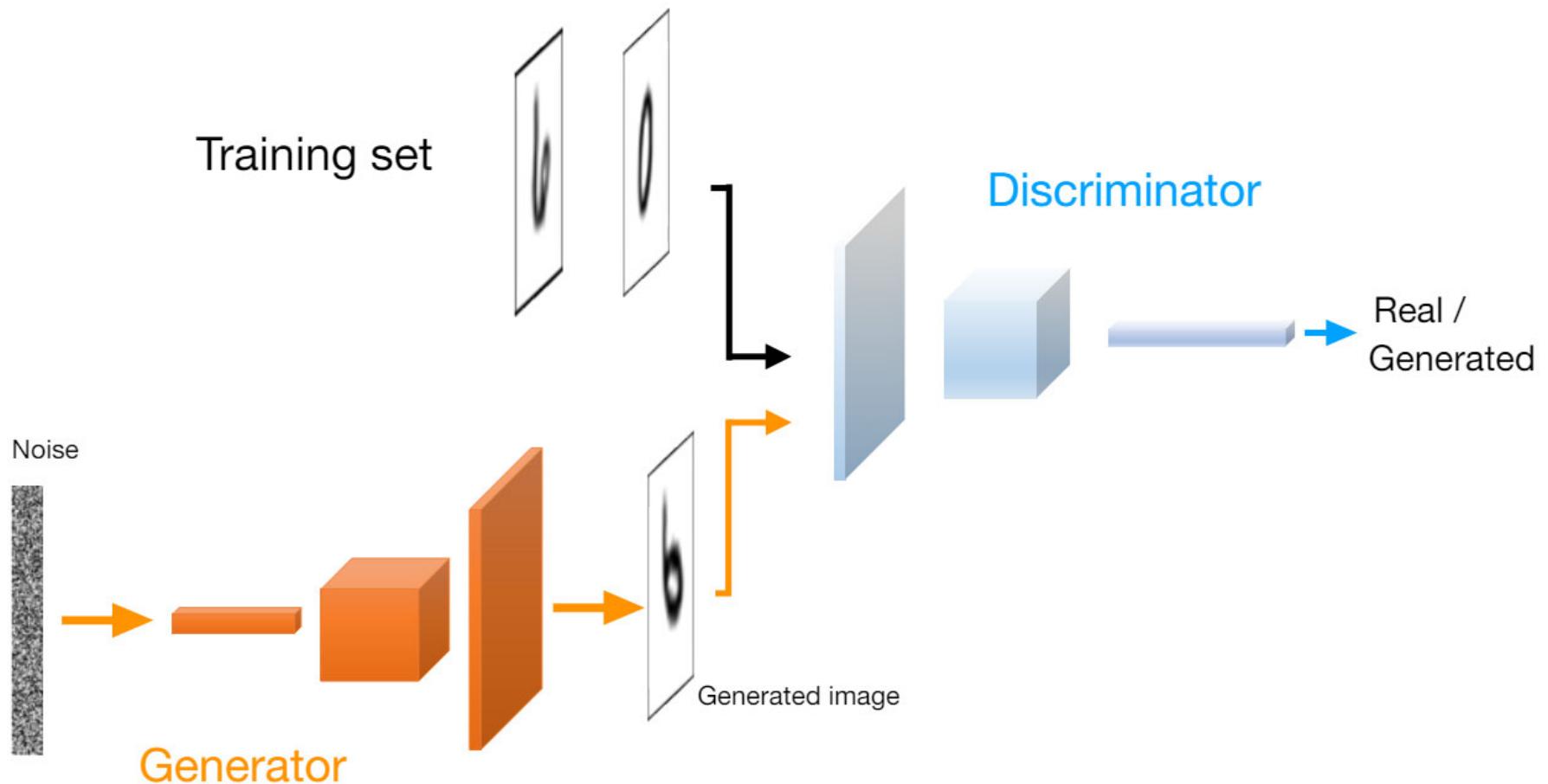


Figure 2. **Mask R-CNN** results on the COCO test set. These results are based on ResNet-101 [15], achieving a *mask AP* of 35.7 and running at 5 fps. Masks are shown in color, and bounding box, category, and confidences are also shown.

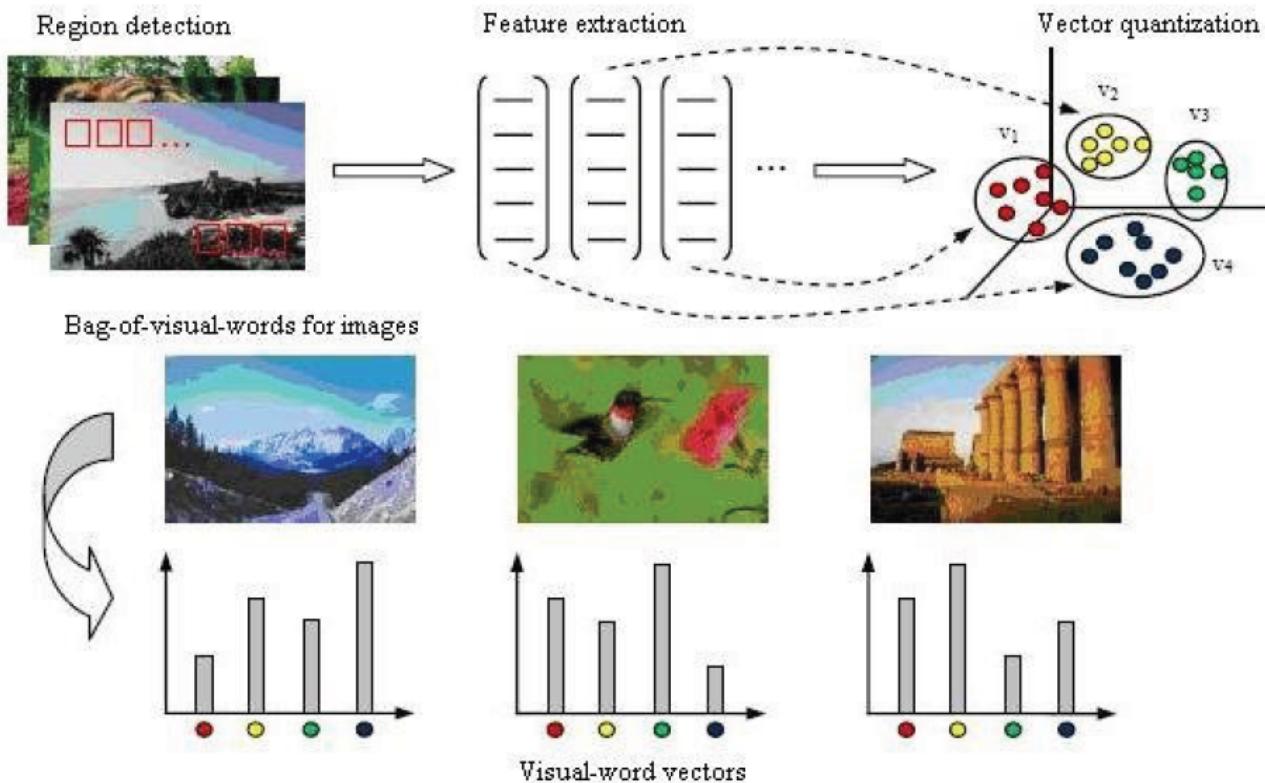
He, Kaiming, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. "Mask R-CNN." In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2961-2969. 2017.

# Image generation



# History of Computer Vision

## Before Deep Learning: Hand-Crafted Features



# History of Computer Vision

## Effect of Deep Learning: Comparison between Deep Learning and Traditional Models

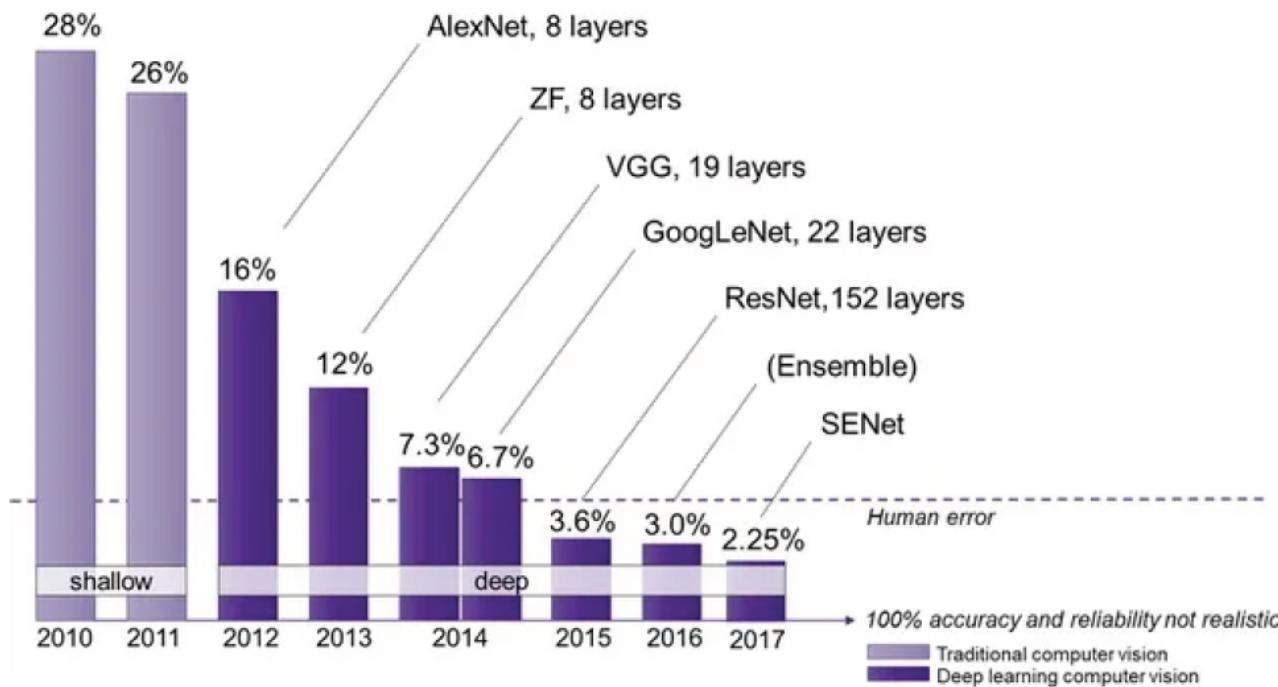


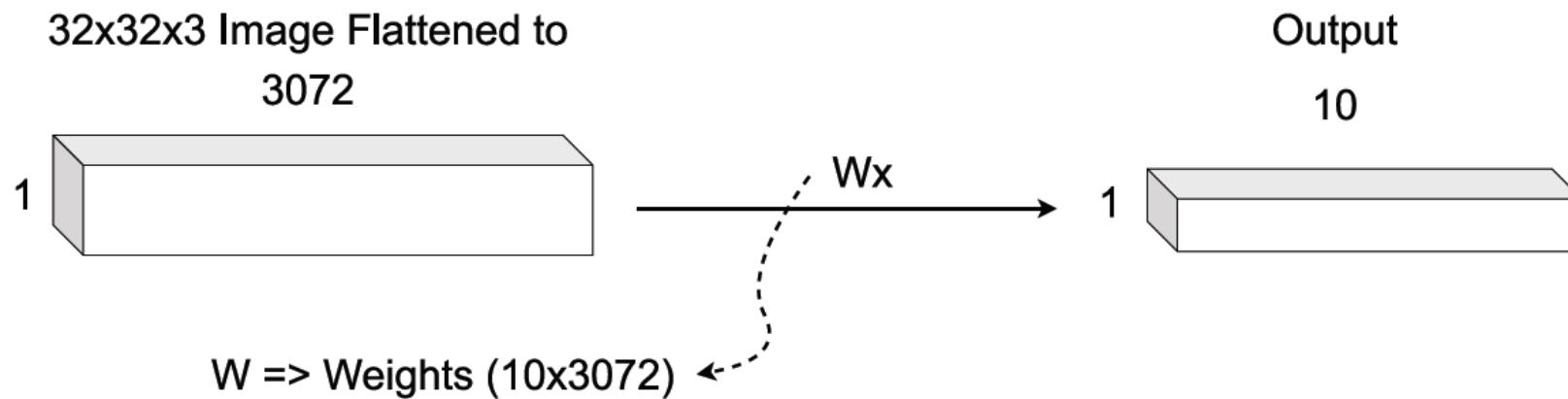
Figure: ImageNet Competition Results Over Time, Source

# Traditional approaches

b) Preprocess images (centering, cropping, etc.)

