

**COPYRIGHT WARNING:** Copyright in these original lectures is owned by Monash University. You may transcribe, take notes, download or stream lectures for the purpose of your research and study only. If used for any other purpose, (excluding exceptions in the Copyright Act 1969 (Cth)) the University may take legal action for infringement of copyright.

Do not share, redistribute, or upload the lecture to a third party without a written permission!

## FIT3181 Deep Learning

Week 04: Convolutional Neural Networks

**Lecturer: Lim Chern Hong**

Email: [lim.chernhong@monash.edu](mailto:lim.chernhong@monash.edu)

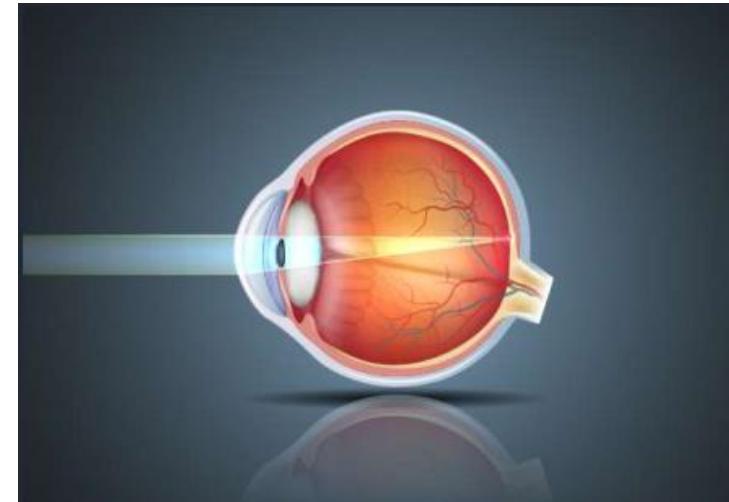
# Outline

- Origin of human vision
- Convolutional Neural Networks
  - Convolutional operators in 1D, 2D, 3D
  - Standard CNN architecture:
    - Convolutional layer, Pooling layer, Fully-connected layer
  - CNN-based image recognition system
    - Batch norm and dropout layers
    - Popular datasets in vision
- Further reading recommendation
  - [HandsOn, chapter 14, [FDL, chapter 5], [Dive into DL, chapter 6].
  - “CS231n: Convolutional Neural Networks for Visual Recognition”, Stanford University, 2016.

# Origin of Human Vision

# Human vision

- The vision begins with eyes, but truly take places in the brain.
- The collaboration between the eyes and the brain, called the **primary visual pathway**, is the reason we can make sense of the world around us.



**JUST BECAUSE YOU'RE MAD AT SOMEONE  
DOESN'T MEAN YOU STOP LOVING THEM**

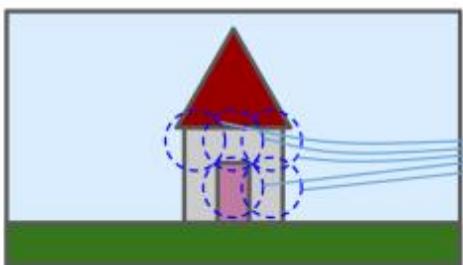


# Recognition is easy and fundamental

(Source: <https://www.youtube.com/watch?v=40riCqvRoMs>)



A boy  
A boy is playing  
A boy is playing in the park



(Source: Hands On, Ch. 14)

(Source: www.pngwing.com)

**Biological neurons** in the **visual cortex** respond to specific patterns in **small regions** of the **visual field** called **receptive fields**; as the visual signal makes its way through **consecutive brain modules**, neurons respond to more **complex patterns** in larger receptive fields.

# Recognition is much harder for machines

Human vision



14	22	1	11	13
10	31	8	23	6
202	139	173	102	5
137	2	63	208	64
55	43	99	198	143
100	32	46	63	104
96	122	29	9	205
103	61	238	108	95
68	145	245	249	160
14	226	68	153	250
189	47	116	181	24
204	87	88	207	11
5	185	66	248	147

# Human vision vs computer vision

**Human vision**



**Computer vision**

58	24	4	93	114	50	38	178	163	127	211	176	39	12	88	191
23	10	160	196	193	223	29	240	25	53	234	232	125	88	70	141
54	7	126	101	194	118	64	108	155	58	53	170	173	72	219	234
224	89	234	149	185	106	252	0	222	118	41	70	193	25	10	86
6	89	54	236	46	55	207	162	198	76	71	18	41	96	136	13
20	131	173	254	166	198	148	44	80	56	126	63	118	52	216	81
143	171	194	205	197	132	125	208	127	29	179	232	109	210	50	10
86	49	90	220	162	41	28	153	96	240	191	186	179	38	57	51
138	90	179	39	42	34	100	246	215	134	39	40	253	167	201	93
116	250	142	106	139	5	222	39	200	150	110	60	125	118	201	18
182	144	96	42	152	216	166	248	176	243	224	76	242	52	224	58
244	117	62	183	80	34	117	125	81	203	77	224	201	167	30	141
142	148	161	241	131	159	188	232	73	134	199	45	109	74	27	250
66	158	244	9	253	149	152	64	108	57	61	192	22	111	73	10
206	19	90	68	185	138	228	107	143	114	10	31	8	238	68	47
29	43	186	2	214	174	33	253	183	181	202	139	173	102	5	72
170	1	170	64	110	247	244	118	163	203	137	2	63	208	64	131
98	34	92	145	14	122	35	111	85	255	55	43	99	198	143	256
226	88	133	62	140	212	235	45	238	83	100	32	46	63	104	151
219	46	170	76	58	213	126	66	61	154	96	122	29	9	205	164
71	106	191	194	78	147	224	190	179	39	103	61	238	108	95	148
98	145	27	125	53	38	189	224	23	239	68	145	245	249	160	161
25	200	201	88	21	160	159	237	82	183	14	236	68	153	250	140
126	182	37	225	118	180	195	118	143	123	189	247	116	181	24	50
167	194	188	101	96	194	143	140	103	2	204	87	88	207	11	36
205	127	184	108	241	66	10	174	128	13	5	185	66	248	147	36
163	37	97	52	70	222	30	14	79	75	81	193	148	106	144	68

**pixel**

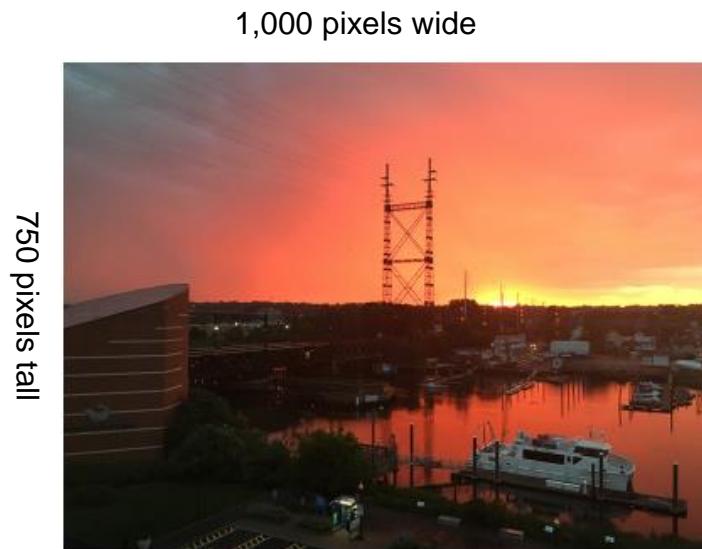
A red arrow points from the word "pixel" to the first value in the 2D array, which is 58.

A red box highlights a 4x4 sub-section of the original image, and another red box highlights a 4x4 sub-section of the corresponding 2D pixel array.

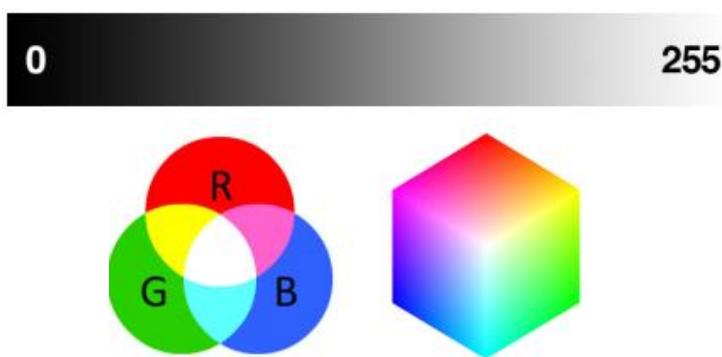
58	24	4	93	114	50	38	178	163	127	211	176	39	12	88	191
23	10	160	196	193	223	29	240	25	53	234	232	125	88	70	141
54	7	126	101	194	118	64	108	155	58	53	170	173	72	219	234
224	89	234	149	185	106	252	0	222	118	41	70	193	25	10	86
6	89	54	236	46	55	207	162	198	76	71	18	41	96	136	13
20	131	173	254	166	198	148	44	80	56	126	63	118	52	216	81
143	171	194	205	197	132	125	208	127	29	179	232	109	210	50	10
86	49	90	220	162	41	28	153	96	240	191	186	179	38	57	51
138	90	179	39	42	34	100	246	215	134	39	40	253	167	201	93
116	250	142	106	139	5	222	39	200	150	110	60	125	118	201	18
182	144	96	42	152	216	166	248	176	243	224	76	242	52	224	58
244	117	62	183	80	34	117	125	81	203	77	224	201	167	30	141
142	148	161	241	131	159	188	232	73	134	199	45	109	74	27	250
66	158	244	9	253	149	152	64	108	57	61	192	22	111	73	10
206	19	90	68	185	138	228	107	143	114	10	31	8	238	68	47
29	43	186	2	214	174	33	253	183	181	202	139	173	102	5	72
170	1	170	64	110	247	244	118	163	203	137	2	63	208	64	131
98	34	92	145	14	122	35	111	85	255	55	43	99	198	143	256
226	88	133	62	140	212	235	45	238	83	100	32	46	63	104	151
219	46	170	76	58	213	126	66	61	154	96	122	29	9	205	164
71	106	191	194	78	147	224	190	179	39	103	61	238	108	95	148
98	145	27	125	53	38	189	224	23	239	68	145	245	249	160	161
25	200	201	88	21	160	159	237	82	183	14	236	68	153	250	140
126	182	37	225	118	180	195	118	143	123	189	247	116	181	24	50
167	194	188	101	96	194	143	140	103	2	204	87	88	207	11	36
205	127	184	108	241	66	10	174	128	13	5	185	66	248	147	36
163	37	97	52	70	222	30	14	79	75	81	193	148	106	144	68

# Pixels: the building block of images

- Pixels are the raw building blocks of an image. Every image consists of a **set of pixels**.
  - Most pixels are represented in two ways:
    - Grayscale/single channel
      - A scalar between 0 (black) and 255 (white)
    - Color
      - Color pixels are represented in RGB system
        - Red, green, and blue channels



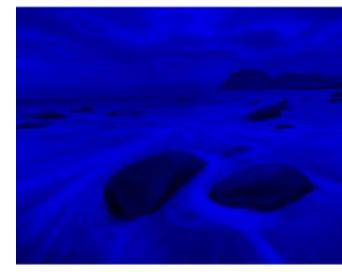
# pixels = 750 x 1,000 = 750,000



# RGB colour space

- A colour pixel is represented as a 3D tuple (R,G,B)
  - $R, G, B \in \{0,1, \dots, 255\}$
- The number of possible colours
  - $256 \times 256 \times 256 = 16,777,216$

$$\begin{array}{c} 252 + 198 + 188 = \text{light orange} \\ 22 + 159 + 230 = \text{cyan} \end{array}$$

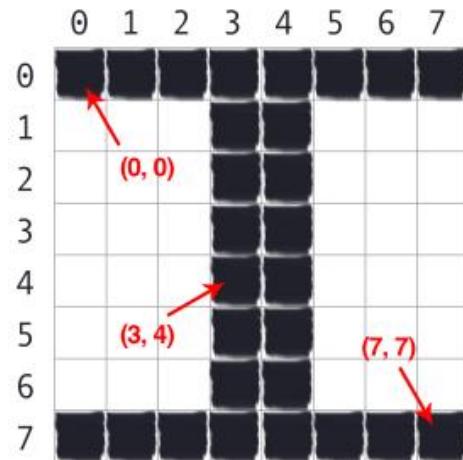


red

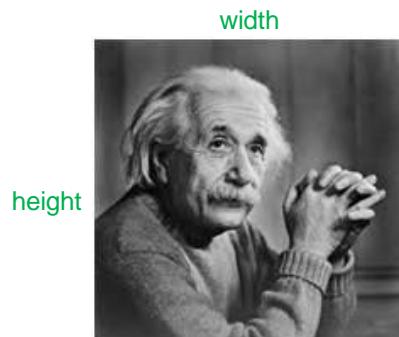
blue

green

# Image coordinate system



An image of letter I. Each pixel has a coordinate.



A grayscale image can be viewed as a 2D tensor



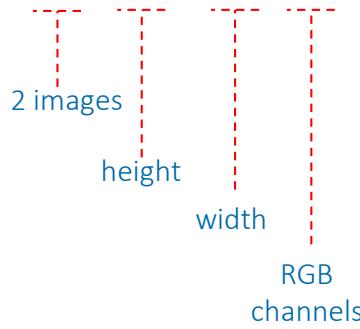
A colour image can be viewed as a 3D tensor

# Images as Tensor

```
import numpy as np
from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")
batch = np.array([china, flower], dtype=np.float32)
print(batch.shape)
```

(2, 427, 640, 3)



-9	4	2	5	7
3	0	12	8	61
1	23	-6	45	2
22	3	-1	72	6

*batch[0]: china*

-9	4	2	5	7
3	0	12	8	61
1	23	-6	45	2
22	3	-1	72	6

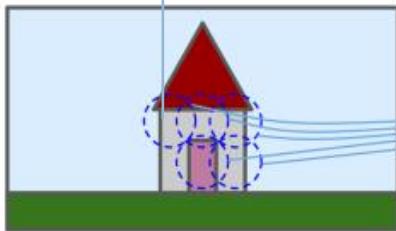
*batch[1]: flower*



# Convolutional Neural Networks

# How to simulate human's vision?

visual receptive field

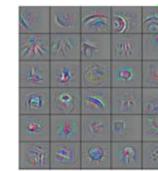
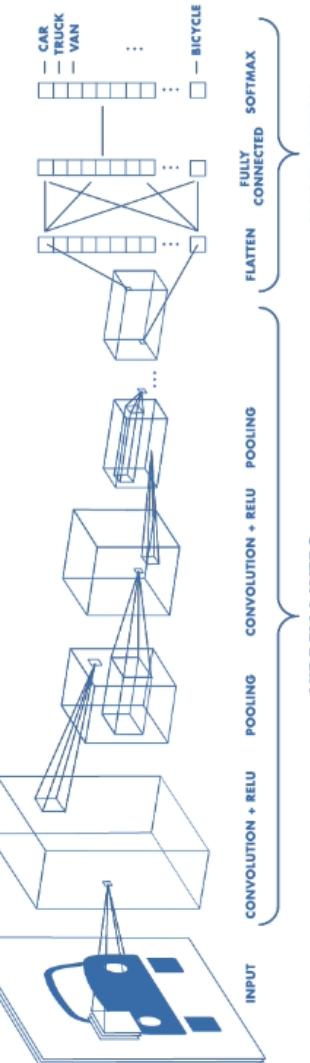
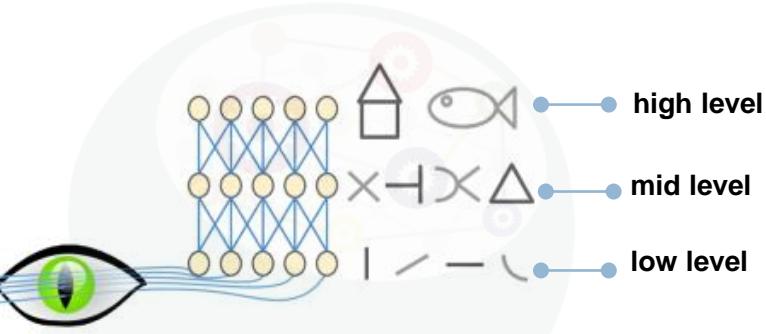


(Source: Hands On, Ch. 14)

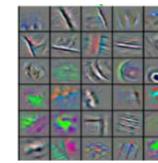
David H. Hubel and Torsten Wiesel performed a series of experiments on cats and monkeys in 1958 and 1959 (the Nobel Prize in Physiology or Medicine in 1981).