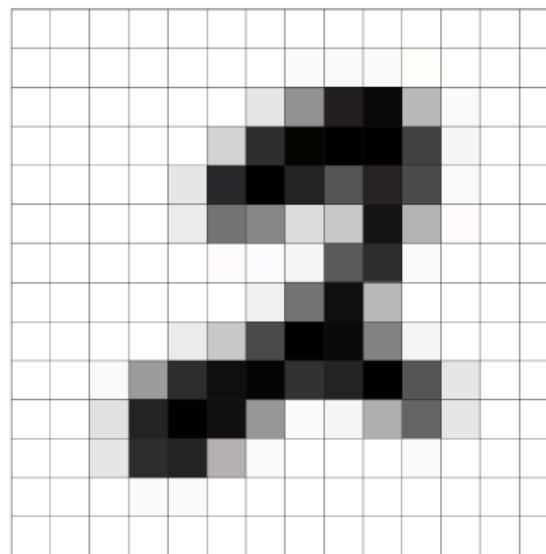


# Using Fully Connected Artichecture



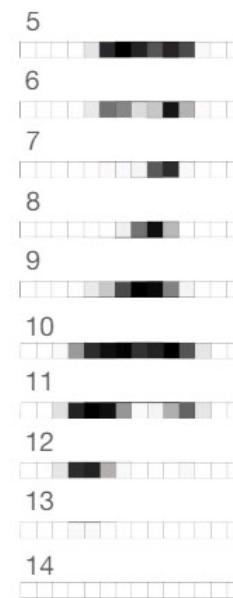
- What is the Problem with FC ?

# Multi layer perceptron on image data



14x14-dimensional  
image

# Concatenate the rows



get a 196-dimensional row vector

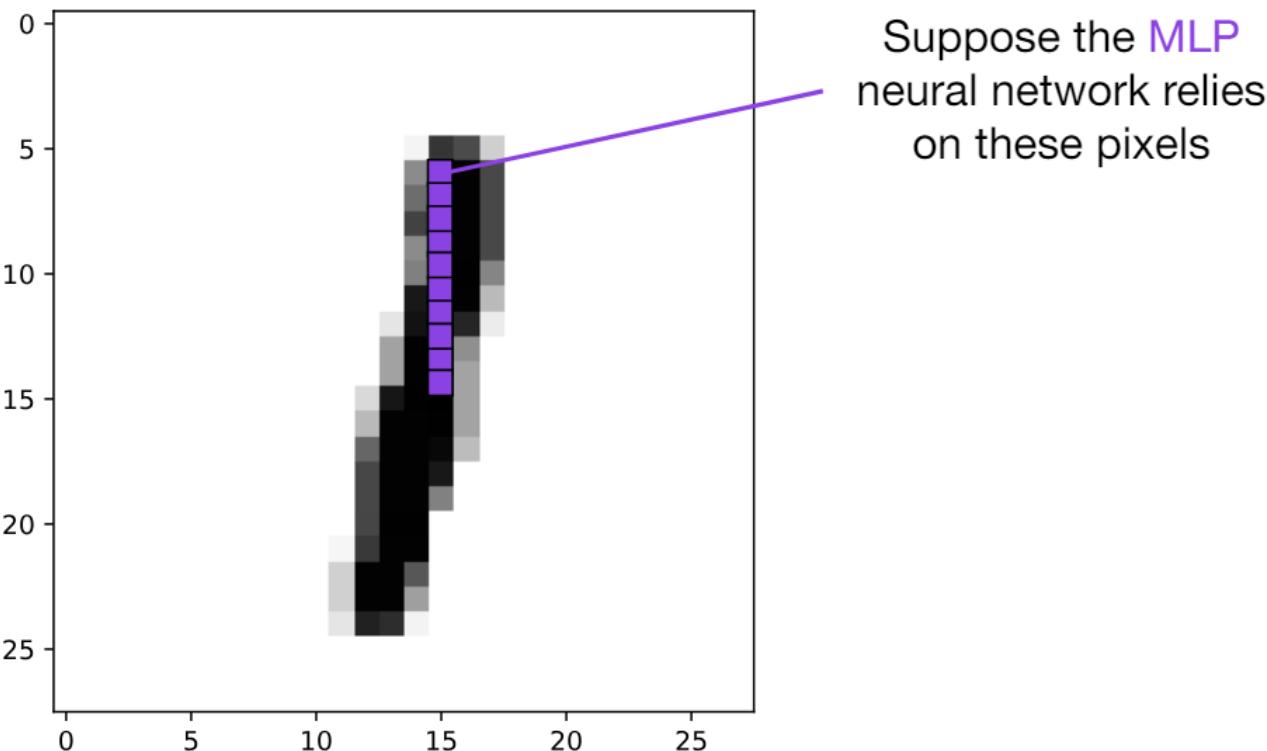


$14 \times 14 \text{ pixels} = 196 \text{ pixels}$

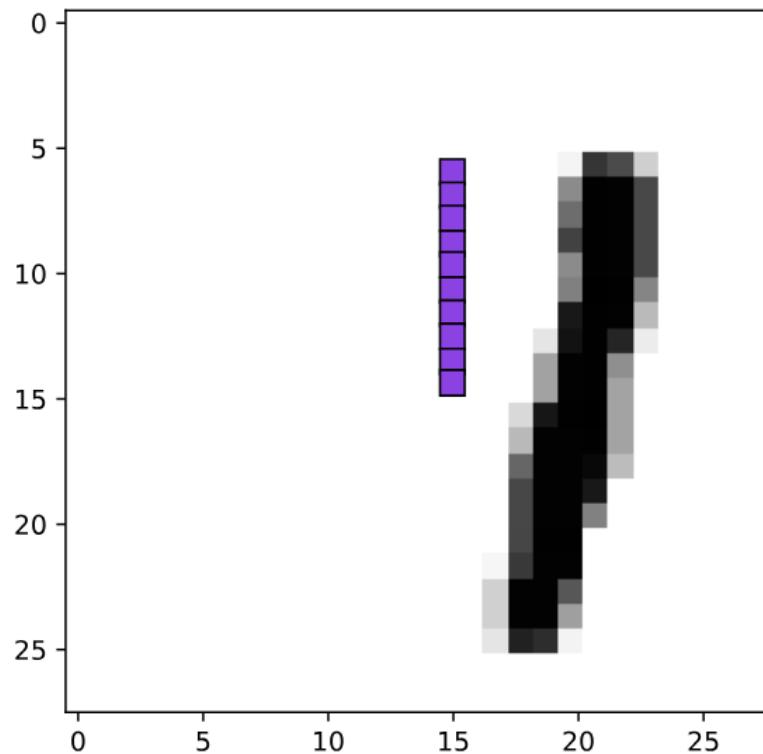
# Training examples



# Problem with MLP for images



# Problem with MLP for images



An **MLP** will not  
recognize the digit  
if it is in slightly  
different position

# Why Image Classification is Hard

Different lighting, contrast, viewpoints, etc.

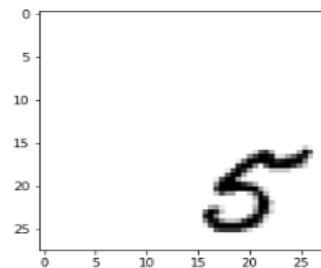
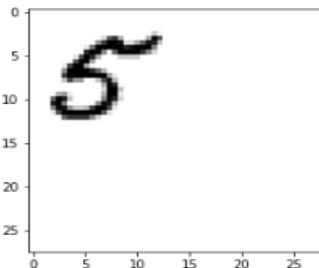


Image Source:  
[twitter.com%2Fcats&psig=AOvVaw30\\_o-PCM-K21DiMAJQimQ4&ust=1553887775741551](https://twitter.com%2Fcats&psig=AOvVaw30_o-PCM-K21DiMAJQimQ4&ust=1553887775741551)



Image Source: [https://www.123rf.com/photo\\_76714328\\_side-view-of-tabby-cat-face-over-white.html](https://www.123rf.com/photo_76714328_side-view-of-tabby-cat-face-over-white.html)

Or even simple translation

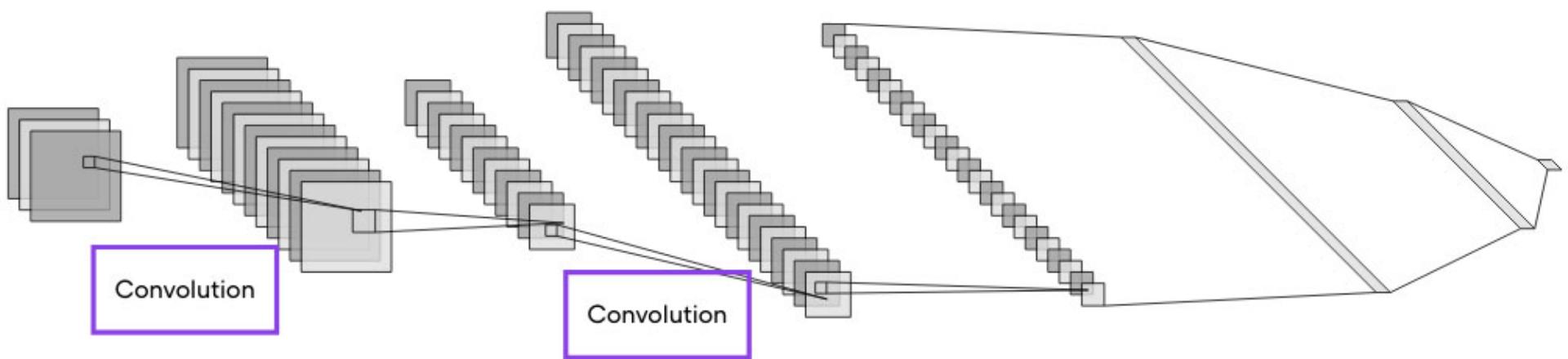


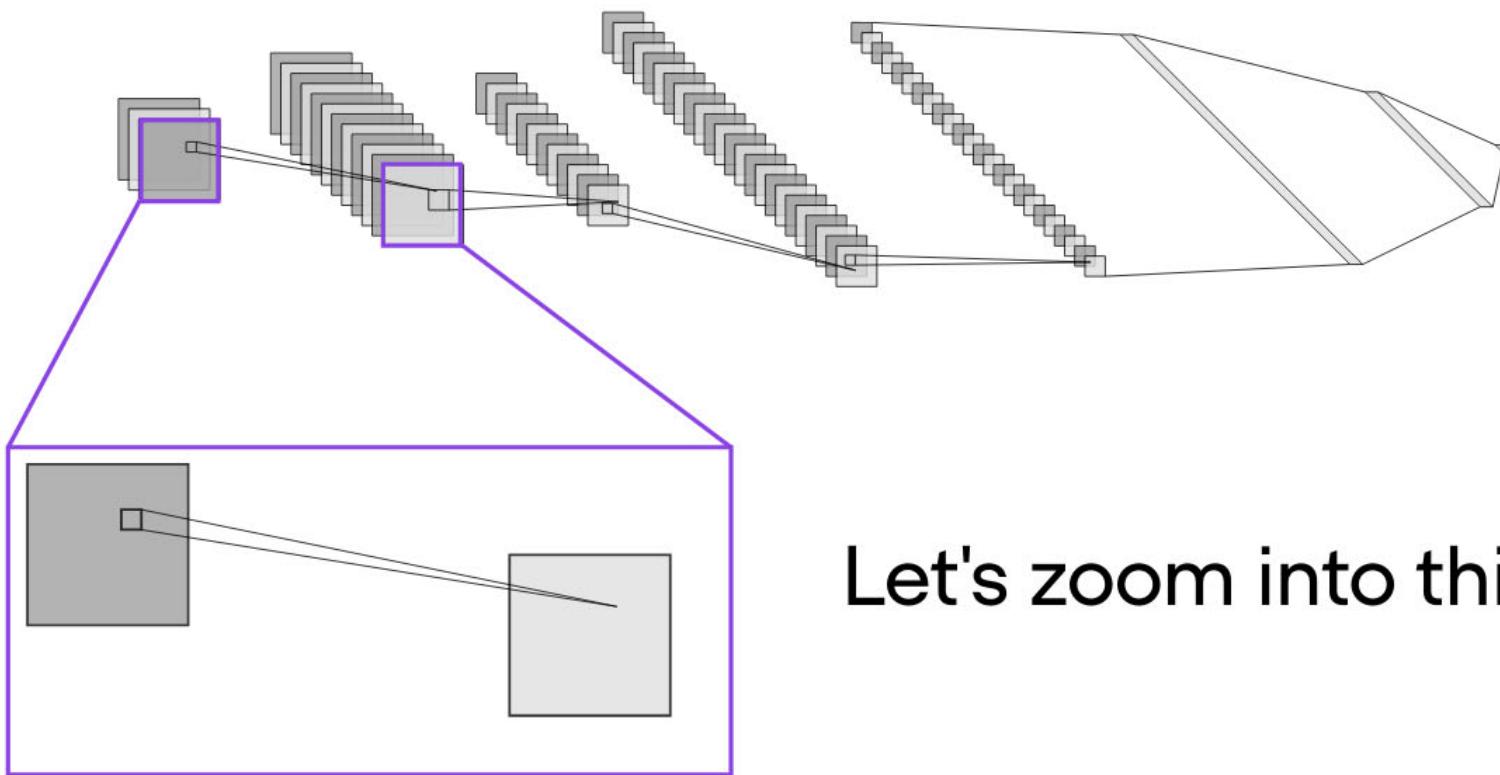
This is hard for traditional methods like multi-layer perceptrons, because the prediction is basically based on a sum of pixel intensities

# Main concept behind convolutional neural networks

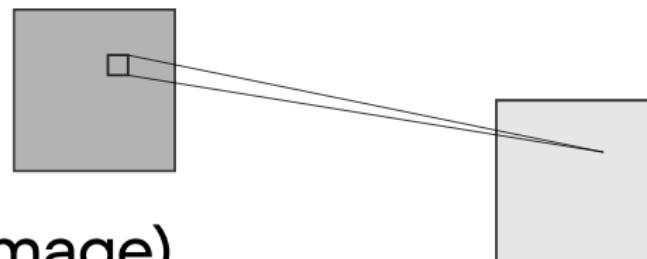
- Sparse-connectivity: A single element in the feature map is connected to only a small patch of pixels. (This is very different from connecting to the whole input image, in the case of multilayer perceptrons.)
- Parameter-sharing: The same weights are used for different patches of the input image.
- Many layers: Combining extracted local patterns to global patterns

# Looking at convolutional layers in more detail



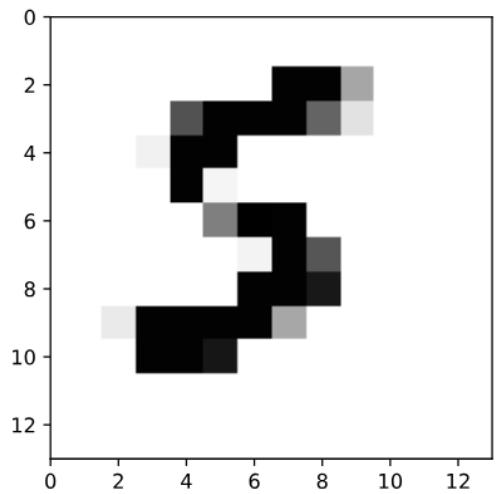


Let's zoom into this part

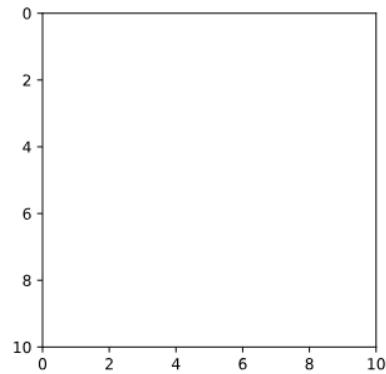


Input (image)

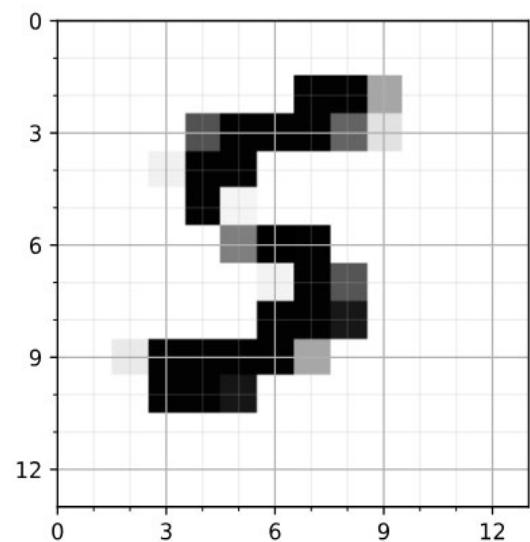
Feature map



Input (image)

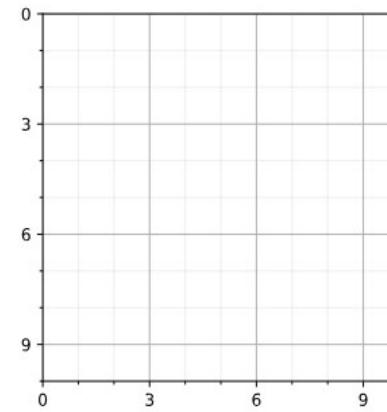
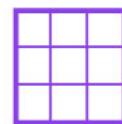


Feature map

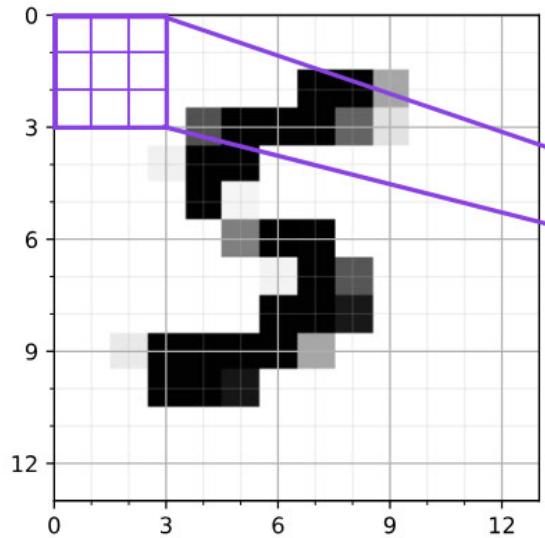


Input (image)

3x3 feature detector (kernel, filter)

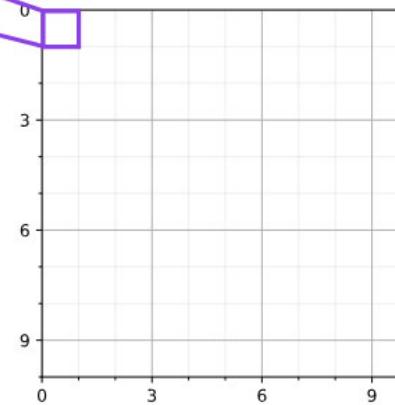


Feature map

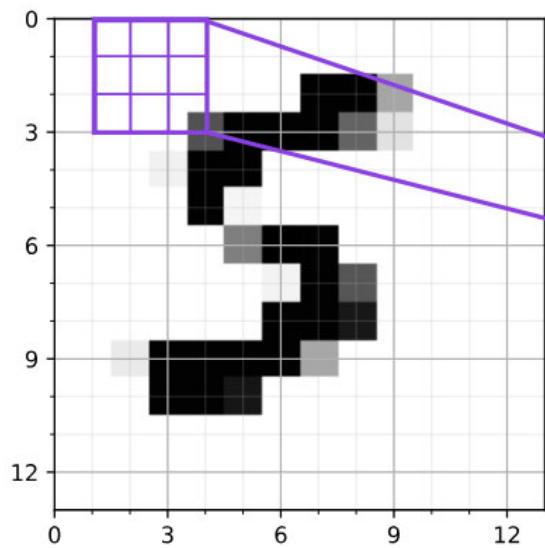


Input (image)

Slide feature detector over inputs

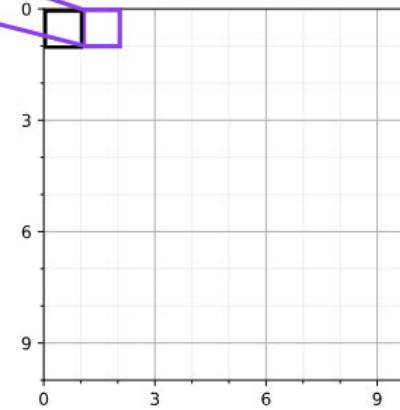


Feature map



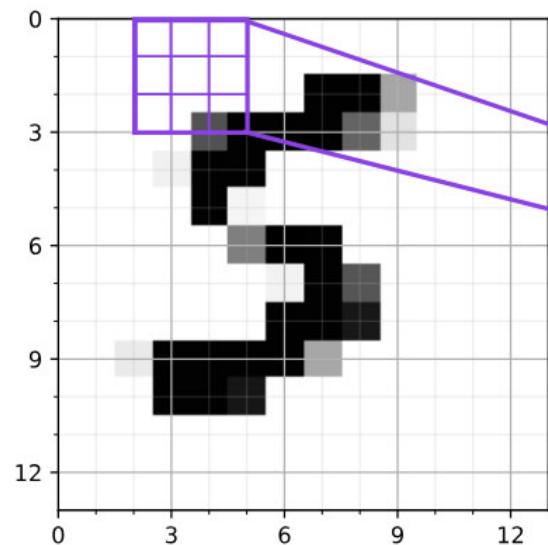
Input (image)

Slide feature detector over inputs

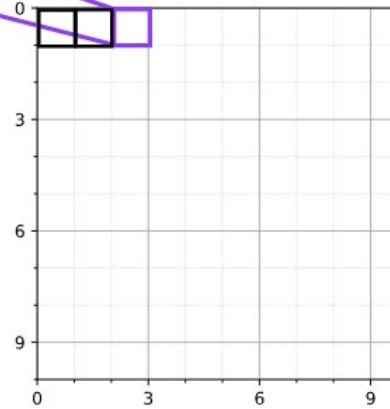


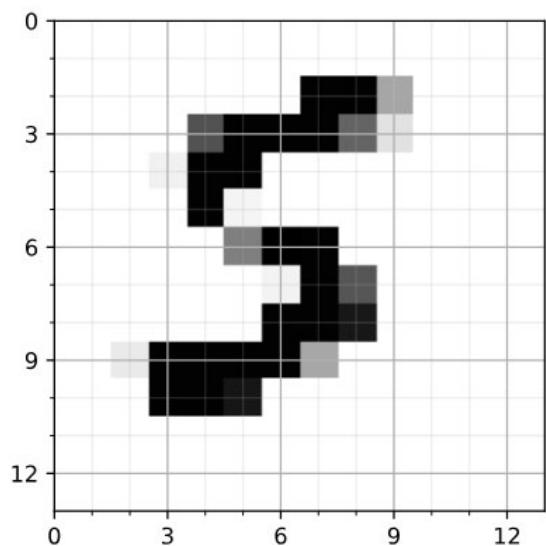
Feature map

Slide feature detector over inputs



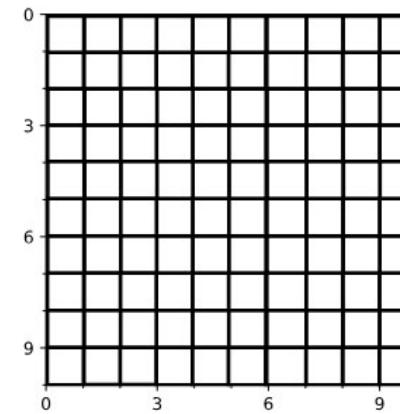
Input (image)



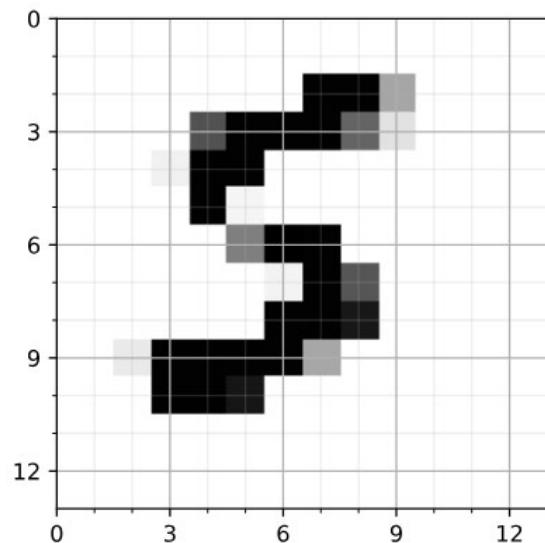


Input (image)

Slide feature detector over inputs

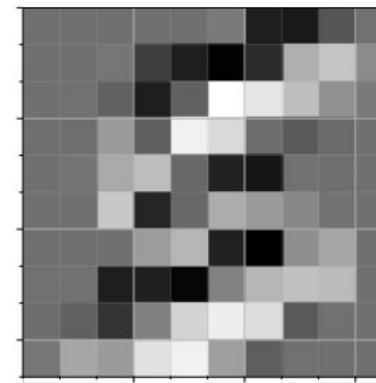


Feature map



Input (image)