- ❖ Low-level neurons can capture small visual receptive fields with low-level patterns.
- ❖ Higher-level neurons are based on the outputs of neighboring lower-level neurons that can detect all sorts of complex patterns in any area of the visual field.

- Building-blocks of **Convolutional Neural Networks**
  - Convolutional layers
  - Pooling layers
    - Max pooling or average pooling
  - Fully-connected layers
  - Batch normalization layers



High-level feature  
- Object



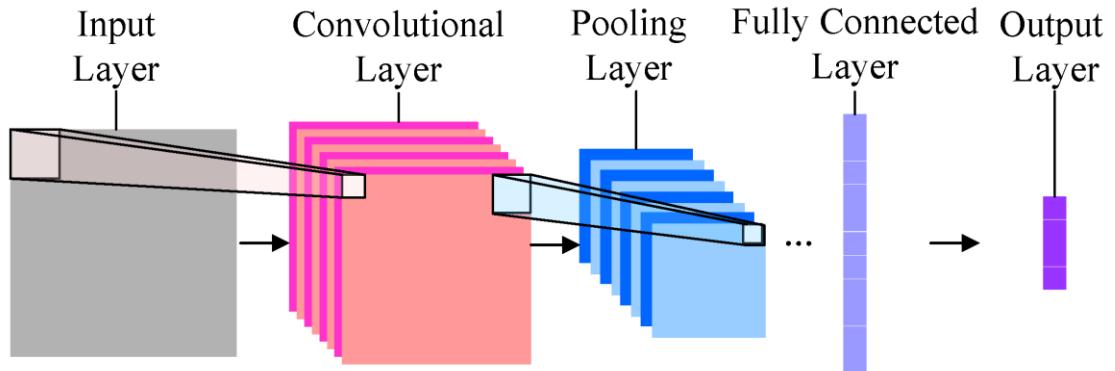
Mid-level feature  
- Shape



Low-level feature  
- Edge



# General architecture of CNNs



1. **Convolutional layer**
2. **Pooling layer**
3. **Fully connected layer**

# Convolutional Layer

# Convolution operation

- ❖  $W$  and  $x$  are **two tensors** with the **same shape**, the **convolutional operation** between  $W$  and  $x$  defined as

$$W * x = \text{sum}(W \otimes x)$$

where  $W \otimes x$  specifies the **element-wise product** and  $\text{sum}$  returns the **sum of all elements** in a tensor and multi-dimensional array.

- **1D convolutional operation** ( $W, x \in \mathbb{R}^{m \times 1}$ )

- $W * x = \sum_{i=1}^m W_i x_i$

$$W = \begin{bmatrix} W_1 \\ \dots \\ W_m \end{bmatrix}, x = \begin{bmatrix} x_1 \\ \dots \\ x_m \end{bmatrix}$$

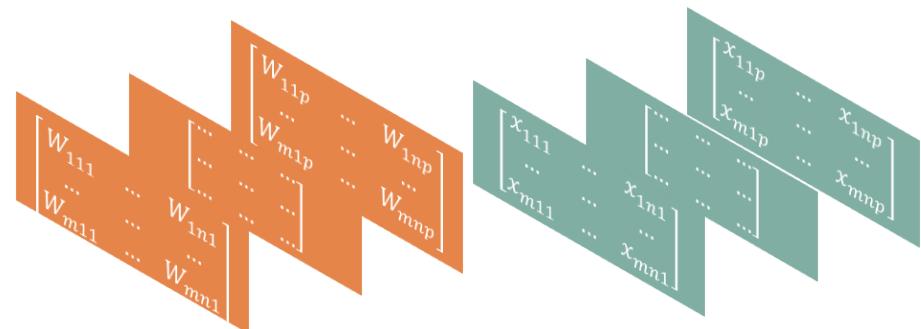
- **2D convolutional operation** ( $W, x \in \mathbb{R}^{m \times n}$ )

- $W * x = \sum_{i=1}^m \sum_{j=1}^n W_{ij} x_{ij}$

$$W = \begin{bmatrix} W_{11} & \dots & W_{1n} \\ \dots & \dots & \dots \\ W_{m1} & \dots & W_{mn} \end{bmatrix} \quad x = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \dots & \dots & \dots \\ x_{m1} & \dots & x_{mn} \end{bmatrix}$$

- **3D convolutional operation** ( $W, x \in \mathbb{R}^{m \times n \times p}$ )

- $W * x = \sum_{i=1}^m \sum_{j=1}^n \sum_{k=1}^p W_{ijk} x_{ijk}$



# Convolution operation – 1D filter

size = 14

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

filter = 2x1

0.5	0.5
-----	-----

# Convolution operation – 1D filter

size = 14

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

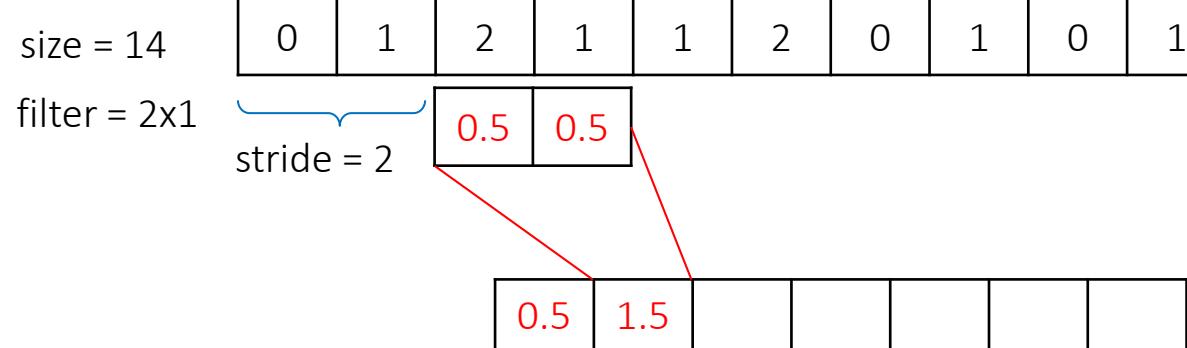
filter = 2x1

0.5	0.5
-----	-----

0.5						
-----	--	--	--	--	--	--

$$0 \times 0.5 + 1 \times 0.5$$

# Convolution operation – 1D filter



# Convolution operation – 1D filter

size = 14

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

filter = 2x1

0.5	0.5
-----	-----

stride = 2

0.5	1.5	1.5				
-----	-----	-----	--	--	--	--

# Convolution operation – 1D filter

size = 14

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

filter = 2x1

stride = 2

0.5	1.5	1.5	0.5	0.5	1.0	0.5
-----	-----	-----	-----	-----	-----	-----

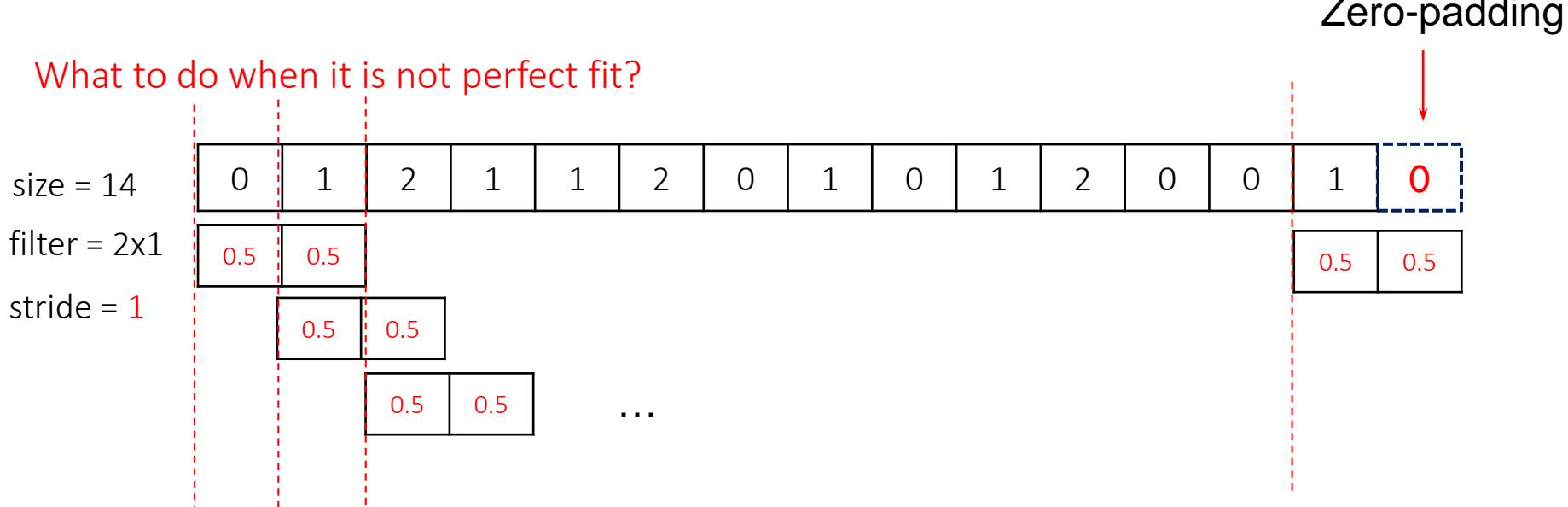
0.5	0.5
-----	-----

$$\text{new size} = 7 = \frac{14}{2} = \frac{\text{original size}}{\text{stride}}$$

- E.g,  $x(t)$  is the speech signal at time  $t$ .
- To obtain a more stable signal, we need to average it using a weight function  $w(a)$ 
  - $s(t) = \int x(a)w(t - a)da$
  - This operation is called **convolution**:  $s(t) = (x * w)(t)$
  - $x$  is input and  $w$  is kernel (or filter).
  - The output  $s$  is sometimes called feature map

# Convolution operation – 1D filter

What to do when it is not perfect fit?



Exercise 1: what is the output size in this case?

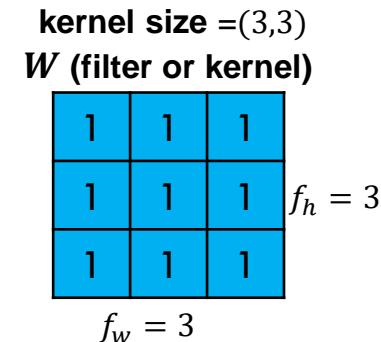
Exercise 2: if stride = 3, do we need zero-padding, and if yes, what would be the output size?

# Convolution layer with zero padding

$x$ (input tensor (7,7))	strides = (2,2)
0 0 0 0 0 0 0 0 0	
0 1 -2 -1 5 3 2 1 0	
0 1 3 -1 4 3 3 1 0	
0 1 -2 1 6 3 3 2 0	
0 2 -2 2 5 2 1 0 0	
0 0 3 -2 5 4 1 2 0	
0 1 2 -3 1 1 2 -1 0	
0 1 -2 -1 1 2 1 1 0	
0 0 0 0 0 0 0 0 0	

$W_i=7$    Zero padding and  $p = 1$

In TF or Keras, padding = 'same'



$H_i = 7$

feature map

3	8	19	7
3	15	30	10
6	11	22	5
2	-2	9	3

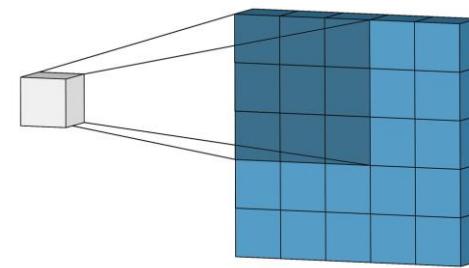
$H_o = 4$   
 $W_o = 4$

- The slicing window moves from left → right, top → bottom with strides.
- We convolve the filter and the slicing window to work out the neurons on the feature map.

- $W_i, H_i$ : The width and height of the input image
- $W_o, H_o$ : The width and height of the output image (feature map)

$$W_o = \left\lceil \frac{W_i-1}{s_w} \right\rceil + 1 \text{ and } H_o = \left\lceil \frac{H_i-1}{s_h} \right\rceil + 1$$

- Our case:  $W_o = \left\lceil \frac{7-1}{2} \right\rceil + 1 = 4$  and  $H_o = \left\lceil \frac{7-1}{2} \right\rceil + 1 = 4$



(Source: <https://towardsdatascience.com/>)

# Convolution layer without zero padding

$x$  (input tensor (7,8))    strides = (2,2)

1	-2	-1	5	3	2	1	1
1	3	-1	4	3	3	1	-1
1	-2	1	6	3	3	2	2
2	-2	2	5	2	1	0	-1
0	3	-2	5	4	1	2	1
1	2	-3	1	1	2	-1	2
1	-2	-1	1	2	1	1	3

$W_i = 8$  **No Zero padding**

In TF or Keras, **padding = 'valid'**

$H_i = 7$

kernel size = (3,3)

$W$  (filter or kernel)

1	1	1
1	1	1
1	1	1

$f_w = 3$

$f_h = 3$

feature map

1	23	21
3	26	18
-1	8	13

$H_o = 3$

$W_o = 3$

- $W_i, H_i$ : The width and height of the **input** image
- $W_o, H_o$ : The width and height of the **output** image (feature map)

$$W_o = \left\lceil \frac{W_i - f_w}{s_w} \right\rceil + 1 \text{ and } H_o = \left\lceil \frac{H_i - f_h}{s_h} \right\rceil + 1$$

$$\text{Our case: } W_o = \left\lceil \frac{8-3}{2} \right\rceil + 1 = 3 \text{ and } H_o = \left\lceil \frac{7-3}{2} \right\rceil + 1 = 3$$

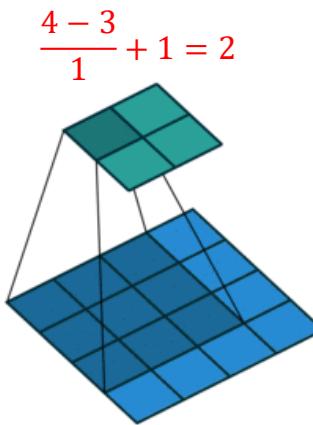
3 <sub>0</sub>	3 <sub>1</sub>	2 <sub>2</sub>	1	0
0 <sub>2</sub>	0 <sub>2</sub>	1 <sub>0</sub>	3	1
3 <sub>0</sub>	1 <sub>1</sub>	2 <sub>2</sub>	2	3
2	0	0	2	2
2	0	0	0	1

12	12	17
10	17	19
9	6	14

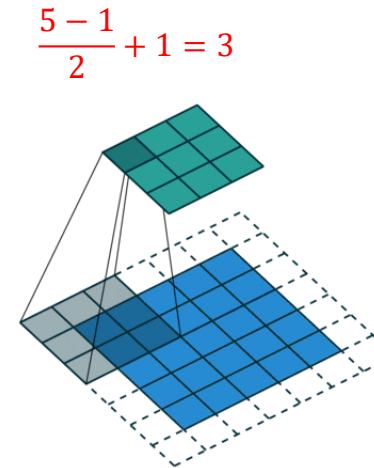
(Source: Internet)

# Convolution layer

## Example



$s=1, p=0, f=3$



$s=2, p=1, f=3$

**s** = stride, **p** = padding, **f** = filter size (square)

(Source: [An intuitive guide to Convolutional Neural Networks](#))

- $W_i, H_i$ : The width and height of the **input** image
- $W_o, H_o$ : The width and height of the **output** image

- **With zeros padding**

$$W_o = \left\lfloor \frac{W_i - 1}{s_w} \right\rfloor + 1 \text{ and } H_o = \left\lfloor \frac{H_i - 1}{s_h} \right\rfloor + 1$$

- **Without zeros padding**

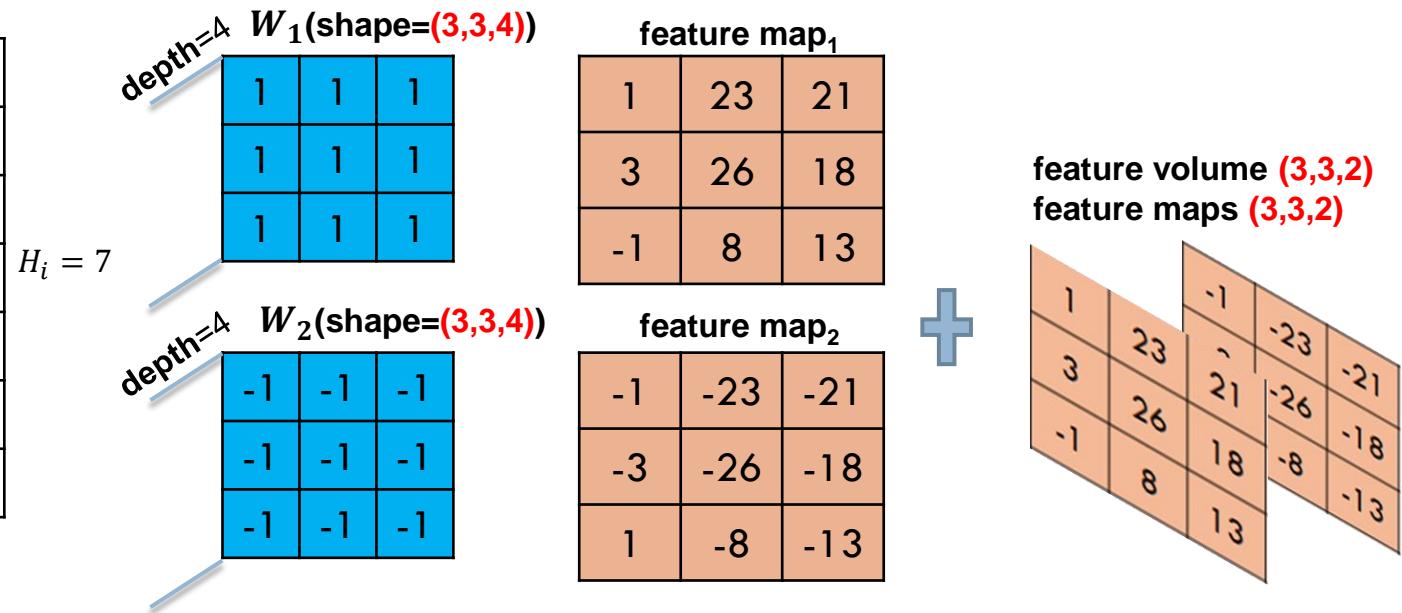
$$W_o = \left\lfloor \frac{W_i - f_w}{s_w} \right\rfloor + 1 \text{ and } H_o = \left\lfloor \frac{H_i - f_h}{s_h} \right\rfloor + 1$$

# Convolution layer with multiple filters and feature maps

$x$  (input tensor (7,8,4)) strides = (2,2)

depth=4	1	-2	-1	5	3	2	1	1
	1	3	-1	4	3	3	1	-1
	1	-2	1	6	3	3	2	2
	2	-2	2	5	2	1	0	-1
	0	3	-2	5	4	1	2	1
	1	2	-3	1	1	2	-1	2
	1	-2	-1	1	2	1	1	3

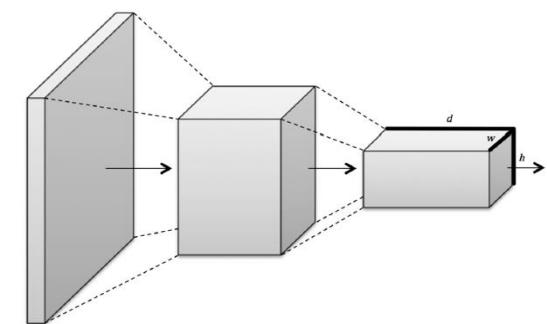
$W_i = 8$



```
feature_maps = tf.nn.conv2d(inputs, filters, strides=[1,2,2,1], padding="VALID")
```

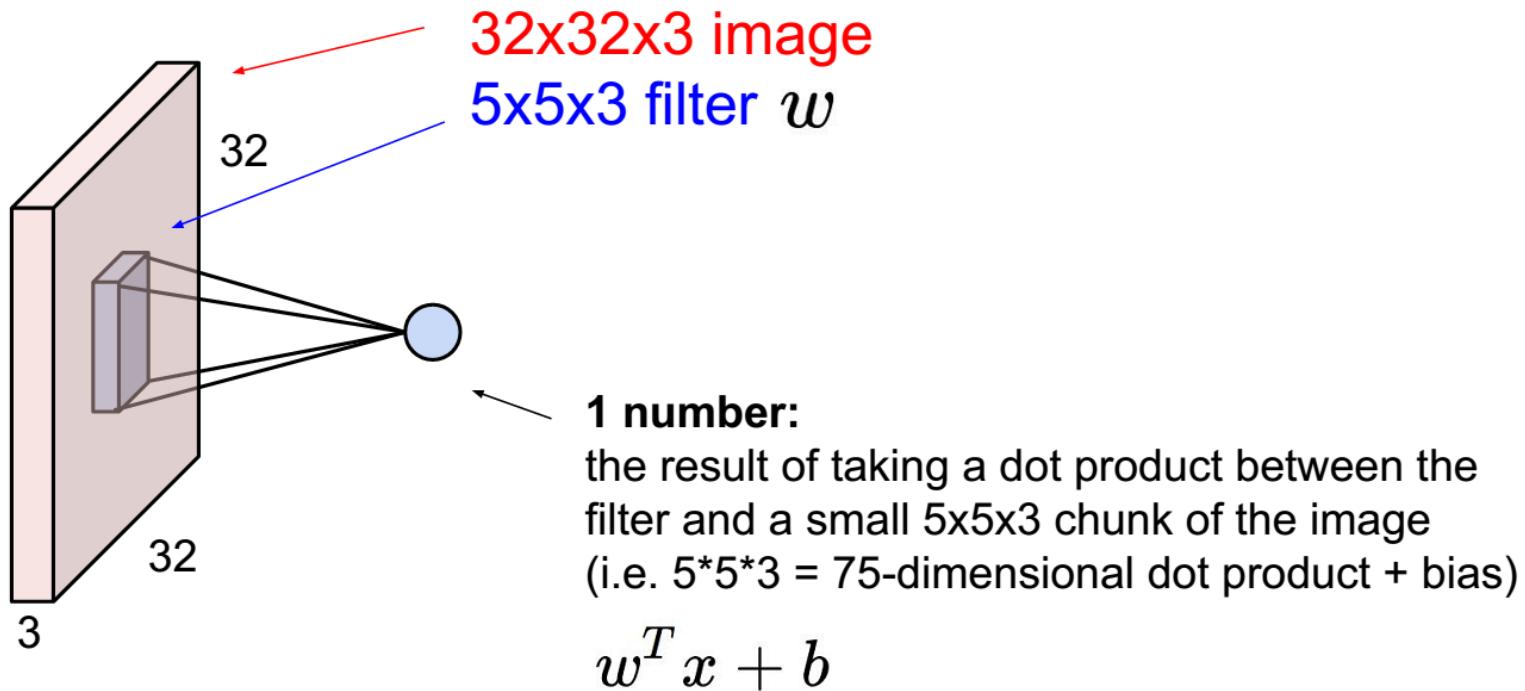
batch\_size, in\_height,  
in\_width, in\_channels