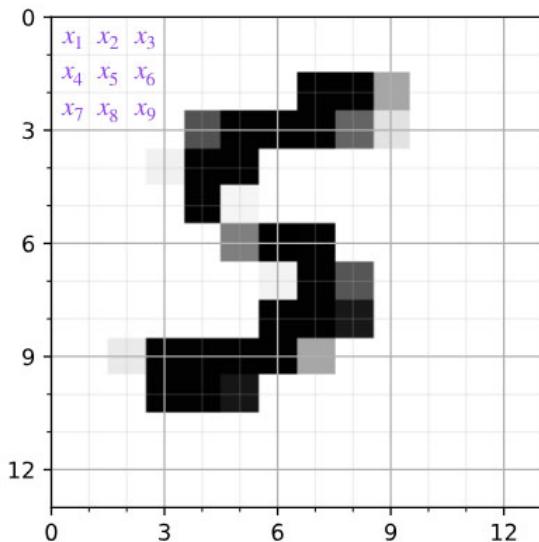
Slide feature detector over inputs



Feature map

# What is happening during this operation?

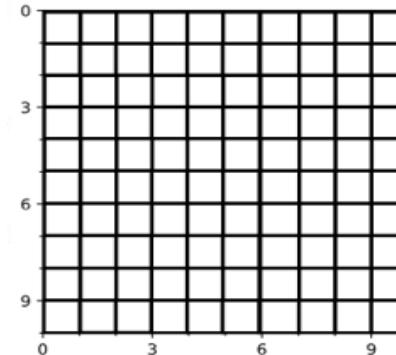
$w_1$	$w_2$	$w_3$
$w_4$	$w_5$	$w_6$
$w_7$	$w_8$	$w_9$



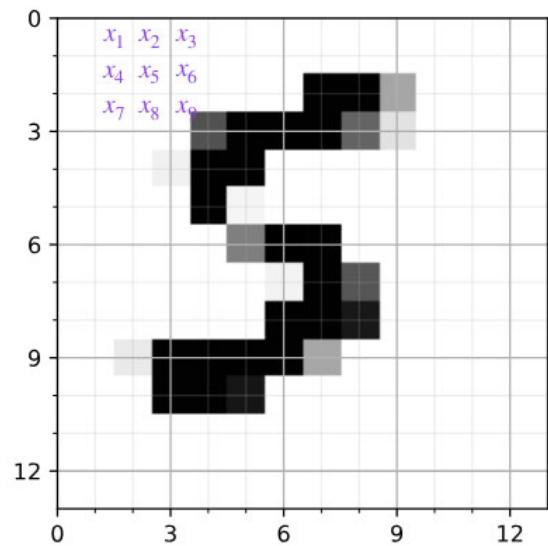
The inputs ( $x$ 's) differ as we slide over the image.

The weights ( $w$ 's) do not differ. → weight sharing

$$z = b + \sum_j w_j x_j$$



$w_1$	$w_2$	$w_3$
$w_4$	$w_5$	$w_6$
$w_7$	$w_8$	$w_9$



## weight sharing

- Rationale: A feature detector that works well in one region may also work well in another region
- A reduction in parameters to fit (compared to MLP)

# Convolution

- This is how we calculate the convolutional layer's output:

$$\text{ConvolvedFeature}(i, j) = (I * K)(i, j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i + a, j + b)K(a, b)$$

I: Input Image

K: Our Kernel

$k_h$  and  $k_w$ : The height and width of the Kernel

1	0	1
0	1	0
1	0	1

Figure:  
Convolv-  
ing Kernel

1 <small><math>\times 1</math></small>	1 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	0	0
0 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	1 <small><math>\times 0</math></small>	1	0
0 <small><math>\times 1</math></small>	0 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved  
Feature

- This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i, j) = (I * K)(i, j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i + a, j + b)K(a, b)$$

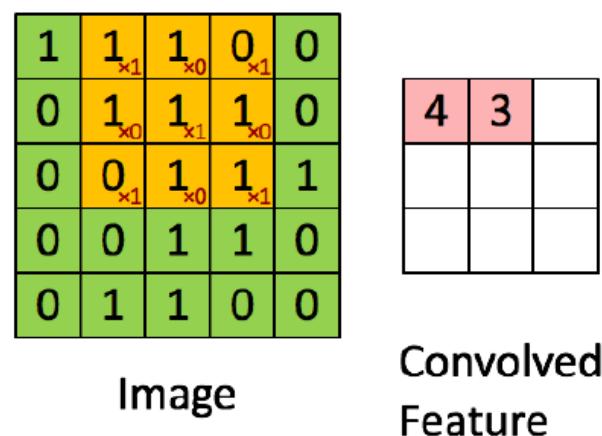
I: Input Image

K: Our Kernel

$k_h$  and  $k_w$ : The height and width of the Kernel

1	0	1
0	1	0
1	0	1

Figure:  
Convolv-  
ing Kernel



- This is how we calculate the convolutional layer's output:

$$\text{ConvolvedFeature}(i, j) = (I * K)(i, j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i + a, j + b)K(a, b)$$

I: Input Image

K: Our Kernel

$k_h$  and  $k_w$ : The height and width of the Kernel

1	0	1
0	1	0
1	0	1

Figure:  
Convolv-  
ing Kernel

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4	3	4

Convolved  
Feature

- This is how we calculate the convolutional layer's output:

$$\text{ConvolvedFeature}(i, j) = (I * K)(i, j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i + a, j + b)K(a, b)$$

I: Input Image

K: Our Kernel

$k_h$  and  $k_w$ : The height and width of the Kernel

1	0	1
0	1	0
1	0	1

Figure:  
Convolv-  
ing Kernel

1	1	1	0	0
0 <small>x1</small>	1 <small>x0</small>	1 <small>x1</small>	1	0
0 <small>x0</small>	0 <small>x1</small>	1 <small>x0</small>	1	1
0 <small>x1</small>	0 <small>x0</small>	1 <small>x1</small>	1	0
0	1	1	0	0

Image

4	3	4
2		

Convolved  
Feature

- This is how we calculate the convolutional layer's output:

$$\text{ConvolvedFeature}(i, j) = (I * K)(i, j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i + a, j + b)K(a, b)$$

I: Input Image

K: Our Kernel

$k_h$  and  $k_w$ : The height and width of the Kernel

1	0	1
0	1	0
1	0	1

Figure:  
Convolv-  
ing Kernel

1	1	1	0	0
0	1	1	0	0
0	0	1	1	0
0	0	1	1	0
0	1	1	0	0

Image

4	3	4
2	4	

Convolved  
Feature

- This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i, j) = (I * K)(i, j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a, b)$$

I: Input Image

K: Our Kernel

$k_h$  and  $k_w$ : The height and width of the Kernel

1	0	1
0	1	0
1	0	1

Figure:  
Convolv-  
ing Kernel

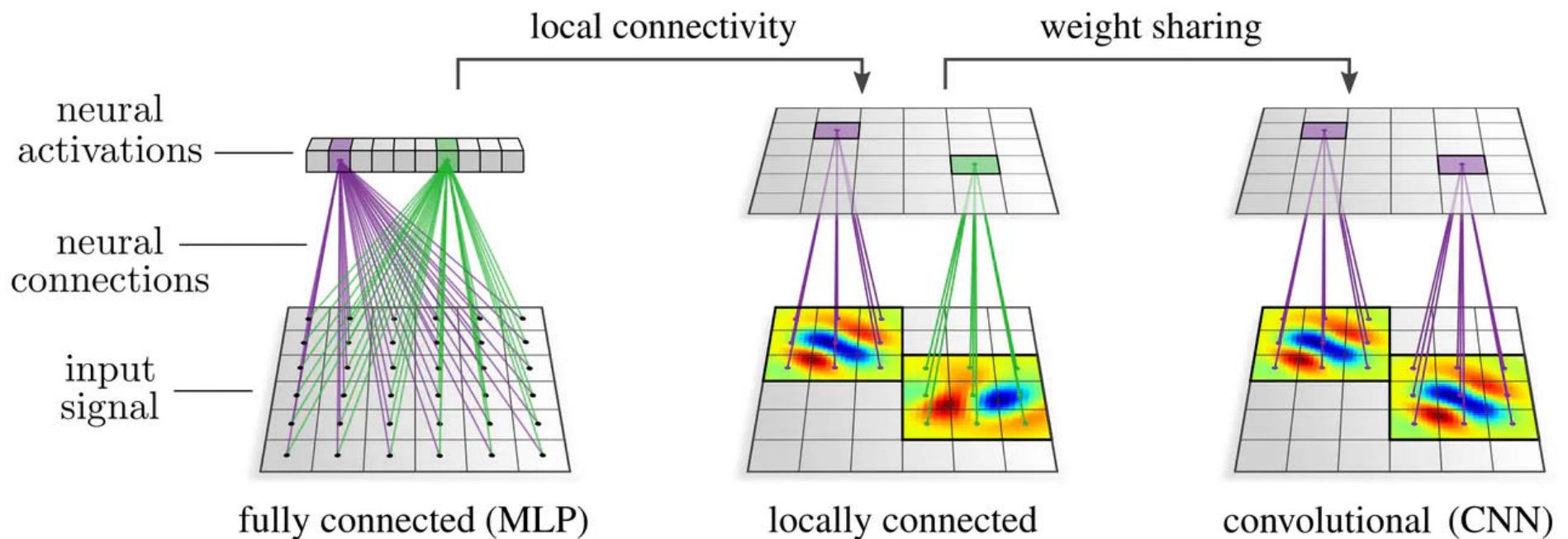
1	1	1	0	0
0	1	1	1	0
0	0	1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>
0	0	1 <sub>x0</sub>	1 <sub>x1</sub>	0 <sub>x0</sub>
0	1	1 <sub>x1</sub>	0 <sub>x0</sub>	0 <sub>x1</sub>

Image

4	3	4
2	4	3
2	3	4

Convolved  
Feature

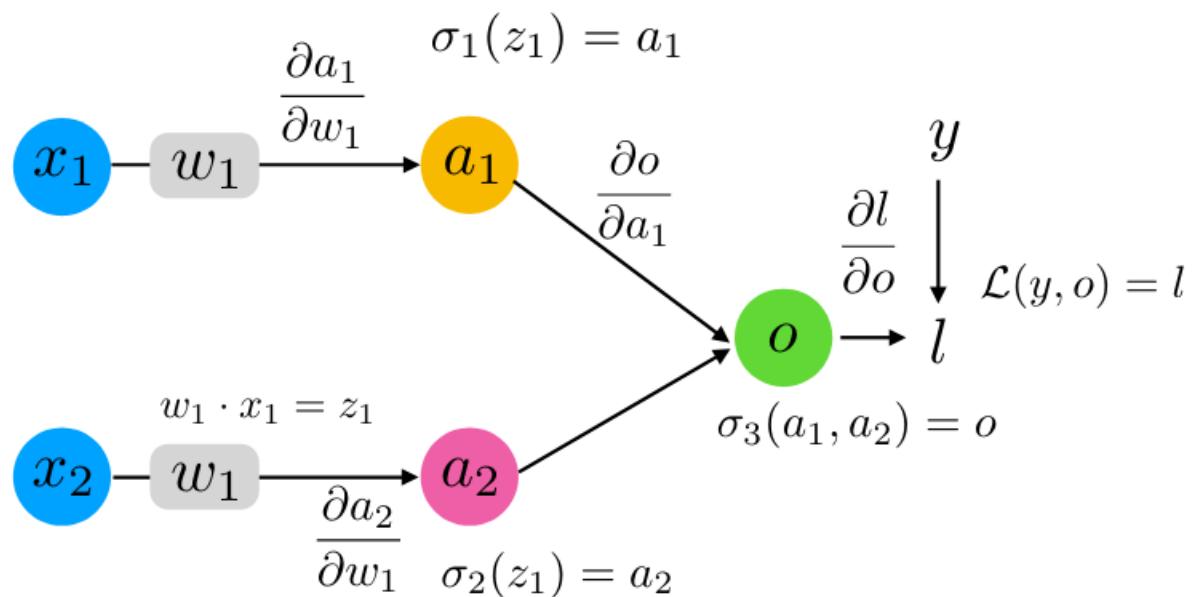
# Locality and Weight sharing



# Backpropagation in CNNs

- Same overall concept as before: Multivariable chain rule, but now with an additional weight sharing constraint

# Graph with Weight Sharing



Upper path

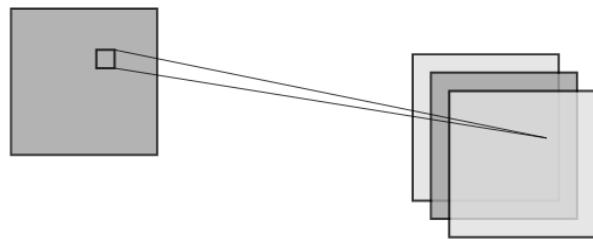
$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1} \quad (\text{multivariable chain rule})$$

Lower path

- Which problem is solved with weigh sharing and locality?

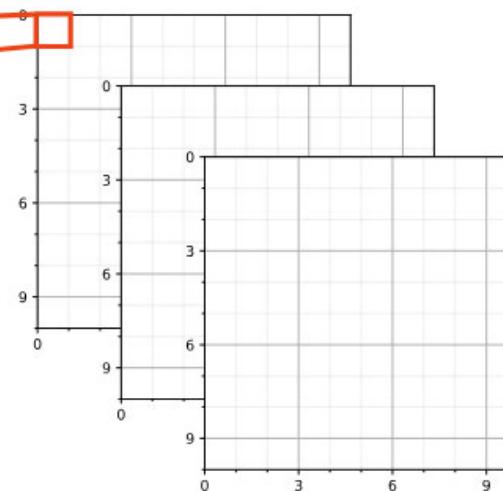
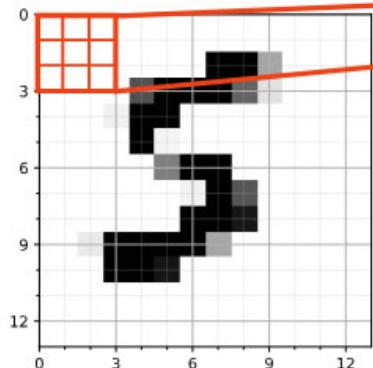
# Multiple Output Channels

1 input channel, 3 output channels



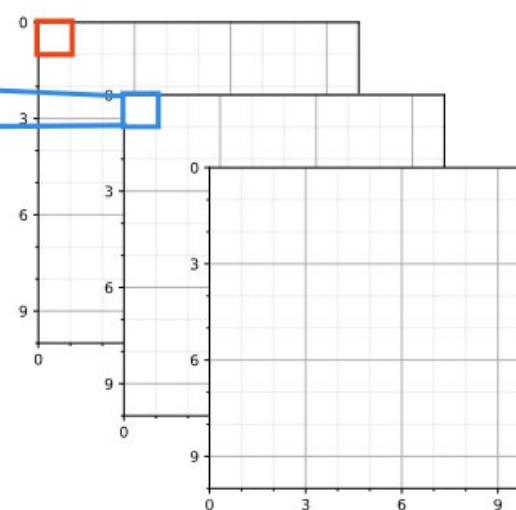
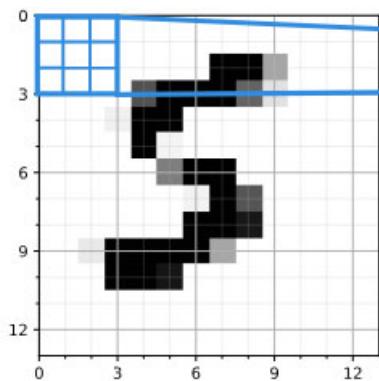
# 1 input channel 3 output channel

$$z = b^{(1)} + \sum_j w_j^{(1)} x_j$$

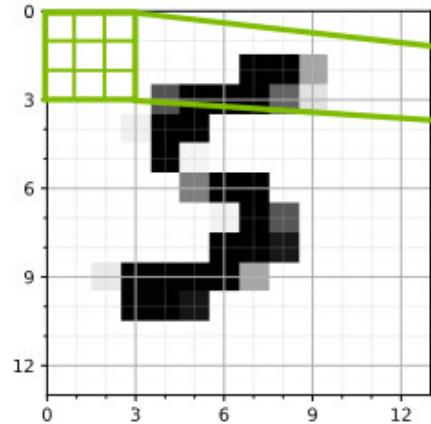


1 input channel 3 output channel

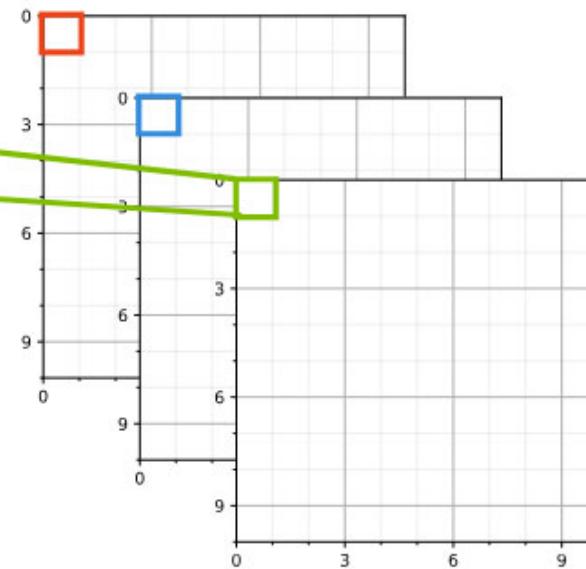
$$z = b^{(\text{@} 2)} + \sum_j w_j^{(\text{@} 2)} x_j$$



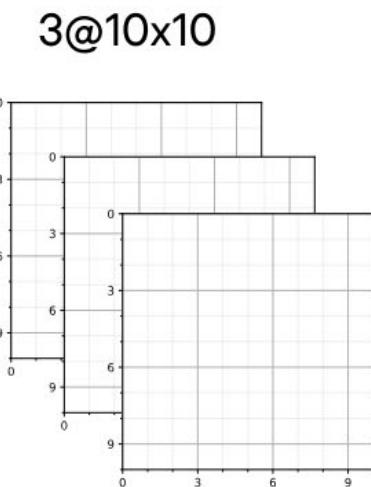
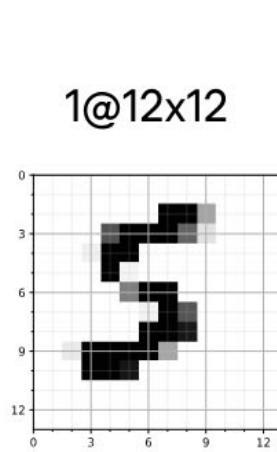
# 1 input channel 3 output channel



$$z = b^{(\text{@}3)} + \sum_j w_j^{(\text{@}3)} x_j$$



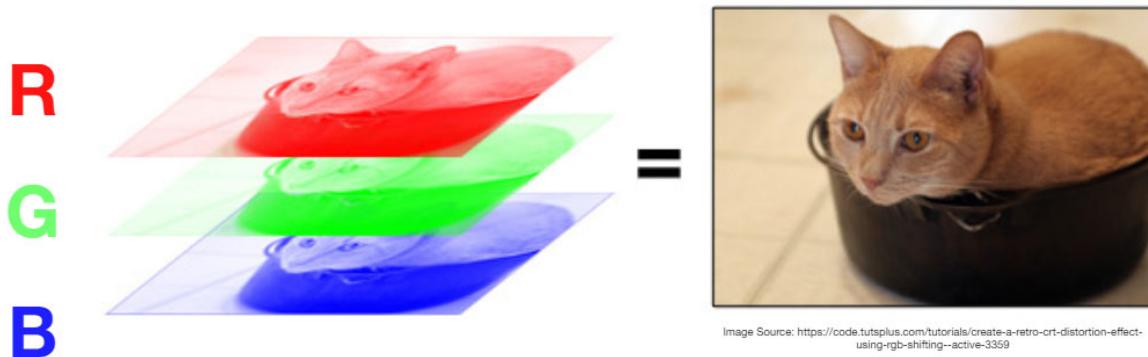
# 1 input channel 3 output channel



Multiple "feature detectors" (kernels) are used to create multiple feature maps

# Multiple input channel

RGB Image

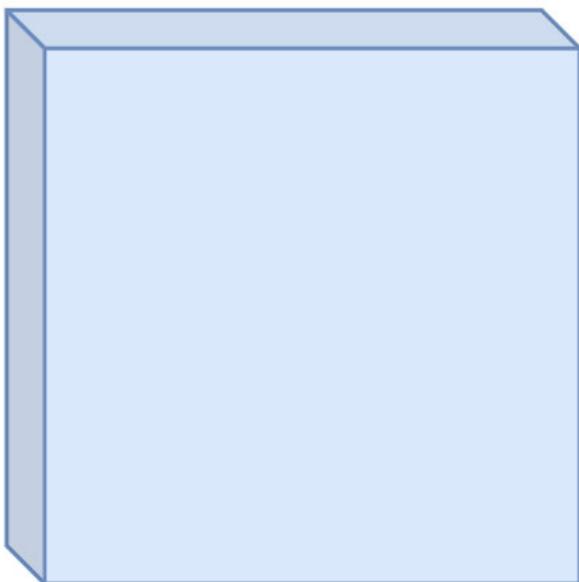


Color image as a stack of matrices

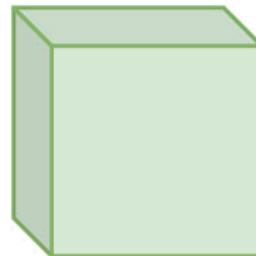
# Input channels

Filters always extend the full depth of the input.  
(#Input channels == #Filter Channels)