fil\_height, fil\_width,  
in\_channels, out\_channels

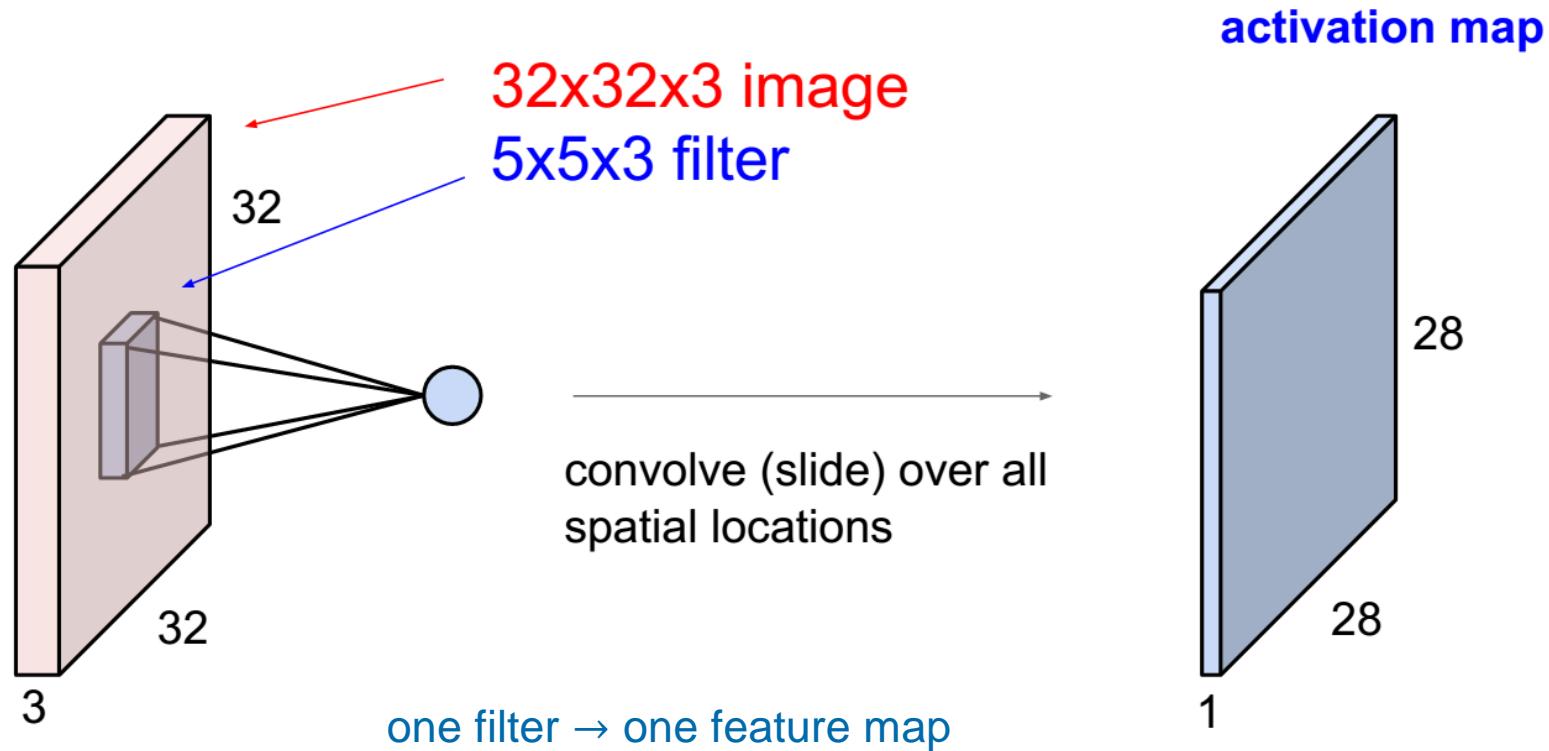


# Convolution layer: 3D convolution

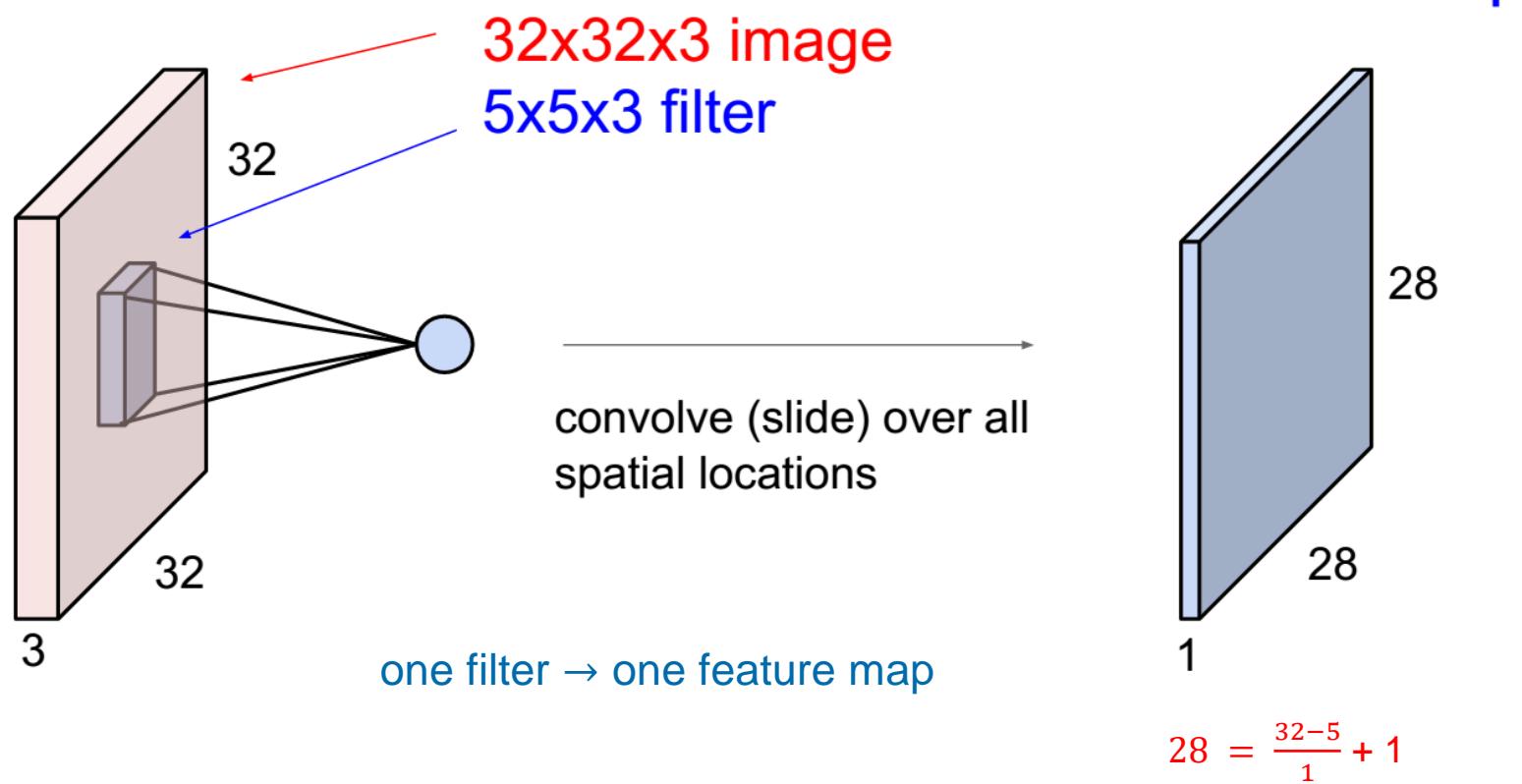
## Example



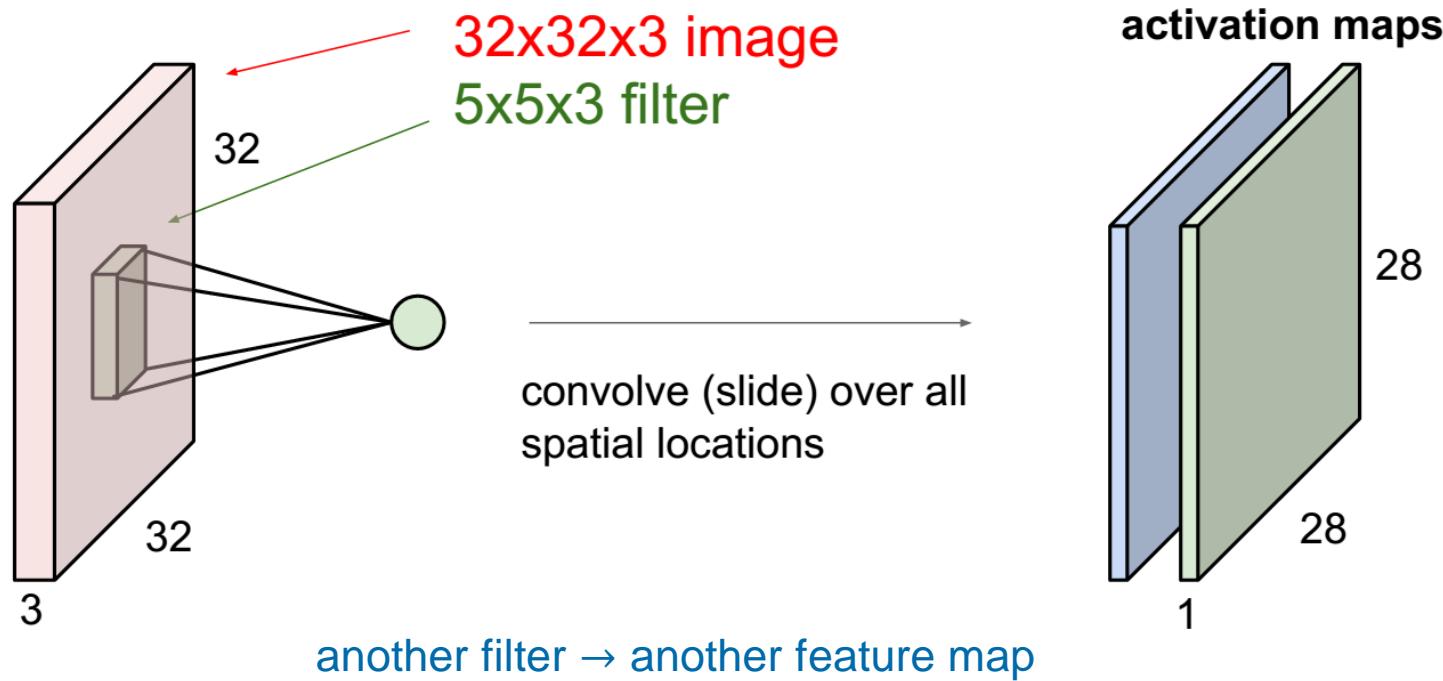
# Convolution layer: one feature map



# Convolution layer: one feature map

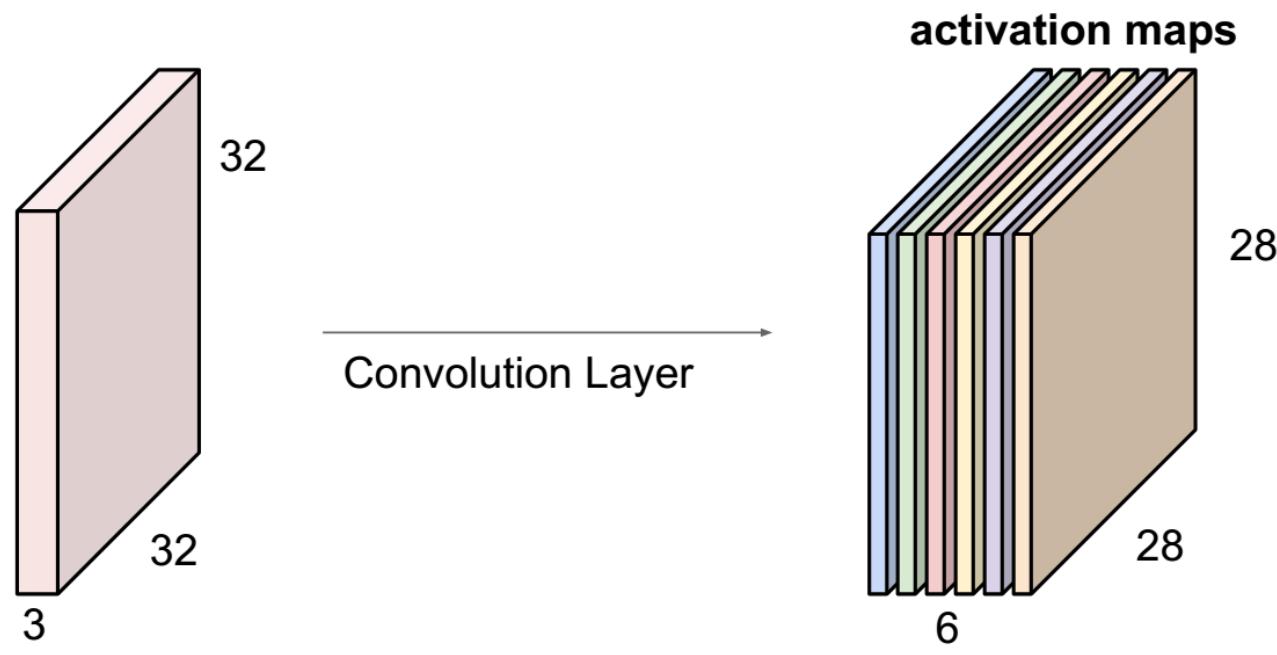


# Convolution layer: two feature maps



# Convolution layer: multiple feature maps

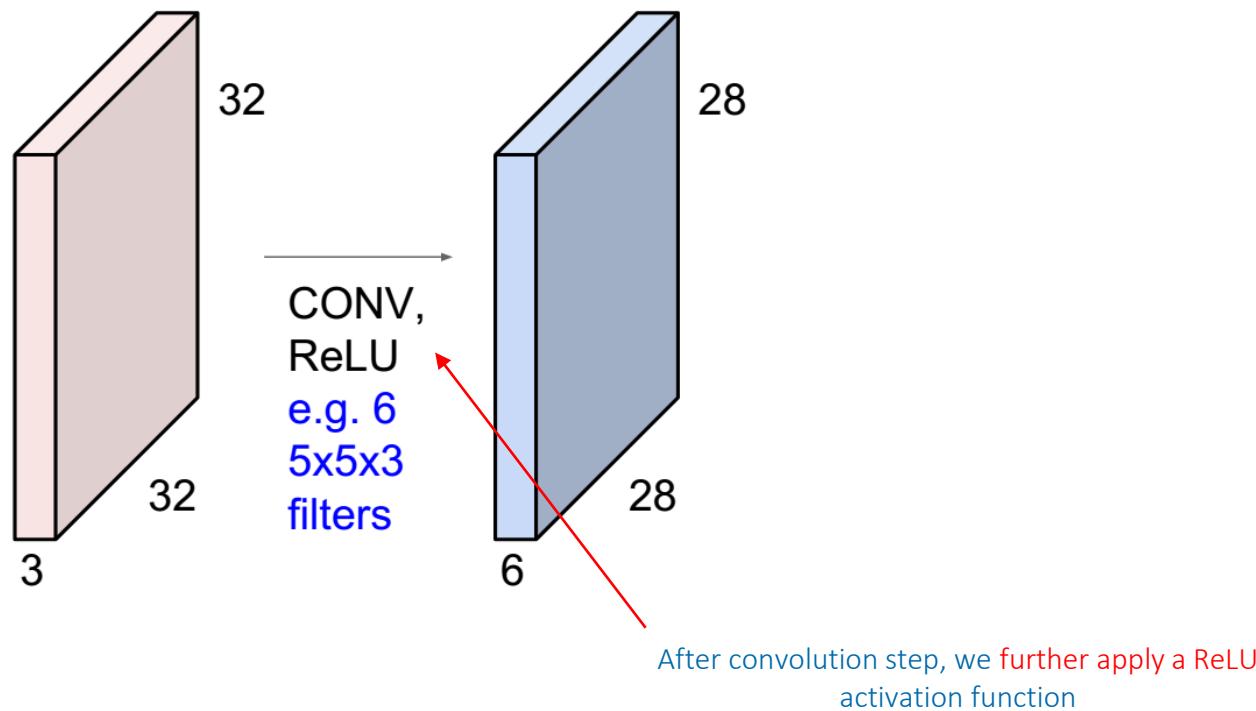
Example



stacking up feature maps to form a new 3D volume = ‘new’ image

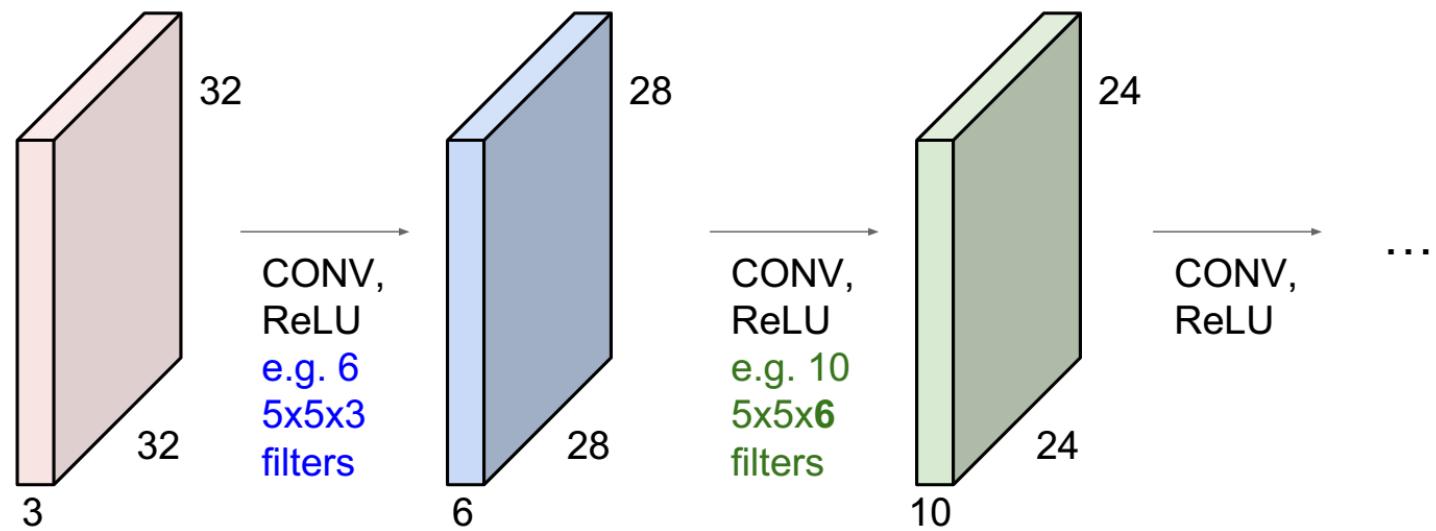
# Convolution Layer with activation function

**Preview:** ConvNet is a sequence of Convolution Layers, interspersed with activation functions



# Convolution Layer with activation function

**Preview:** ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



# Example of convolutional operation with TF 2.x

```

import numpy as np
from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg")[80:360, 70:390]
flower = load_sample_image("flower.jpg")[80:360, 130:450]
batch = np.array([china, flower], dtype=np.float32)
print(batch.shape)

(2, 280, 320, 3)

```

```

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, :, 0] = 1 # vertical line, why?
filters[3, :, :, 1] = 1 # horizontal line, why?

```

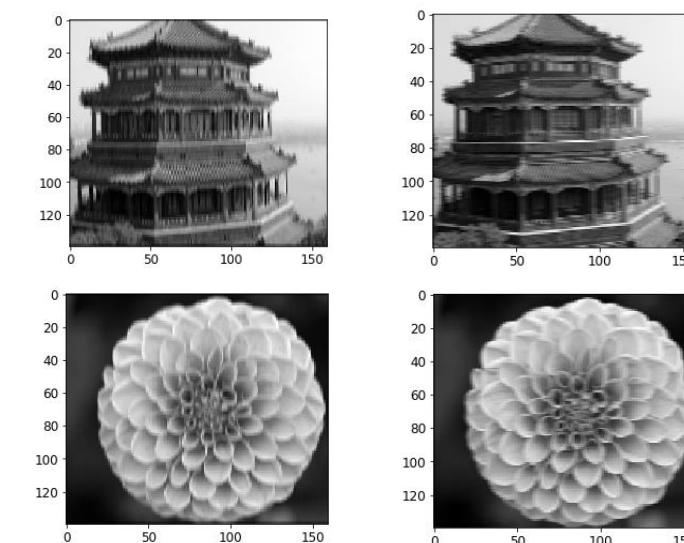
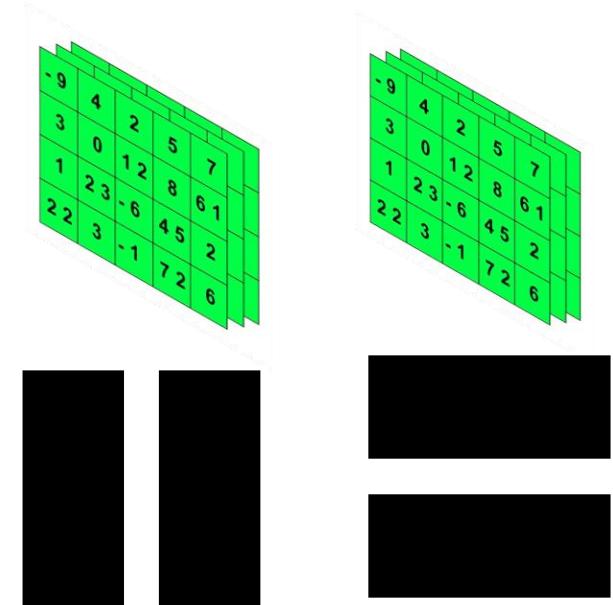
```

output = tf.nn.conv2d(batch, filters, strides=[1,2,2,1], padding="SAME")
print("Output shape:" + str(output.shape))
plt.imshow(output[0, :, :, 0], cmap='gray') # plot 1st image's 1nd feature map, channel 0
plt.show()
plt.imshow(output[0, :, :, 1], cmap='gray') # plot 1st image's 2nd feature map, channel 1
plt.show()

plt.imshow(output[1, :, :, 0], cmap='gray') # plot 2nd image's 1nd feature map, channel 0
plt.show()
plt.imshow(output[1, :, :, 1], cmap='gray') # plot 2nd image's 2nd feature map, channel 1
plt.show()

Output shape:(2, 140, 160, 2)

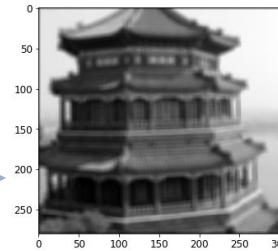
```



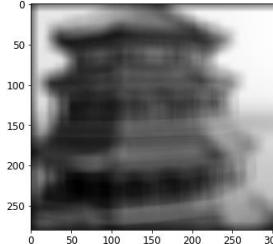
# Effects of filters/kernels to images



```
smallBlur = np.ones((7, 7), dtype="float") * (1.0 / (7 * 7))
```



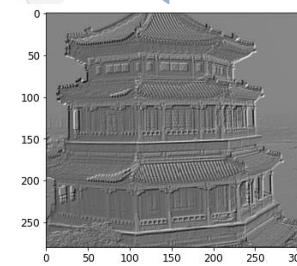
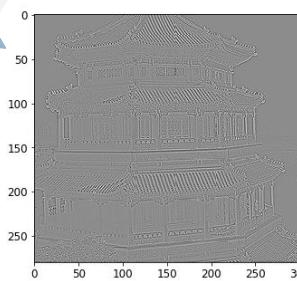
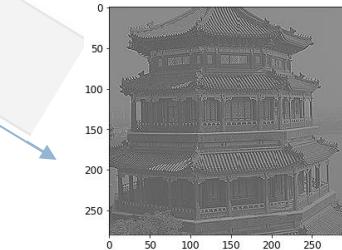
```
largeBlur = np.ones((21, 21), dtype="float") * (1.0 / (21 * 21))
```