32x32x3 Image

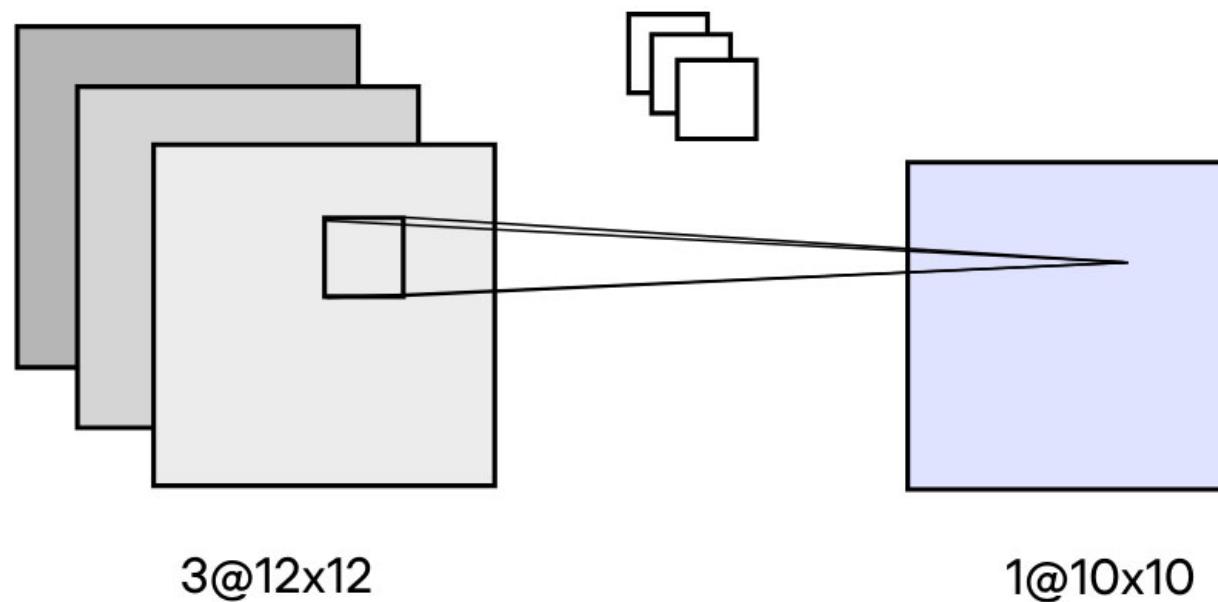


5x5x3 Filter

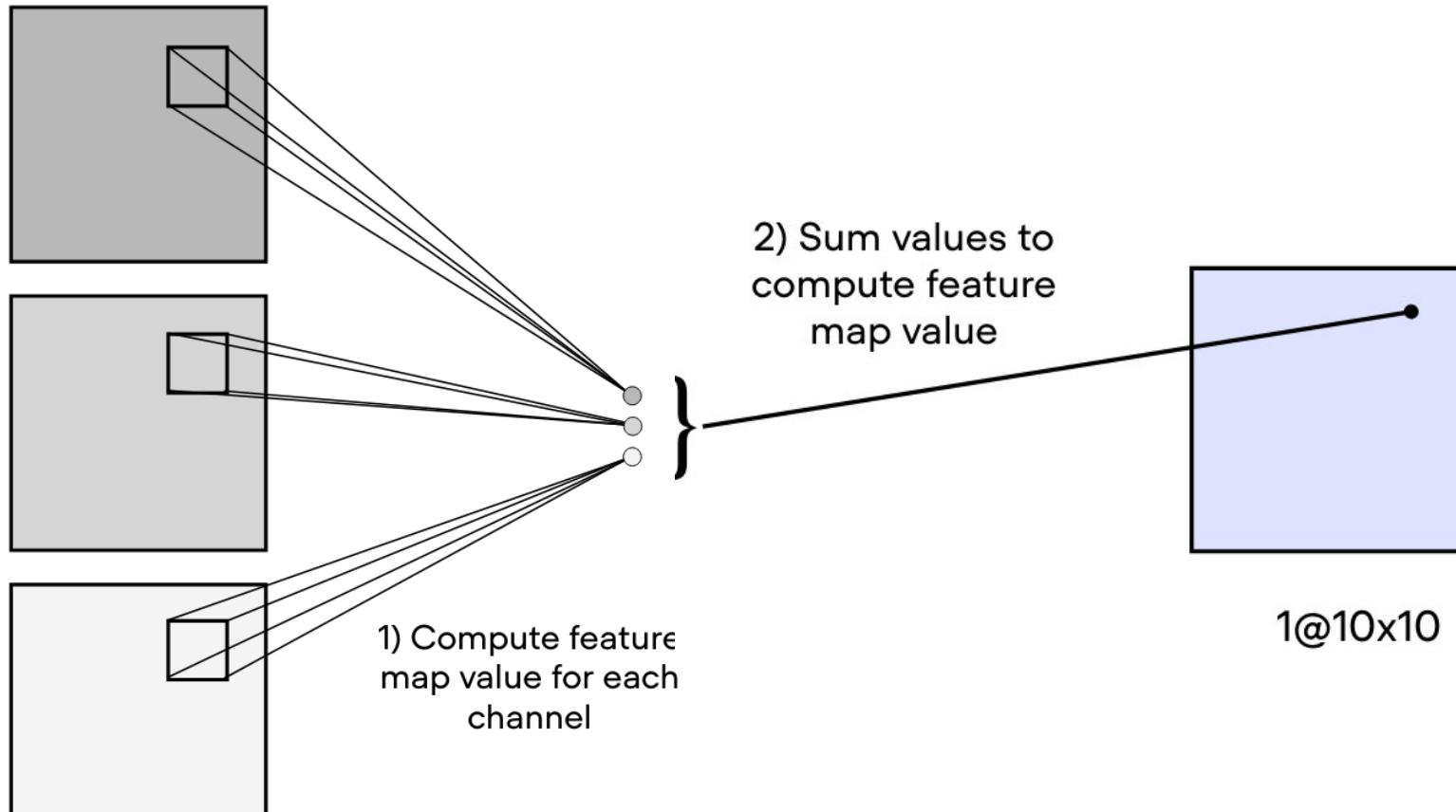


3 input channel, 1 output channel

kernel has 3 channels

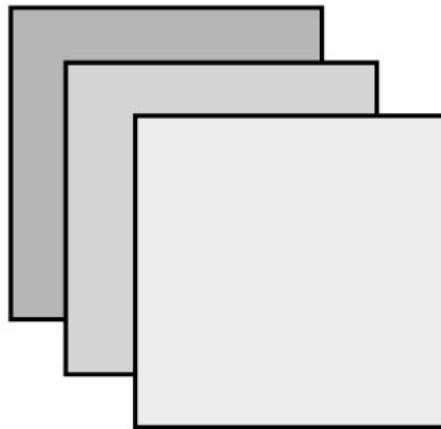


# 3 input channel, 1 output channel



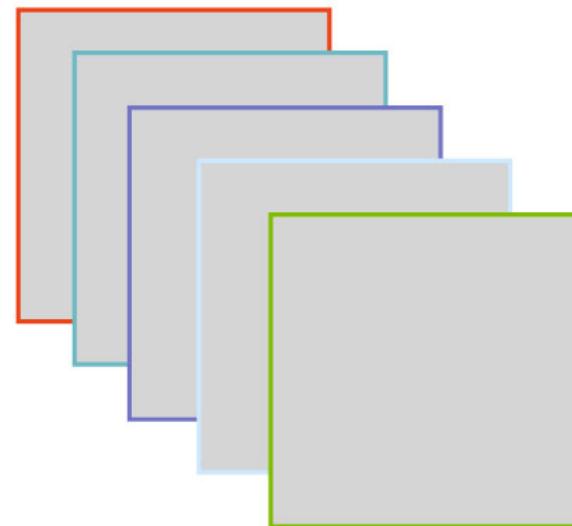
# Multiple input and output channels

3 **input** channels



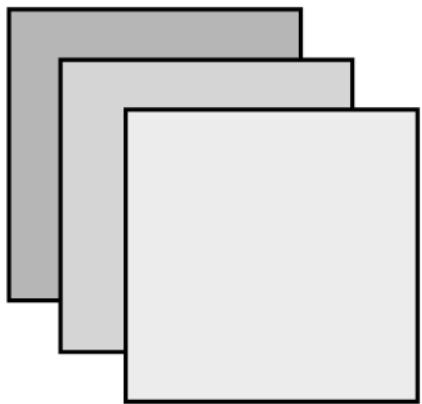
3@64x64

5 **output** channels

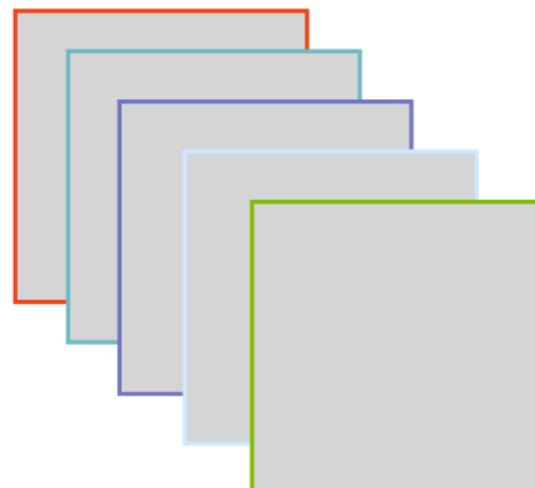


5@64x64

# Multiple input and output channels



3@64x64

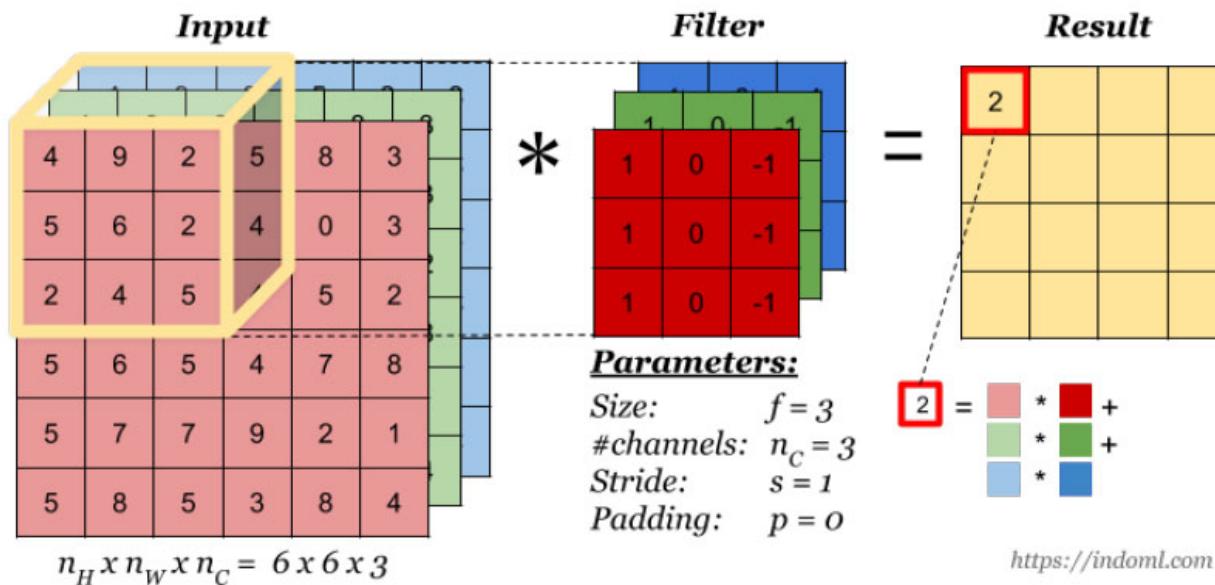


5@64x64

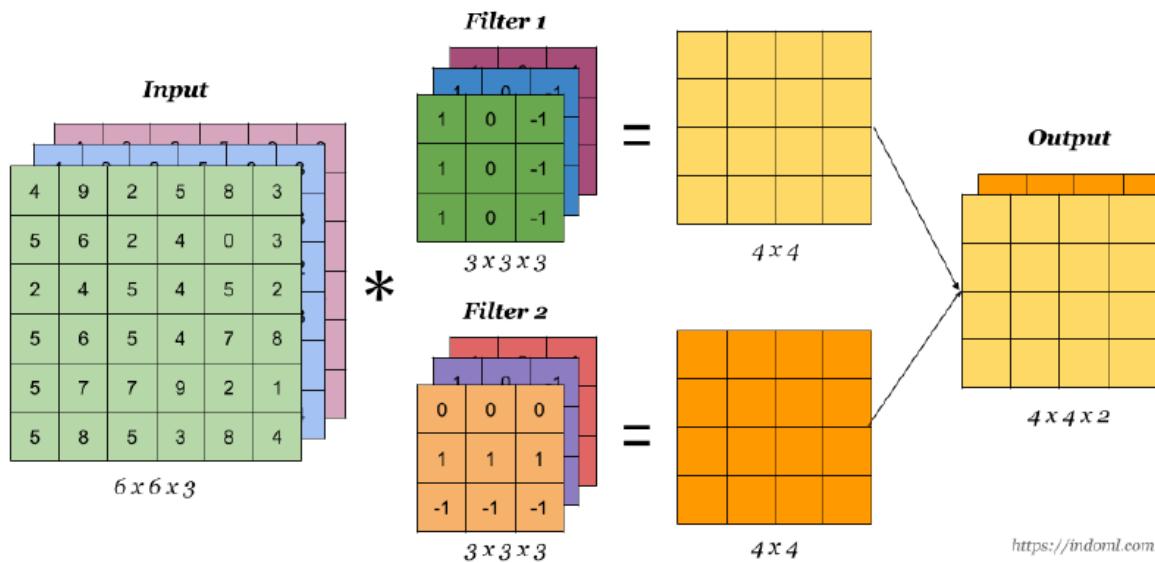
5 kernels with  
3 channels each



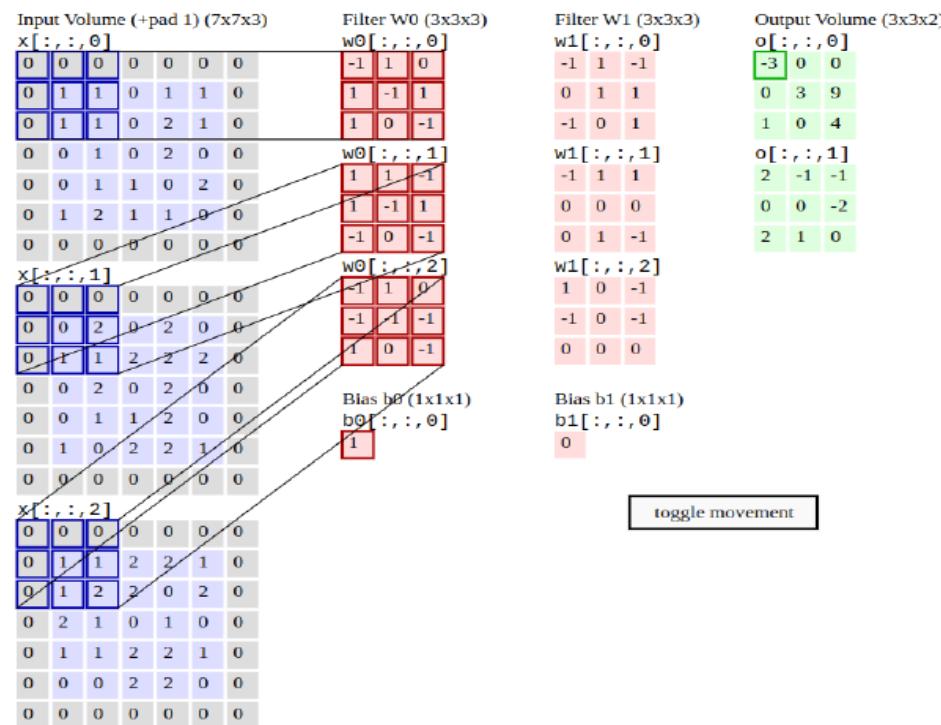
- As said before: "Filters always extend the full depth of the input. (#Input channels == #Filter Channels)"



- As said before: "Filters always extend the full depth of the input volume.  
(#Input channels == #Filter Channels)"
- Each filter results in one output channel. We can apply multiple different filters to obtain multiple output channels. Each channel can learn something distinct.



Let's have a closer look to how the calculations with more than one filter work:



Input Volume (+pad 1) (7x7x3)  
 $x[:, :, 0]$

0 0 0	0 0 0	0 0 0
0 1 1	0 1 1	0 1 0
0 1 1	0 2 1	0 0 0

Filter W0 (3x3x3)  
 $w0[:, :, 0]$

Filter W1 (3x3x3)  
 $w1[:, :, 0]$

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

Output Volume (3x3x2)  
 $o[:, :, 0]$

-3 0 0	0 3 9	1 0 4
2 -1 -1	0 0 0	0 0 -2
2 1 0	0 1 -1	0 0 0

$x[:, :, 1]$

0 0 0	0 0 0	0 0 0
0 0 2	0 2 0	0 0 0
0 1 1	2 2 2	0 0 0

$w0[:, :, 1]$

$w1[:, :, 1]$

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

$x[:, :, 2]$

0 0 2	0 2 0	0 0 0
0 0 1	1 2 0	0 0 0
0 1 0	2 2 1	0 0 0

$w0[:, :, 2]$

$w1[:, :, 2]$

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

Bias b0 (1x1x1)  
 $b0[:, :, 0]$

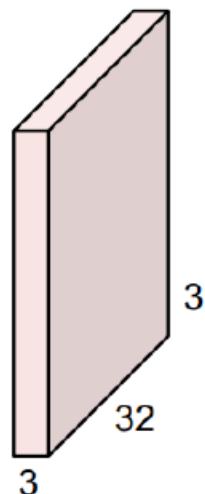
Bias b1 (1x1x1)  
 $b1[:, :, 0]$

toggle movement

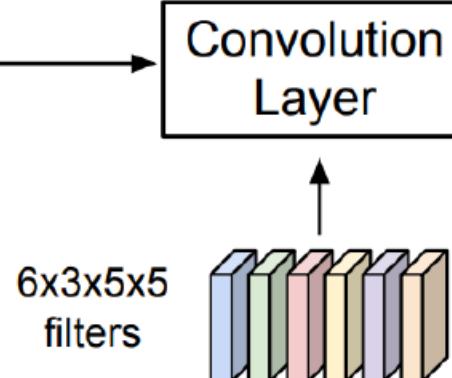
- We apply several filters and stack the output together to get a multilayer output, with each layer representing something it learned.

## Convolution Layer

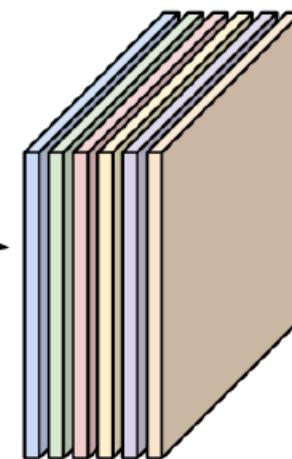
3x32x32 image



Consider 6 filters,  
each 3x5x5



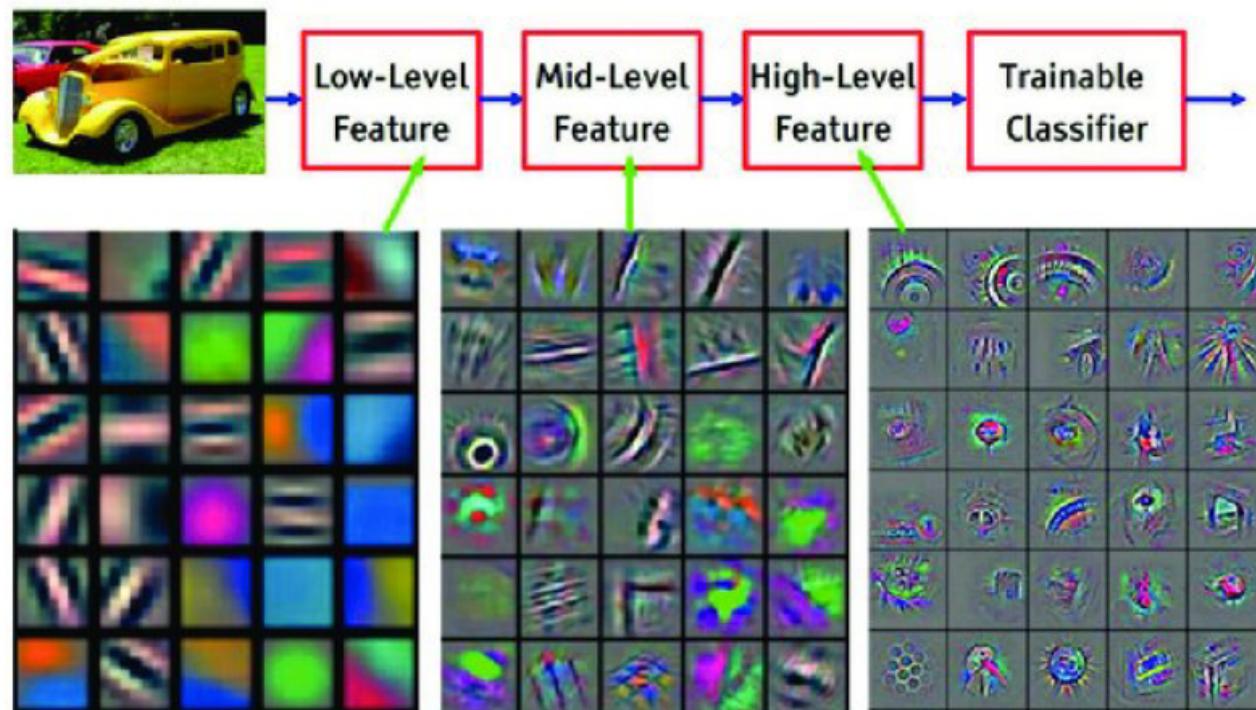
6 activation maps,  
each 1x28x28

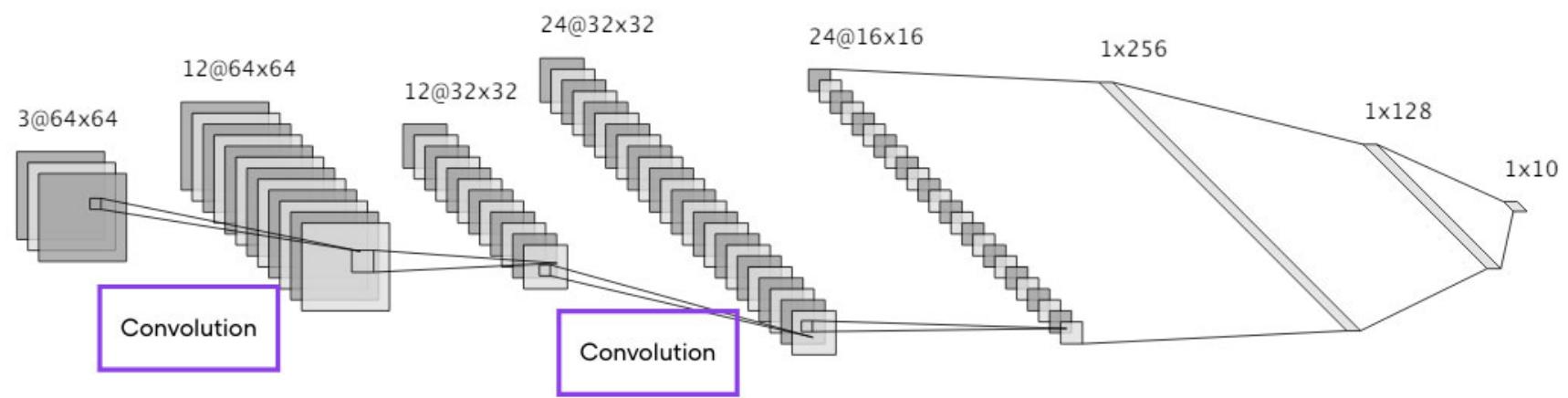


Stack activations to get a  
6x28x28 output image!

Slide inspiration: Justin Johnson

- We apply several filters and stack the output together to get a multilayer output, with each layer representing something it learned.
- Some kernels learn to recognize vertical lines, some circles, and some, specific objects:



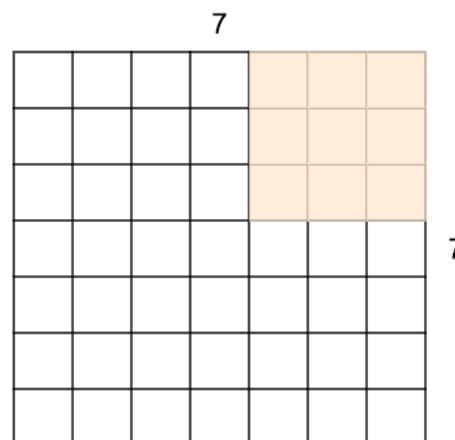


# What is a Stride

The amount of movement between applications of the filter to the input image is referred to as the stride, and it is almost always symmetrical in height and width dimensions.

Closer look

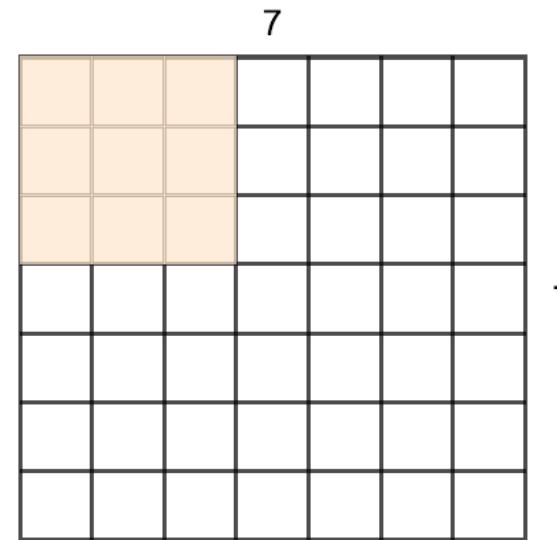
- 7x7 input with 3x3 filter
- This was a Stride 1 filter
- => Outputs 5x5



# Strides

Now let's use Stride 2

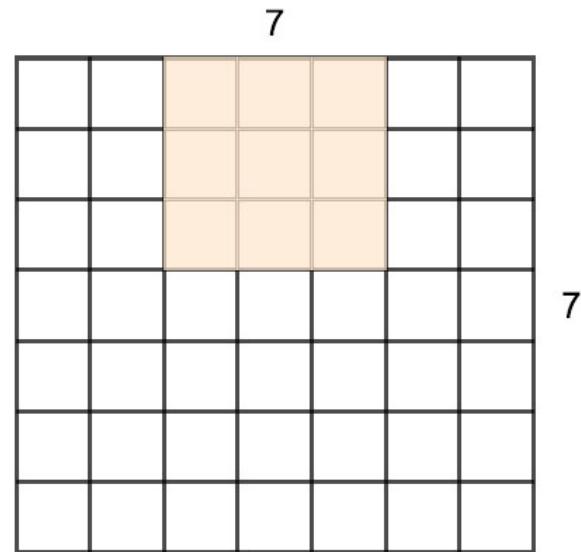
- 7x7 input with 3x3 filter



## Strides

Now let's use Stride 2

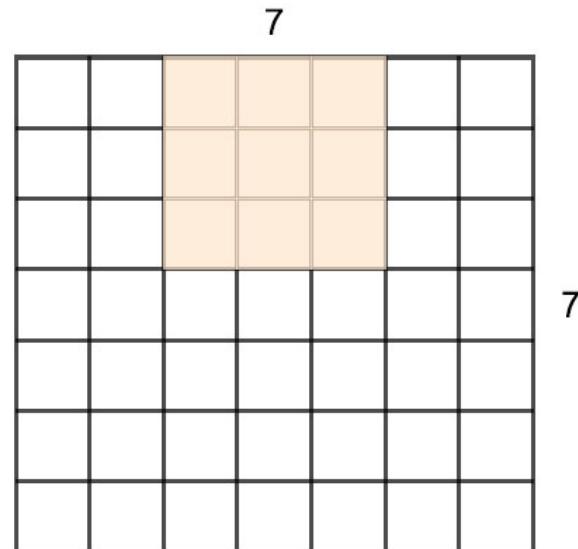
- #### ■ 7x7 input with 3x3 filter



# Strides

Now let's use Stride 2

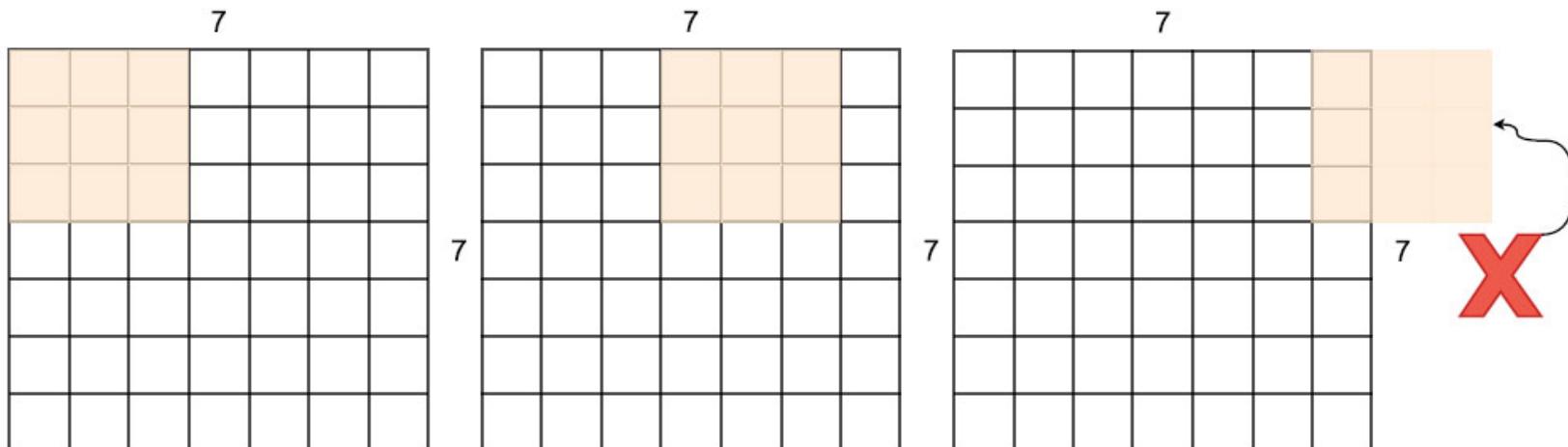
- 7x7 input with 3x3 filter



# Strides

Stride 3?