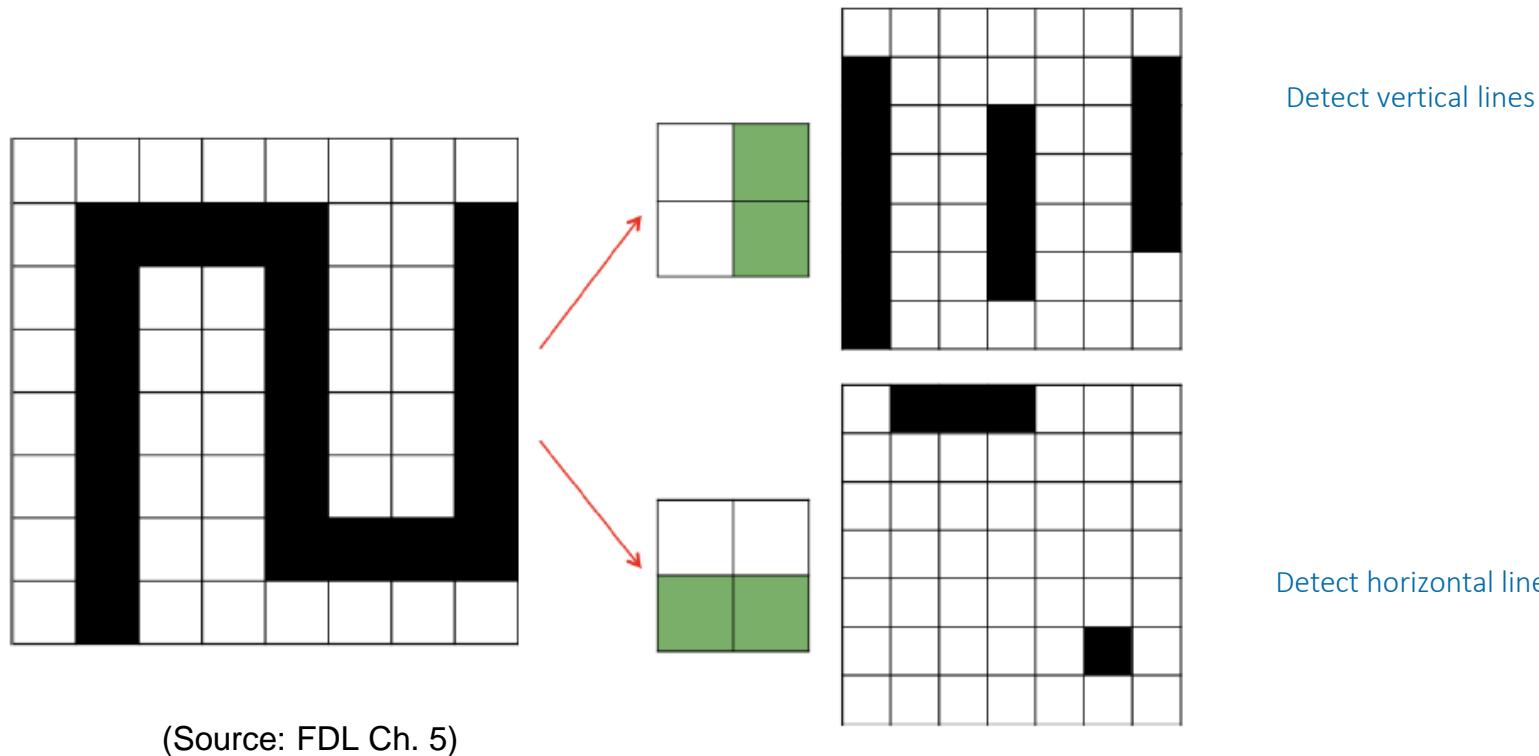
```
# construct a sharpening filter
sharpen = np.array([[0, -1, 0],
                   [-1, 5, -1],
                   [0, -1, 0]], dtype="int")
```

```
# construct the Laplacian kernel used to detect edge-like
# regions of an image
laplacian = np.array([[0, 1, 0],
                      [1, -4, 1],
                      [0, 1, 0]], dtype="int")
```

```
# construct an emboss kernel
emboss = np.array([[-2, -1, 0],
                  [-1, 1, 1],
                  [0, 1, 2]], dtype="int")
```



# Convolution – 2D filter



# Summary: full convolutional layer

- Input is a volume, e.g., an RGB image:

- Width  $W_i$
- Height  $H_i$
- Depth  $D$
- Zero padding

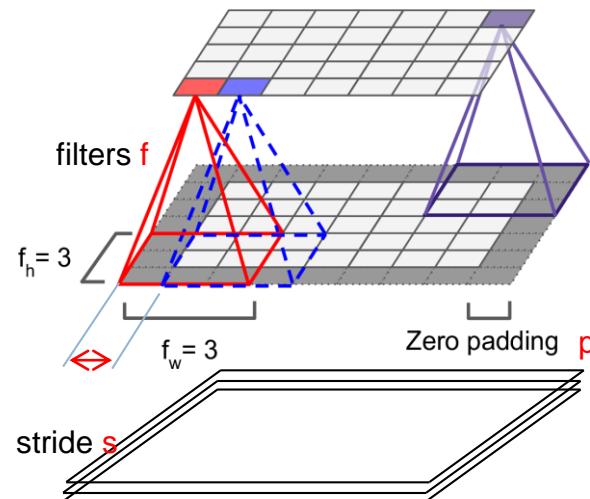
- Output is another volume

- Width  $W_o$
- Height  $H_o$
- Depth  $K$  (= number of filters)

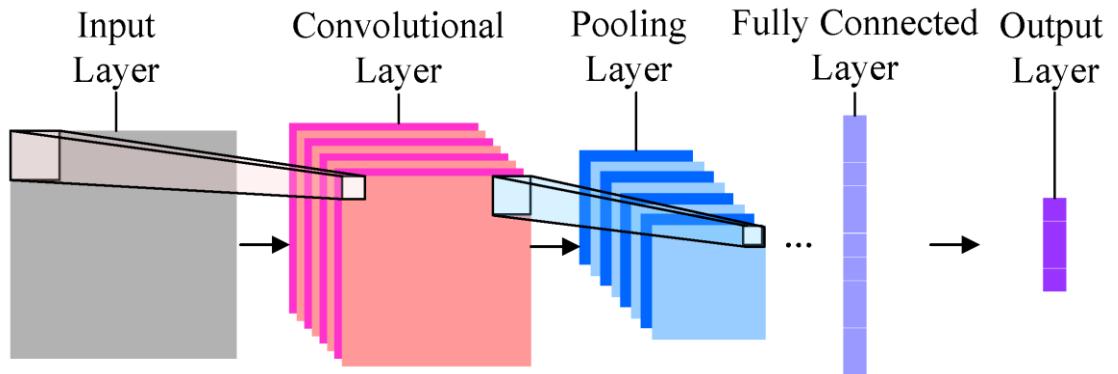
- $W_i, H_i$ : The width and height of the input image

- $W_o, H_o$ : The width and height of the output image

$$W_o = \left\lceil \frac{W_i - 1}{s} \right\rceil + 1 \text{ and } H_o = \left\lceil \frac{H_i - 1}{s} \right\rceil + 1$$



# General architecture of CNNs



1. Convolutional layer
2. **Pooling layer**
3. Fully connected layer

# Pooling layer

# Pooling operation

- Makes the representations smaller and more manageable
- Subsample the image
- Operates over each activation map independently

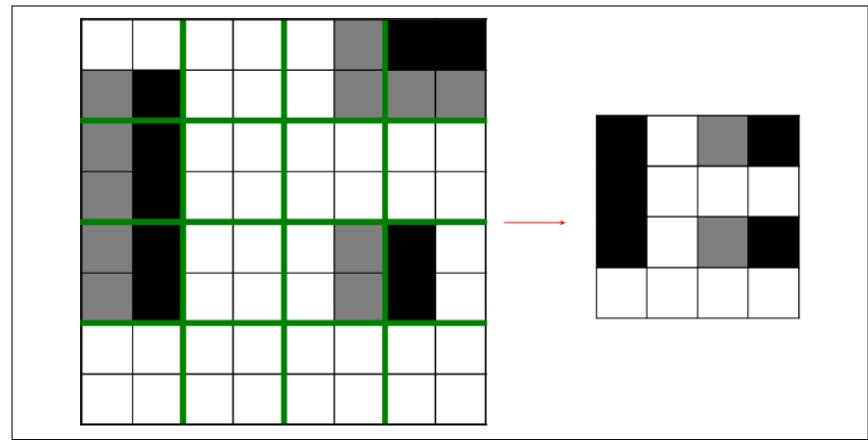


Figure 5-13. An illustration of how max pooling significantly reduces parameters as we move up the network

(Source: FDL Ch. 5)

# Max/average pooling

- Max/average-pooling is **locally invariant** and applied **independently to each feature map**
- In practice, we don't overlap,  $s=f$

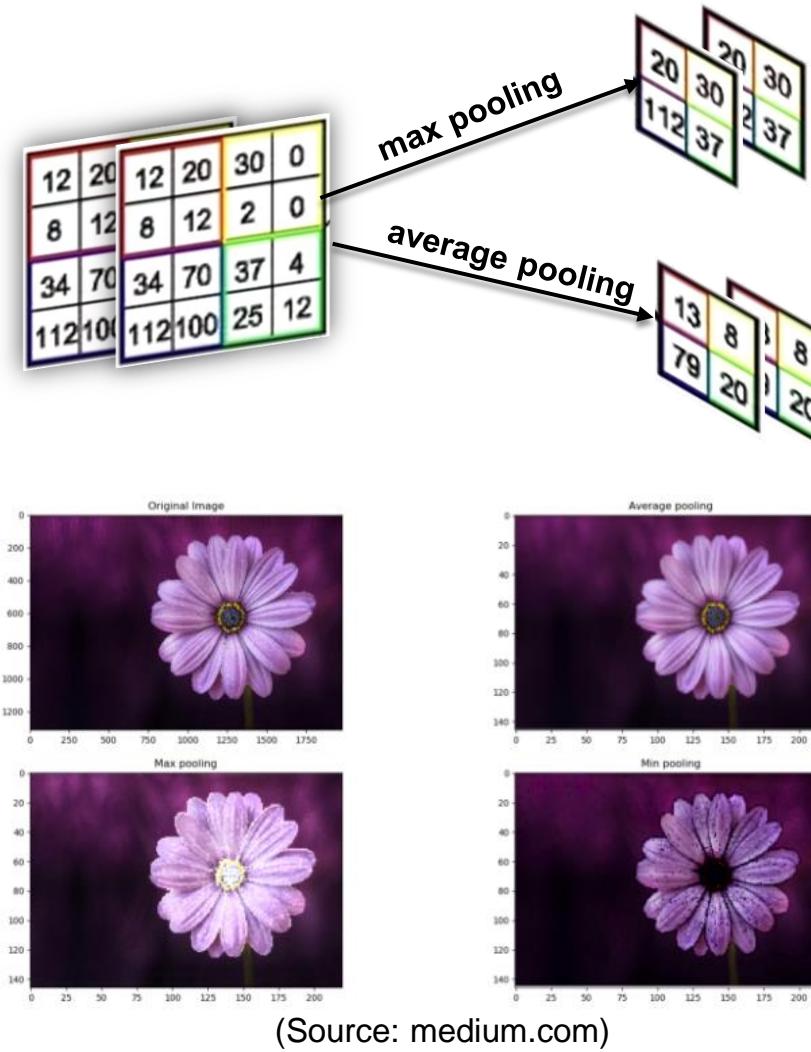
- $W_o = \left\lfloor \frac{W_i-1}{s} \right\rfloor + 1$
  - $H_o = \left\lfloor \frac{H_i-1}{s} \right\rfloor + 1$

- In the SOTA CNNs (ResNet, GoogleLeNet, VGG), we employ the pooling layer with

- Kernel/filter size = (2,2) and strides = (2,2)
  - Therefore, the output tensor has the size

- $W_o = \left\lfloor \frac{224-1}{2} \right\rfloor + 1 = 112$
    - $H_o = \left\lfloor \frac{224-1}{2} \right\rfloor + 1 = 112$

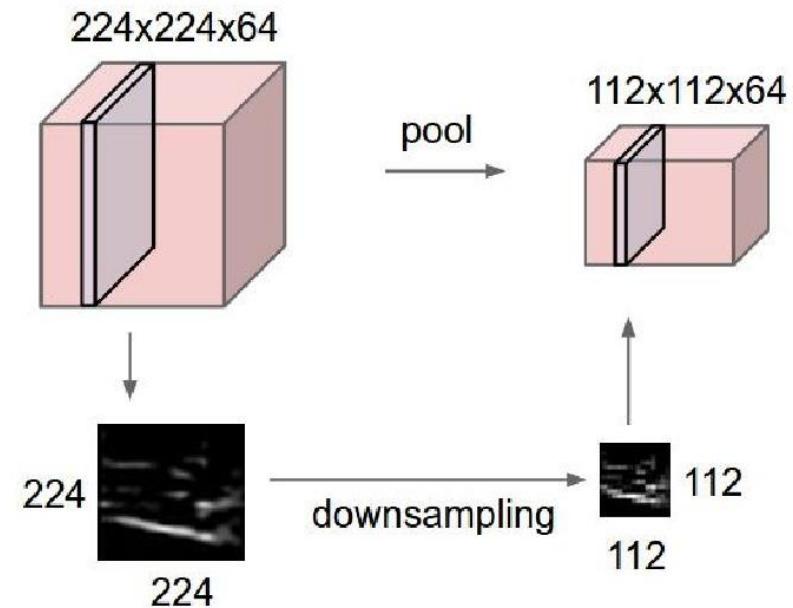
- With this setting, we **down-sample the input size by 2**



(Source: medium.com)

# Pooling operation

- Makes the representations smaller and more manageable
- Operates over each activation map independently



# Example of pooling with TF 2.x

## Max pooling

```
output = tf.nn.max_pool(input = batch, ksize=(2,2), strides=(2,2), padding="SAME")
output = output.numpy() #convert Eager-Execution Tensor object to numpy array
print(batch[0].shape)
print(output[0].shape)
plot_actual_image(batch[0]) #plot the 1st original image
plot_actual_image(output[0]) # plot the output for the 1st image

(280, 320, 3)
(140, 160, 3)
```



## Average pooling

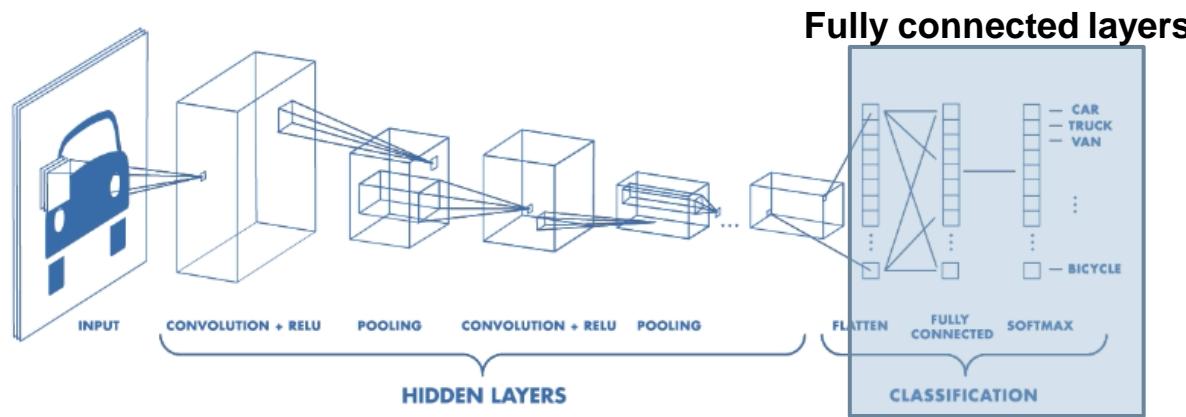
```
output = tf.nn.avg_pool(input = batch, ksize=(2,2), strides=(2,2), padding="SAME")
output = output.numpy() #convert Eager-Execution Tensor object to numpy array
print(batch[0].shape)
print(output[0].shape)
plot_actual_image(batch[0]) #plot the 1st original image
plot_actual_image(output[0]) # plot the output for the 1st image

(280, 320, 3)
(140, 160, 3)
```