- 7x7 input with 3x3 filter

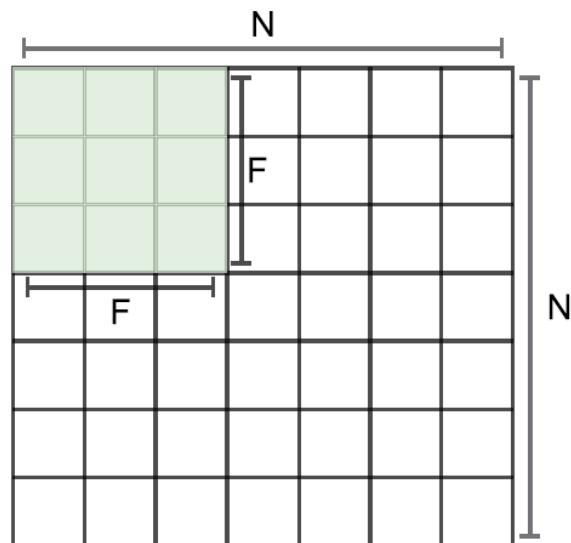


# Strides

So 7x7 input with 3x3 filter and stride 3 doesn't work!

Let's do the calculations:

$$OutputSize = (N - F)/Stride + 1$$

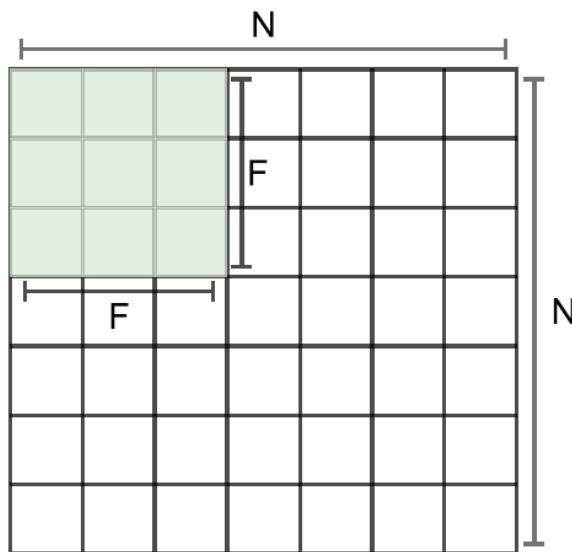


# Strides

$$OutputSize = (N - F)/Stride + 1$$

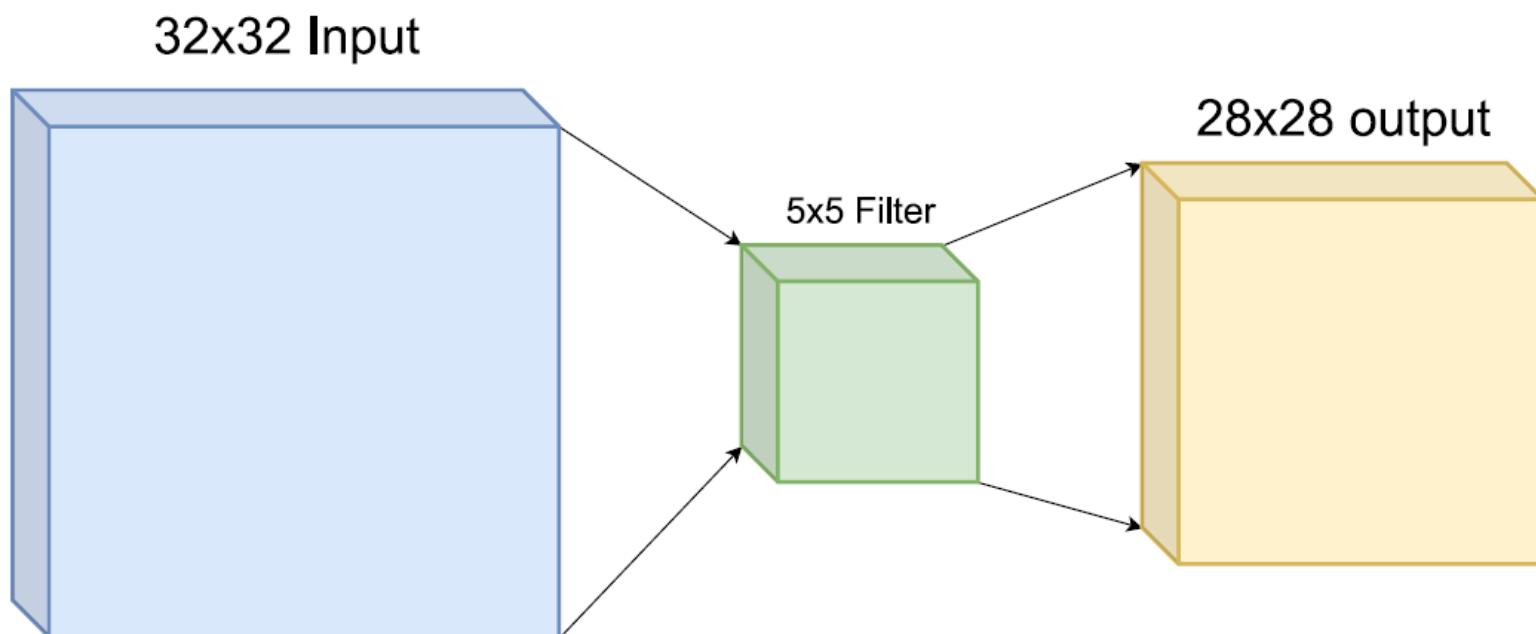
$N = 7, F = 3 \Rightarrow$

- Stride 1  $\Rightarrow (7 - 3)/1 + 1 = 5$
- Stride 2  $\Rightarrow (7 - 3)/2 + 1 = 3$
- Stride 3  $\Rightarrow (7 - 3)/3 + 1 = 2.33 :))$



# Padding

1. Borders don't get enough attention.
2. Outputs shrink!



# Padding

- | We can use Padding to preserve the dimensionality
- | It ensures that all pixels are used equally frequently

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

$6 \times 6 \rightarrow 8 \times 8$

\*

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

$3 \times 3$

=

-10	-13	1			
-9	3	0			

$6 \times 6$

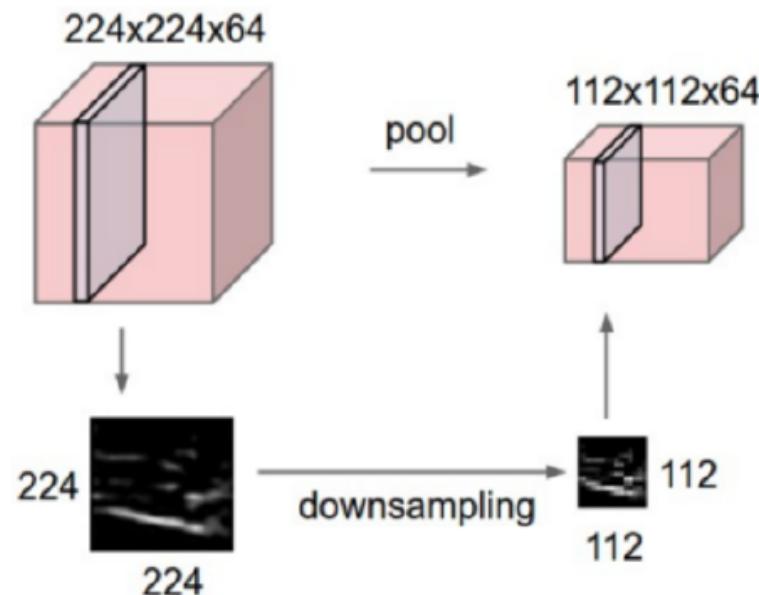
## Padding

$$OutputSize = (N + 2P - F)/Stride + 1$$

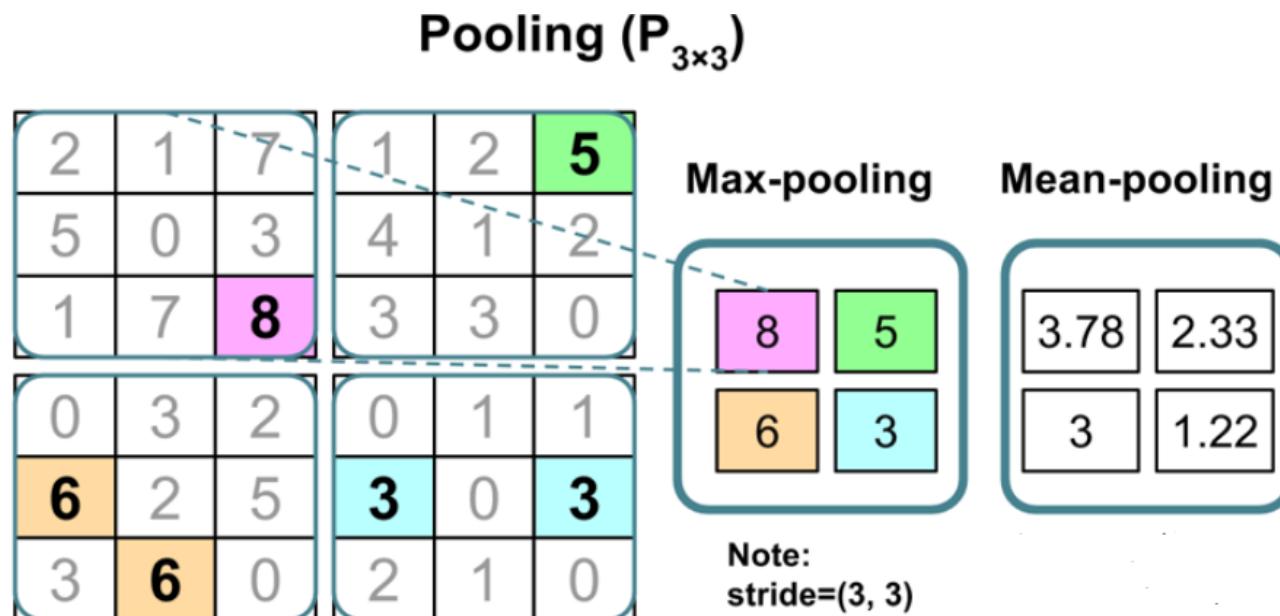
It is common to use  $P = (F - 1)/2$  with stride 1 to preserve the input size.

# Pooling

Pooling is performed in neural networks to reduce variance and computation complexity



# Pooling Layers Can Help With Local Invariance



Note that typical pooling layers do not have any learnable parameters

Downside: Information is lost.

May not matter for classification, but applications where relative position is important (like face recognition)

In practice for CNNs: some image preprocessing still recommended

# Pooling benefits

A Pooling layer's goal is to sub-sample the input in order to reduce:

- ▶ The computational load
- ▶ The memory usage
- ▶ The number of parameters
- ▶ Limiting the risk of overfitting

# Max Pooling

Most common type of pooling layer

Invariance to small translations