# Fully connected layer

# Fully connected layer



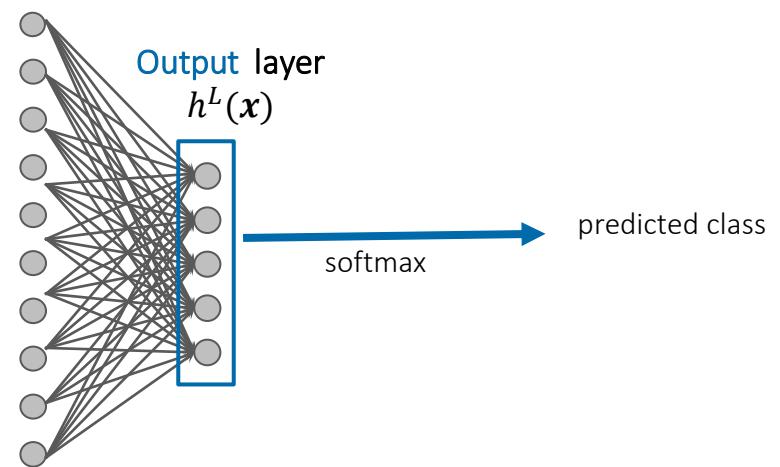
- The last tensor is **flattened** and **some fully connected layers** are added to classify the input.
  - The last tensor  $[5,5,10] \rightarrow$  1 layer with  $5 \times 5 \times 10 = 250$  neurons

# What is a softmax layer?

## Classification setting

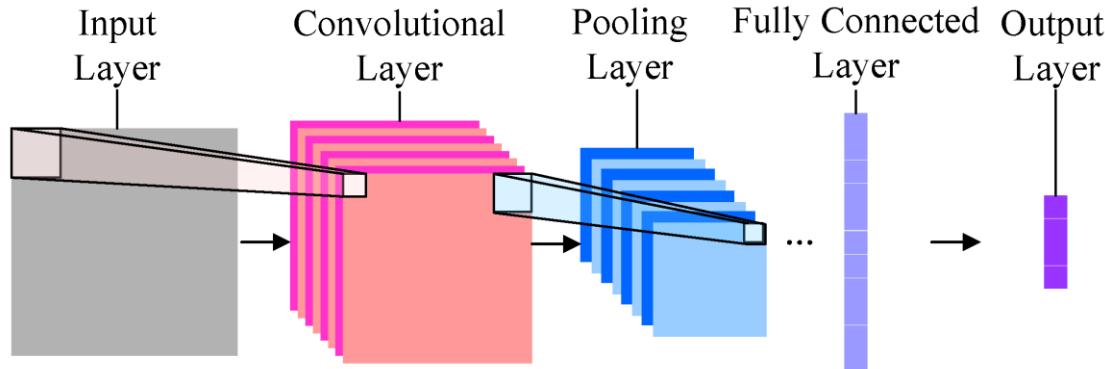
- **Input:** vector of dimension  $M$ 
  - e.g.,  $h = [1, 2, 3]$ ,  $M = 3$
- **Output:** a discrete distribution of dimension  $M$ 
  - e.g.,  $y = [0.09, 0.24, 0.67]$
- How to calculate prediction probability  $p$ ?

$$[p_1, \dots, p_i, \dots, p_M] = \text{softmax}(h) = \frac{\exp(h_i)}{\sum_j \exp(h_j)}$$

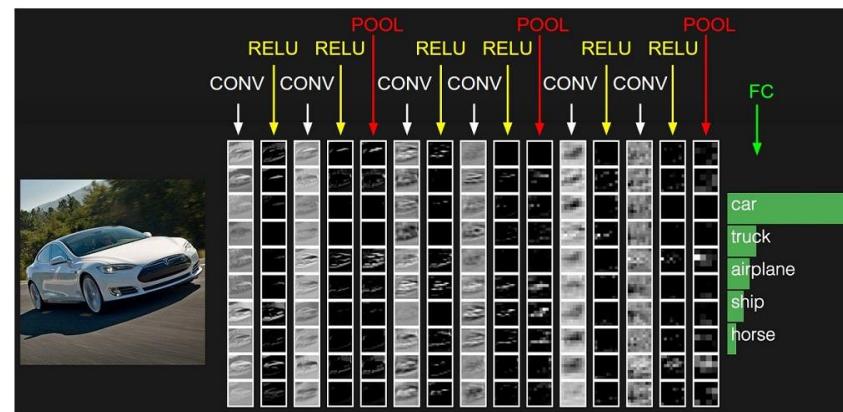


Put them all together

# General architecture of CNNs



- Convolution stage
  - Pooling stage
  - Fully connected stage

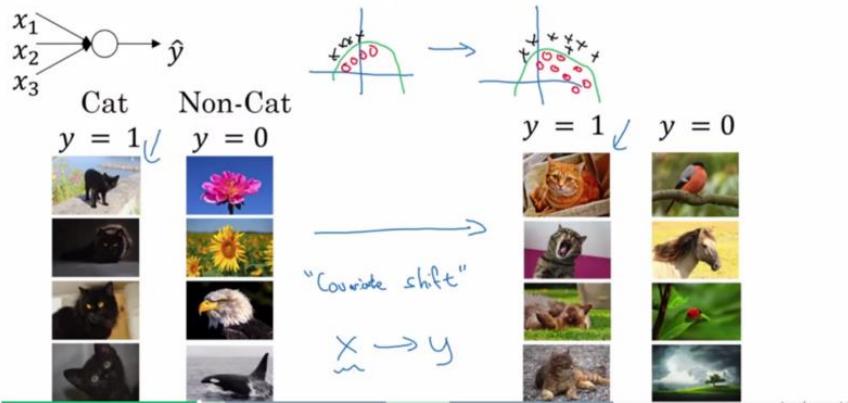


*“CS231n: Convolutional Neural Networks for Visual Recognition”, Stanford, 2016*

# Batch Normalization Layer

# Covariate shift vs internal covariate shift

## Covariate shift problem

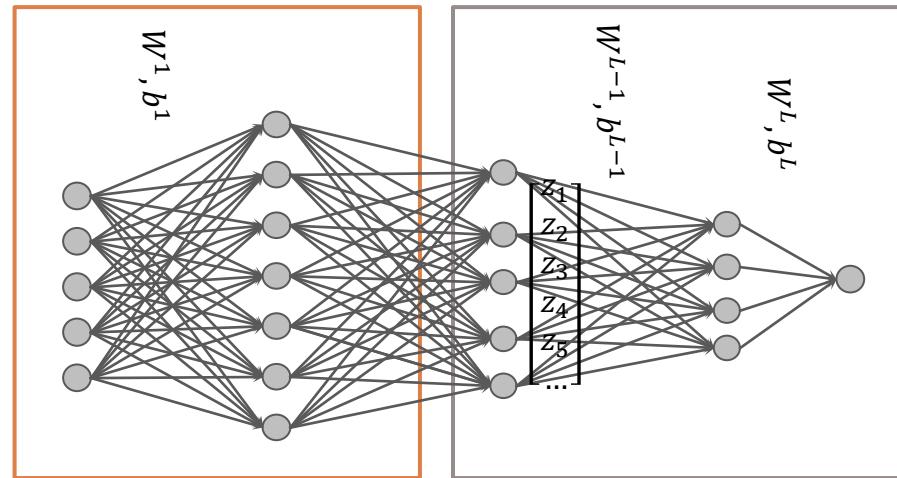


(Source: DL course Andrew Ng)

### Covariate shift problem

- The **distribution (nature) of training data** is different from **that of testing data**.

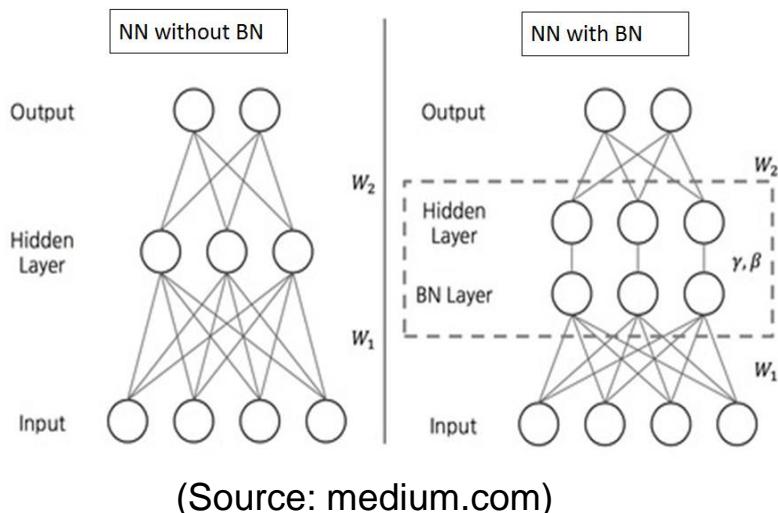
## Internal covariate shift



- ❑ The **distribution** of  $[z_1, z_2, \dots, z_5]$  changes due to
  - ❑ The **updates** of  $[W^1, b^1], [W^2, b^2]$
  - ❑ The **changes** in mini-batches

# Batch Normalization

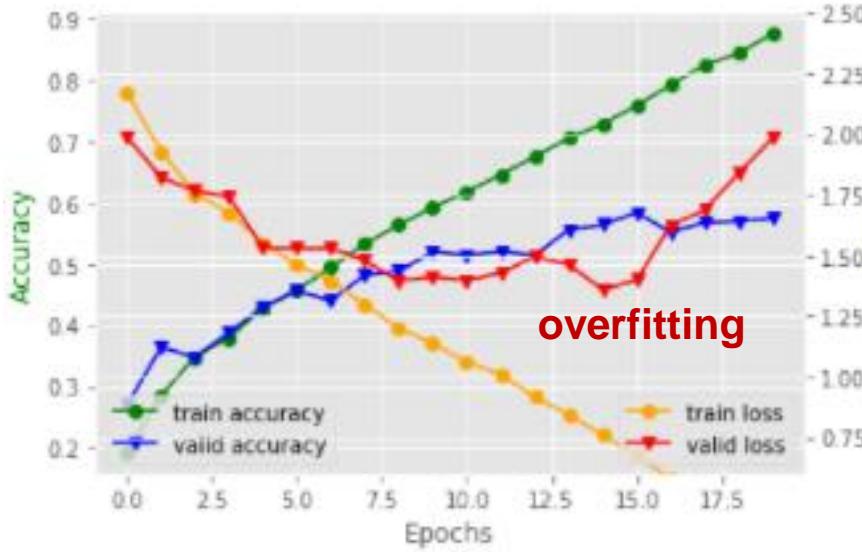
- Cope with **internal covariate shift**
- Reduce **gradient vanishing/exploding**
- Reduce **overfitting**
- Make training **more stable**
- Converge **faster**
  - Allow us to train with bigger learning rate



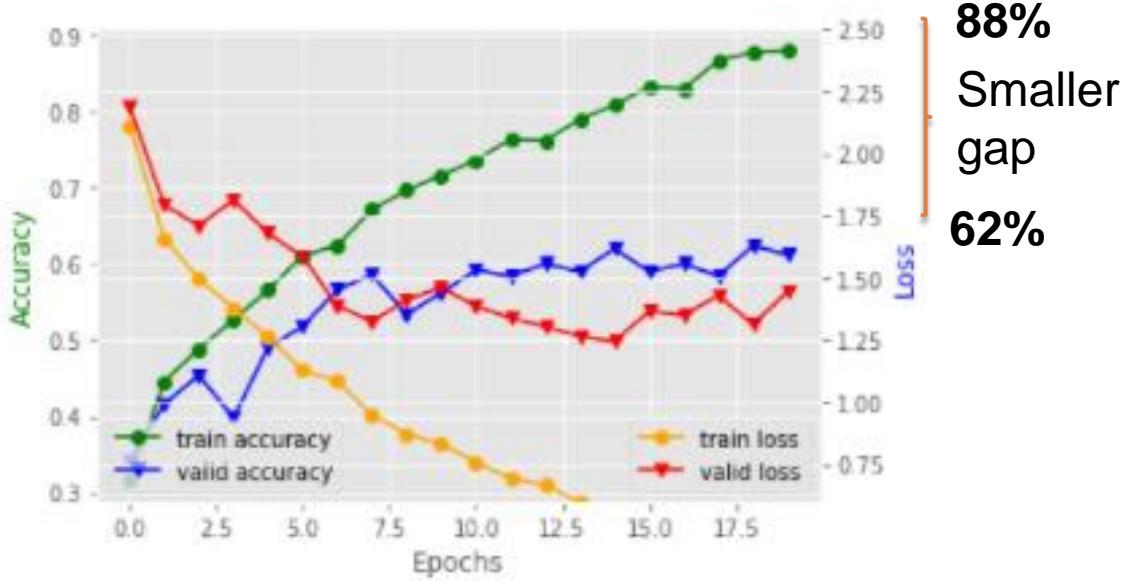
- ❖ Let  $z = W^k h^k + b^k$  be the mini-batch before activation. We compute the normalized  $\hat{z}$  as
  - ❖  $\hat{z} = \frac{z - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$  where  $\epsilon$  is a small value such as  $1e^{-7}$
  - ❖  $\mu_B = \frac{1}{b} \sum_{i=1}^b z_i$  is the empirical mean
  - ❖  $\sigma_B^2 = \frac{1}{b} \sum_{i=1}^b (z_i - \mu_B)^2$  is the empirical variance
- ❖ We scale the normalized  $\hat{z}$ 
  - ❖  $z_{BN} = \gamma \hat{z} + \beta$  where  $\gamma, \beta > 0$  are two learnable parameters (i.e., scale and shift parameters)
- ❖ We then apply the activation to obtain the next layer value
  - ❖  $h^{k+1} = \sigma(z_{BN})$

- ✓ At testing time, we replace the mini-batch  $\mu_B$  and  $\sigma_B$  with running averages of  $\tilde{\mu}_B$  and  $\tilde{\sigma}_B$  computed during the training process.
  - ✓  $\tilde{\mu}_B = (1 - \theta)\tilde{\mu}_B + \theta\mu_B$
  - ✓  $\tilde{\sigma}_B = (1 - \theta)\tilde{\sigma}_B + \theta\sigma_B$
  - ✓  $0 < \theta < 1$  is the momentum decay

# Training with versus without batch norm



```
Epoch 1/20
63/63 [=====] - 12s 191ms/step - loss: 2.1712 - accuracy: 0.1912 - val_loss: 1.9883 - val_accuracy: 0.2700
Epoch 2/20
63/63 [=====] - 12s 186ms/step - loss: 1.9299 - accuracy: 0.2853 - val_loss: 1.8219 - val_accuracy: 0.3640
Epoch 3/20
63/63 [=====] - 12s 188ms/step - loss: 1.7560 - accuracy: 0.3485 - val_loss: 1.7670 - val_accuracy: 0.3480
Epoch 4/20
63/63 [=====] - 12s 186ms/step - loss: 1.6772 - accuracy: 0.3798 - val_loss: 1.7426 - val_accuracy: 0.3880
Epoch 5/20
63/63 [=====] - 12s 185ms/step - loss: 1.5529 - accuracy: 0.4277 - val_loss: 1.5336 - val_accuracy: 0.4300
Epoch 6/20
63/63 [=====] - 12s 193ms/step - loss: 1.4617 - accuracy: 0.4575 - val_loss: 1.5300 - val_accuracy: 0.4560
Epoch 7/20
63/63 [=====] - 12s 193ms/step - loss: 1.3966 - accuracy: 0.4942 - val_loss: 1.5295 - val_accuracy: 0.4400
Epoch 8/20
63/63 [=====] - 12s 192ms/step - loss: 1.2965 - accuracy: 0.5330 - val_loss: 1.4800 - val_accuracy: 0.4820
Epoch 9/20
63/63 [=====] - 12s 193ms/step - loss: 1.2024 - accuracy: 0.5638 - val_loss: 1.3958 - val_accuracy: 0.4880
Epoch 10/20
63/63 [=====] - 12s 192ms/step - loss: 1.1372 - accuracy: 0.5925 - val_loss: 1.4123 - val_accuracy: 0.5200
Epoch 11/20
63/63 [=====] - 12s 194ms/step - loss: 1.0619 - accuracy: 0.6177 - val_loss: 1.3976 - val_accuracy: 0.5140
Epoch 12/20
63/63 [=====] - 13s 199ms/step - loss: 1.0103 - accuracy: 0.6460 - val_loss: 1.4282 - val_accuracy: 0.5200
Epoch 13/20
63/63 [=====] - 12s 193ms/step - loss: 0.9173 - accuracy: 0.6752 - val_loss: 1.4955 - val_accuracy: 0.5140
Epoch 14/20
63/63 [=====] - 12s 197ms/step - loss: 0.8444 - accuracy: 0.7055 - val_loss: 1.4631 - val_accuracy: 0.5560
Epoch 15/20
63/63 [=====] - 12s 192ms/step - loss: 0.7609 - accuracy: 0.7287 - val_loss: 1.3603 - val_accuracy: 0.5640
Epoch 16/20
63/63 [=====] - 12s 192ms/step - loss: 0.6788 - accuracy: 0.7588 - val_loss: 1.3981 - val_accuracy: 0.5840
Epoch 17/20
63/63 [=====] - 13s 199ms/step - loss: 0.5783 - accuracy: 0.7922 - val_loss: 1.6249 - val_accuracy: 0.5520
Epoch 18/20
63/63 [=====] - 11s 181ms/step - loss: 0.5094 - accuracy: 0.8253 - val_loss: 1.6868 - val_accuracy: 0.5680
Epoch 19/20
63/63 [=====] - 13s 200ms/step - loss: 0.4471 - accuracy: 0.8443 - val_loss: 1.8391 - val_accuracy: 0.5700
Epoch 20/20
63/63 [=====] - 12s 193ms/step - loss: 0.3486 - accuracy: 0.8760 - val_loss: 1.9838 - val_accuracy: 0.5740
```

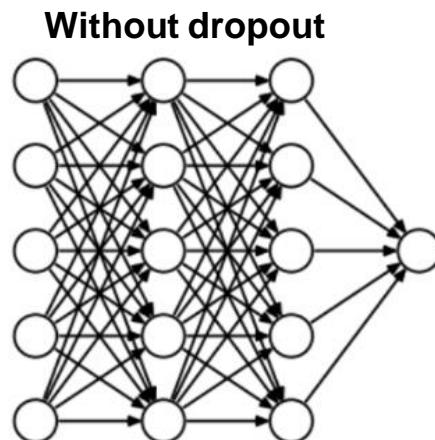


```
Epoch 1/20
63/63 [=====] - 21s 336ms/step - loss: 2.1120 - accuracy: 0.3162 - val_loss: 2.1910 - val_accuracy: 0.3420
Epoch 2/20
63/63 [=====] - 21s 330ms/step - loss: 1.6594 - accuracy: 0.4455 - val_loss: 1.7928 - val_accuracy: 0.4160
Epoch 3/20
63/63 [=====] - 21s 332ms/step - loss: 1.4950 - accuracy: 0.4882 - val_loss: 1.7103 - val_accuracy: 0.4540
Epoch 4/20
63/63 [=====] - 21s 339ms/step - loss: 1.3785 - accuracy: 0.5268 - val_loss: 1.8133 - val_accuracy: 0.3980
Epoch 5/20
63/63 [=====] - 21s 334ms/step - loss: 1.2618 - accuracy: 0.5658 - val_loss: 1.6789 - val_accuracy: 0.4900
Epoch 6/20
63/63 [=====] - 21s 341ms/step - loss: 1.1282 - accuracy: 0.6108 - val_loss: 1.5839 - val_accuracy: 0.5180
Epoch 7/20
63/63 [=====] - 21s 330ms/step - loss: 1.0834 - accuracy: 0.6242 - val_loss: 1.3882 - val_accuracy: 0.5660
Epoch 8/20
63/63 [=====] - 21s 329ms/step - loss: 0.9509 - accuracy: 0.6712 - val_loss: 1.3235 - val_accuracy: 0.5860
Epoch 9/20
63/63 [=====] - 21s 330ms/step - loss: 0.8708 - accuracy: 0.6967 - val_loss: 1.4123 - val_accuracy: 0.5340
Epoch 10/20
63/63 [=====] - 21s 331ms/step - loss: 0.8319 - accuracy: 0.7160 - val_loss: 1.4604 - val_accuracy: 0.5620
Epoch 11/20
63/63 [=====] - 21s 330ms/step - loss: 0.7582 - accuracy: 0.7365 - val_loss: 1.3860 - val_accuracy: 0.5920
Epoch 12/20
63/63 [=====] - 21s 331ms/step - loss: 0.6939 - accuracy: 0.7638 - val_loss: 1.3378 - val_accuracy: 0.5840
Epoch 13/20
63/63 [=====] - 21s 330ms/step - loss: 0.6677 - accuracy: 0.7623 - val_loss: 1.3038 - val_accuracy: 0.6000
Epoch 14/20
63/63 [=====] - 21s 331ms/step - loss: 0.5946 - accuracy: 0.7895 - val_loss: 1.2651 - val_accuracy: 0.5900
Epoch 15/20
63/63 [=====] - 21s 329ms/step - loss: 0.5466 - accuracy: 0.8087 - val_loss: 1.2441 - val_accuracy: 0.6200
Epoch 16/20
63/63 [=====] - 21s 332ms/step - loss: 0.4855 - accuracy: 0.8330 - val_loss: 1.3652 - val_accuracy: 0.5900
Epoch 17/20
63/63 [=====] - 21s 330ms/step - loss: 0.4734 - accuracy: 0.8300 - val_loss: 1.3503 - val_accuracy: 0.6000
Epoch 18/20
63/63 [=====] - 22s 347ms/step - loss: 0.3872 - accuracy: 0.8670 - val_loss: 1.4286 - val_accuracy: 0.5840
Epoch 19/20
63/63 [=====] - 22s 344ms/step - loss: 0.3438 - accuracy: 0.8783 - val_loss: 1.3120 - val_accuracy: 0.6240
Epoch 20/20
63/63 [=====] - 22s 357ms/step - loss: 0.3572 - accuracy: 0.8800 - val_loss: 1.4453 - val_accuracy: 0.6120
```

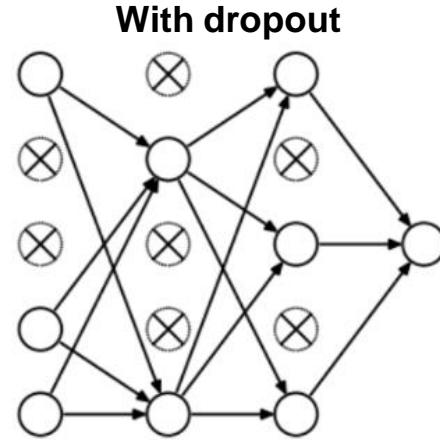
# Dropout Layer

# Dropout

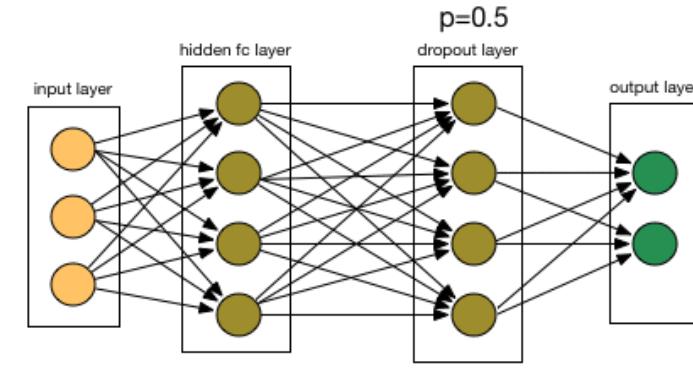
## Reduce Overfitting



**Without dropout**



**With dropout**



**Training time**

(Source: chatbotslife)

- This is a **cheap technique** to reduce model capacity
  - Reduce overfitting
- In each iteration, at each layer, **randomly choose** some neurons and **drop all connections** from these neurons
  - `dropout_rate = 1 - keep_prob`

# Dropout

## Reduce Overfitting

- Computationally efficient
- Can be considered a **bagged ensemble** of an exponential number ( $2^N$ ) of neural networks.
- **Training**

$$\mathbf{r} \sim \text{Bernoulli}(\mu)$$

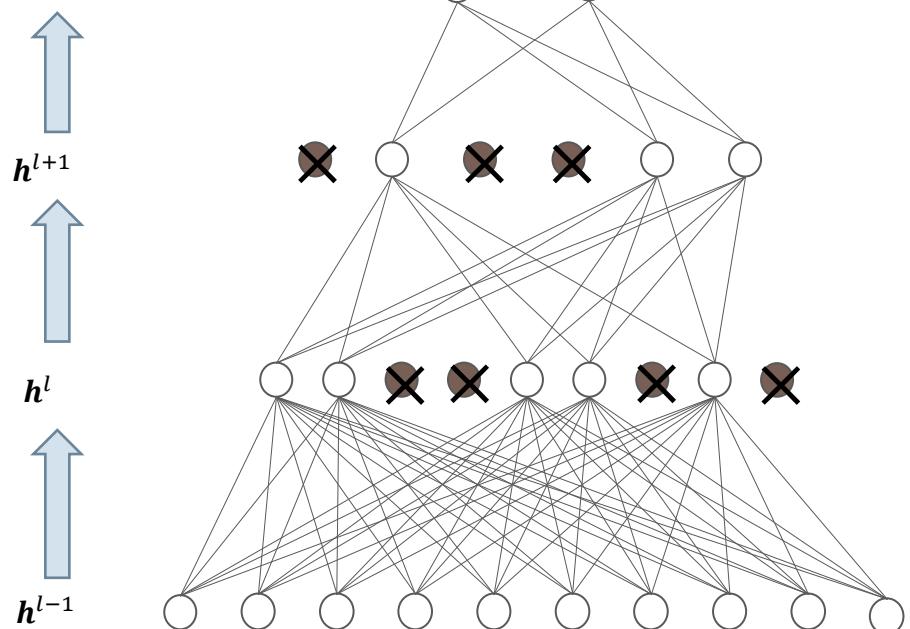
$$\tilde{\mathbf{h}}^l = \mathbf{h}^l \odot \mathbf{r}$$

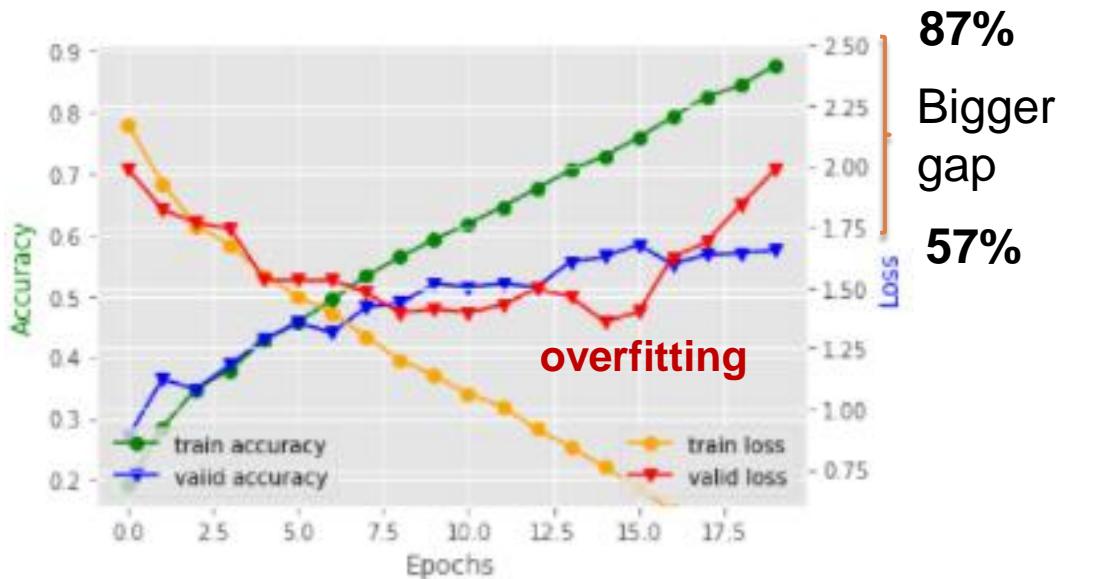
$$\mathbf{h}^{l+1} = \sigma(W^{(l)\top} \tilde{\mathbf{h}}^l + \mathbf{b}^l)$$

- **Testing**

$$\tilde{\mathbf{h}}^l = \mathbf{h}^l \times (1 - \mu)$$

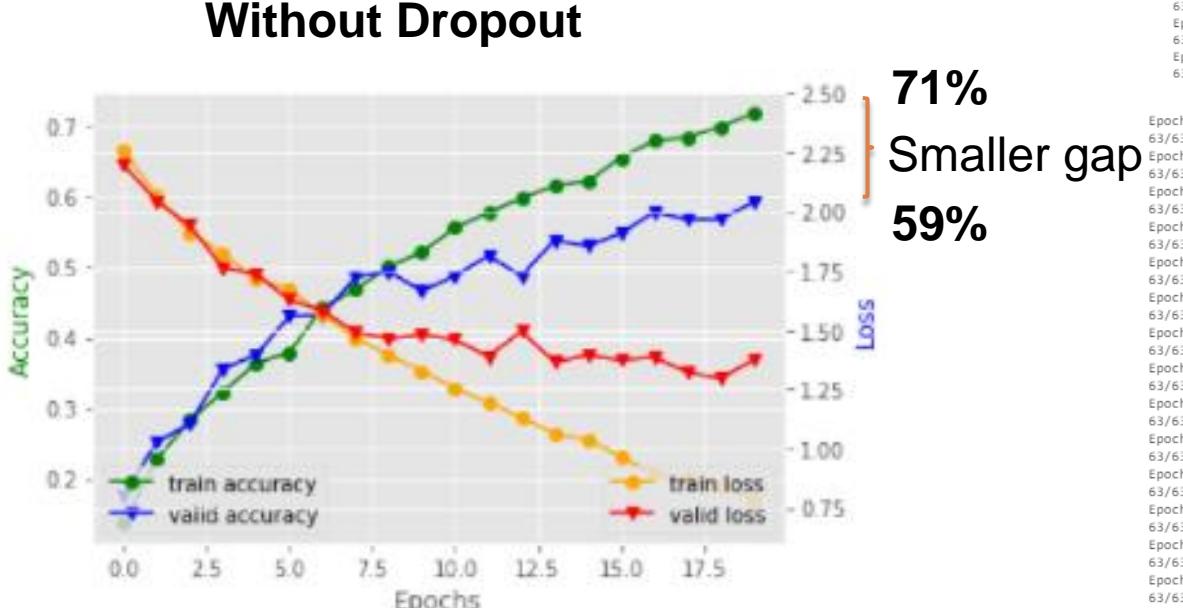
$$\mathbf{h}^{l+1} = \sigma(W^{l\top} \tilde{\mathbf{h}}^l + \mathbf{b}^l)$$





87%  
Bigger gap  
57%

overfitting



71%  
Smaller gap  
59%

```
Epoch 1/20
63/63 [=====] - 12s 191ms/step - loss: 2.1712 - accuracy: 0.1912 - val_loss: 1.9883 - val_accuracy: 0.2700
Epoch 2/20
63/63 [=====] - 12s 186ms/step - loss: 1.9299 - accuracy: 0.2853 - val_loss: 1.8219 - val_accuracy: 0.3640
Epoch 3/20
63/63 [=====] - 12s 188ms/step - loss: 1.7560 - accuracy: 0.3485 - val_loss: 1.7670 - val_accuracy: 0.3480
Epoch 4/20
63/63 [=====] - 12s 186ms/step - loss: 1.6772 - accuracy: 0.3798 - val_loss: 1.7426 - val_accuracy: 0.3880
Epoch 5/20
63/63 [=====] - 12s 185ms/step - loss: 1.5529 - accuracy: 0.4277 - val_loss: 1.5336 - val_accuracy: 0.4300
Epoch 6/20
63/63 [=====] - 12s 193ms/step - loss: 1.4617 - accuracy: 0.4575 - val_loss: 1.5300 - val_accuracy: 0.4560
Epoch 7/20
63/63 [=====] - 12s 193ms/step - loss: 1.3966 - accuracy: 0.4942 - val_loss: 1.5295 - val_accuracy: 0.4400
Epoch 8/20
63/63 [=====] - 12s 192ms/step - loss: 1.2965 - accuracy: 0.5330 - val_loss: 1.4800 - val_accuracy: 0.4820
Epoch 9/20
63/63 [=====] - 12s 193ms/step - loss: 1.2024 - accuracy: 0.5638 - val_loss: 1.3958 - val_accuracy: 0.4880
Epoch 10/20
63/63 [=====] - 12s 192ms/step - loss: 1.1372 - accuracy: 0.5925 - val_loss: 1.4123 - val_accuracy: 0.5200
Epoch 11/20
63/63 [=====] - 12s 194ms/step - loss: 1.0619 - accuracy: 0.6177 - val_loss: 1.3976 - val_accuracy: 0.5140
Epoch 12/20
63/63 [=====] - 13s 199ms/step - loss: 1.0103 - accuracy: 0.6460 - val_loss: 1.4282 - val_accuracy: 0.5200
Epoch 13/20
63/63 [=====] - 12s 193ms/step - loss: 0.9173 - accuracy: 0.6752 - val_loss: 1.4955 - val_accuracy: 0.5140
Epoch 14/20
63/63 [=====] - 12s 197ms/step - loss: 0.8444 - accuracy: 0.7055 - val_loss: 1.4631 - val_accuracy: 0.5560
Epoch 15/20
63/63 [=====] - 12s 192ms/step - loss: 0.7609 - accuracy: 0.7287 - val_loss: 1.3603 - val_accuracy: 0.5640
Epoch 16/20
63/63 [=====] - 12s 192ms/step - loss: 0.6788 - accuracy: 0.7588 - val_loss: 1.3981 - val_accuracy: 0.5840
Epoch 17/20
63/63 [=====] - 13s 199ms/step - loss: 0.5783 - accuracy: 0.7922 - val_loss: 1.6249 - val_accuracy: 0.5520
Epoch 18/20
63/63 [=====] - 11s 181ms/step - loss: 0.5094 - accuracy: 0.8253 - val_loss: 1.6868 - val_accuracy: 0.5680
Epoch 19/20
63/63 [=====] - 13s 200ms/step - loss: 0.4471 - accuracy: 0.8443 - val_loss: 1.8391 - val_accuracy: 0.5700
Epoch 20/20
63/63 [=====] - 12s 193ms/step - loss: 0.3486 - accuracy: 0.8760 - val_loss: 1.9838 - val_accuracy: 0.5740
```

# Dropout

## Real-world Example

## Advantage and Rational of CNNs

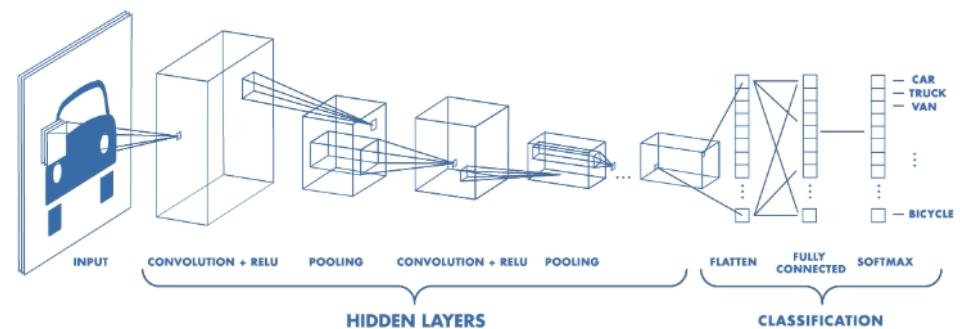
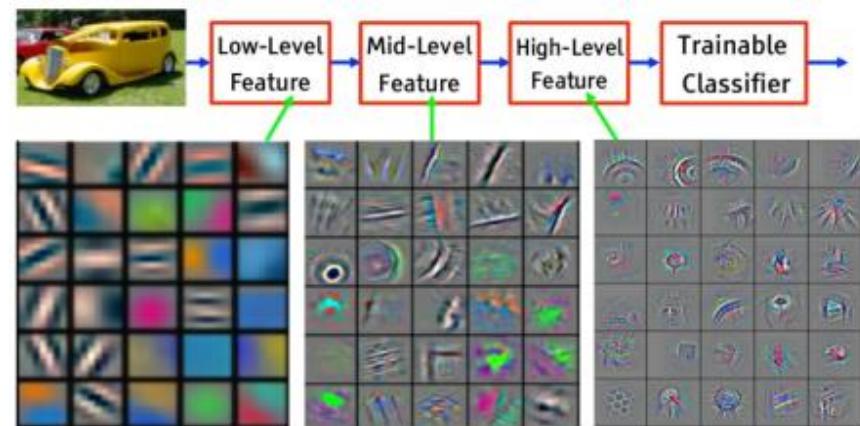
# CNN scales by reducing #params

- Reduces the number of learnable weights
- Deep layer indirectly interacts with a larger portion of input.
- The weight matrix is shared across the kernel/filter.



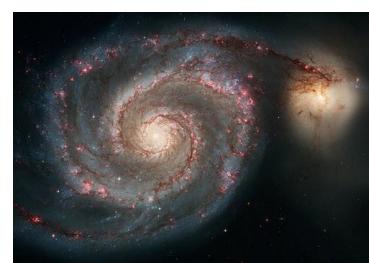
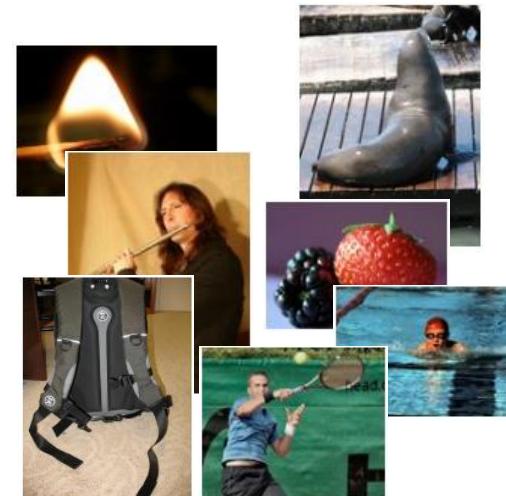
# Convolution neural networks

- **Motivation:** vision processing in the brain is fast
  - Simple cells detect local features
  - Complex cells pool local features
- **Technical:** sparse interactions and sparse weights reduce #params.
  - Parameter sharing: using a kernel with same set of weights while applying onto different location
  - Translation invariance.
- Directly extract meaningful/useful features
  - Automatic feature extraction and do not need hand-crafted features
  - **Low-level features**: edges, pixels, corners,
  - **Mid-level features**: circles, triangles, boxes,
  - **High-level features**: cats, dogs, objects

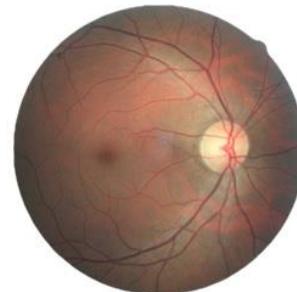


# Factors for successful CNNs

- ❑ ConvNets won most, if not all, recent computer vision challenges!
  
- ❑ Larger models with new training techniques:
  - ❑ Dropout, Maxout, Batchnorm, Maxnorm, and etc.
  
- ❑ Large ImageNet dataset [Fei-Fei et al. 2012]
  - ❑ 1.2 million training samples
  - ❑ 1000 categories
  
- ❑ Fast graphical processing units (GPU, TPU)
  - ❑ Capable of 1 trillion operations per second



Galaxy Image  
Challenge



Diabetes recognition  
from retina image



# Datasets for CNNs

# Some datasets for computer vision



## MNIST dataset

#training: 50,000, #testing: 10,000, #classes: 10



## Fashion MNIST dataset

#training: 50,000, #testing: 10,000, #classes: 10



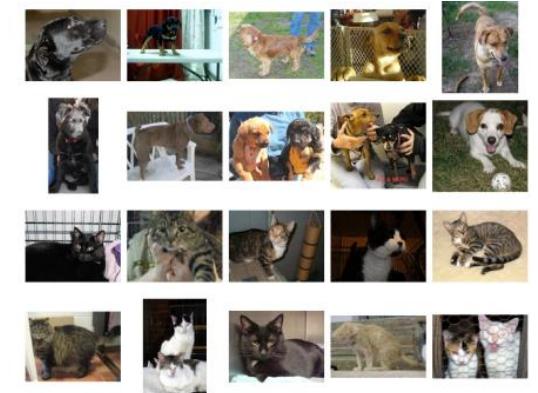
## CIFAR 10 dataset

#examples: 60,000, #classes: 10



## Smile 10 dataset

#examples: 13,165, #classes: 2



## Animals: Cat, Dog dataset Kaggle competition

#examples: 25,000, #classes: 2



## Flower-17 dataset

#examples: 1,360, #classes: 17

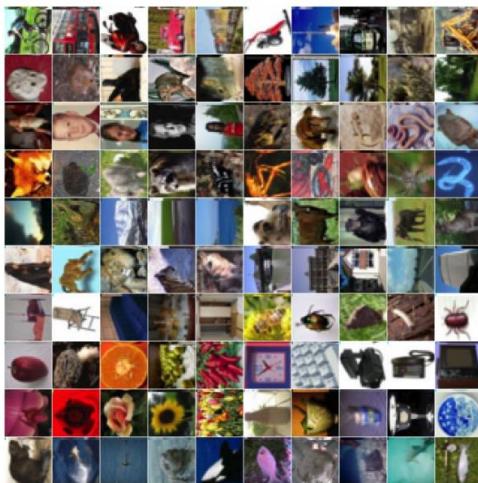
# Some datasets for computer vision



**Adience dataset for age and gender recognition**

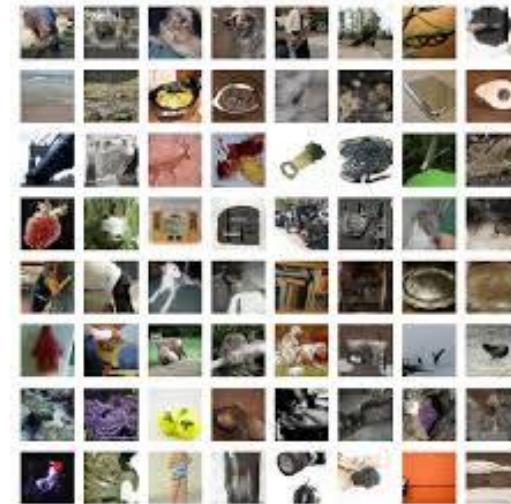
Age: 0 → 60+

#examples: 26,580



**CIFAR 100 dataset**

100 classes, each class has 500 images for training and 100 images for testing



**Tiny ImageNet 200 dataset**

A subset of ImageNet data set with size (64,64,3)  
200 classes, each class has 500 for training, 50 for validating, and 50 for testing



**CALTECH 101 dataset for object recognition**  
8,677 images includes 101 categories



**Kaggle: Facial Expression Recognition Challenge**  
35,888 images with 6 expressions

# Golden dataset: ImageNet



## Large-scale dataset (ImageNet)

- over 1.2 million images and 1,000 possible object categories

printer housing animal weight drop egg white  
 offspring teacher computer headquarters television  
 register measure album garage dorm flower  
 gallery court key structure light date spread breakfast  
 king horse fireplace church press market lighter  
 restaurant counter road paper side site door pack  
 hotel screen coast tree file concert coffee  
 sport tower means fan hill can camp fish bathroom  
 sky plant wine fox house school railcar  
 bread table top man car gun study bird  
 weapon cover cloud range leash van suite mirror seat  
 spring fruit dog descent net menu ball flash button  
 range leash van suite mirror seat  
 bed shop train camera kit roll goal  
 kitchen engine box tea overall center step  
 dinner stone boat memory sieve cell kid barwatch  
 apple girl flat chain boat stand  
 flagbank cross chair inine castle home room student  
 radio valley castle office rule hall  
 beach support level line street club  
 base library stage video food building  
 tool material player leg shirt desk security call clock  
 football hospital match equipment cell phone mountain crowd  
 short circuit bridge scale gas pedal microphone recording telephone

# IM<sup>A</sup>GENET Large Scale Visual Recognition Challenge (ILSVRC)

## Competition

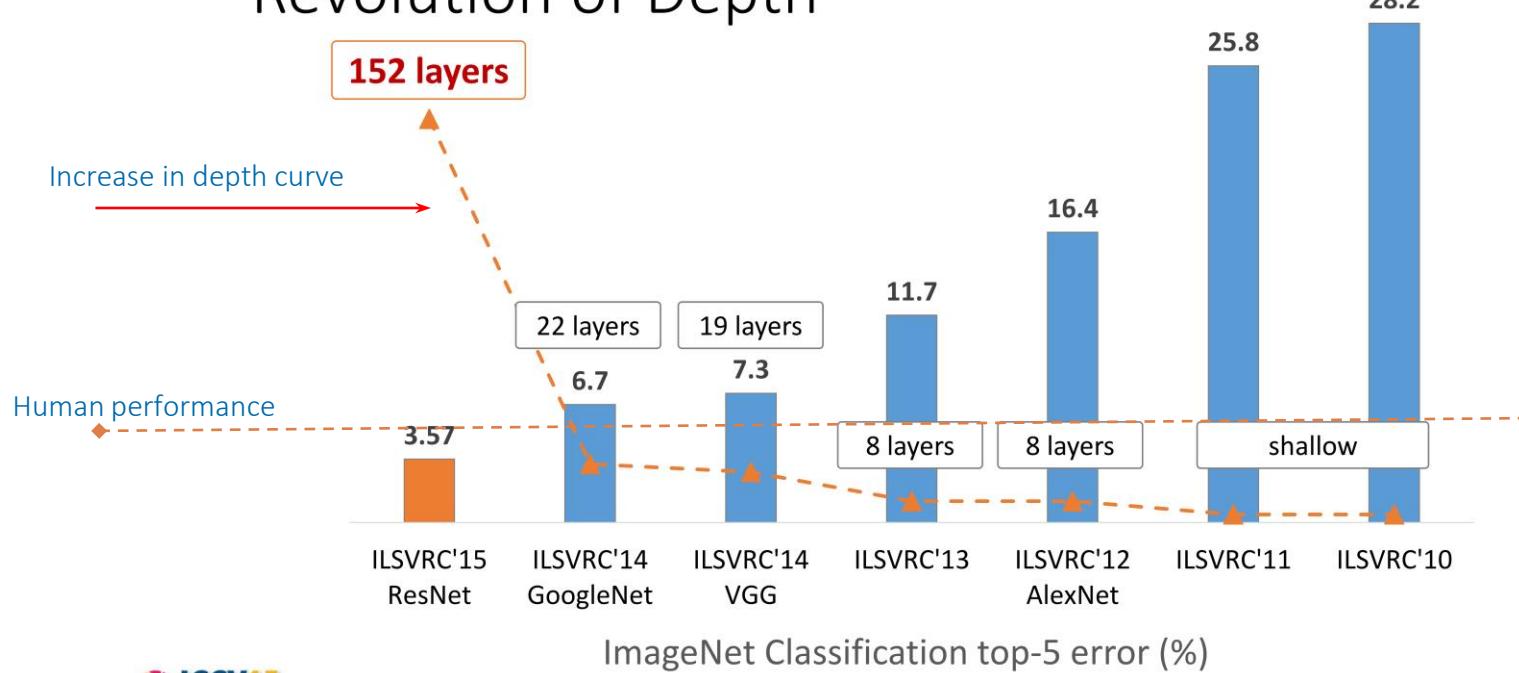
The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) evaluates algorithms for object detection and image classification at large scale. One high level motivation is to allow researchers to compare progress in detection across a wider variety of objects -- taking advantage of the quite expensive labeling effort. Another motivation is to measure the progress of computer vision for large scale image indexing for retrieval and annotation.

## Introduction

This challenge evaluates algorithms for object localization/detection from images/videos at scale. Most successful and innovative teams will be invited to present at [CVPR 2017 workshop](#).

- I. [Object localization](#) for 1000 categories.
- II. [Object detection](#) for 200 fully labeled categories.
- III. [Object detection from video](#) for 30 fully labeled categories.

## Revolution of Depth



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

And ... ResNet 2015 beats human performance (5.1%)!



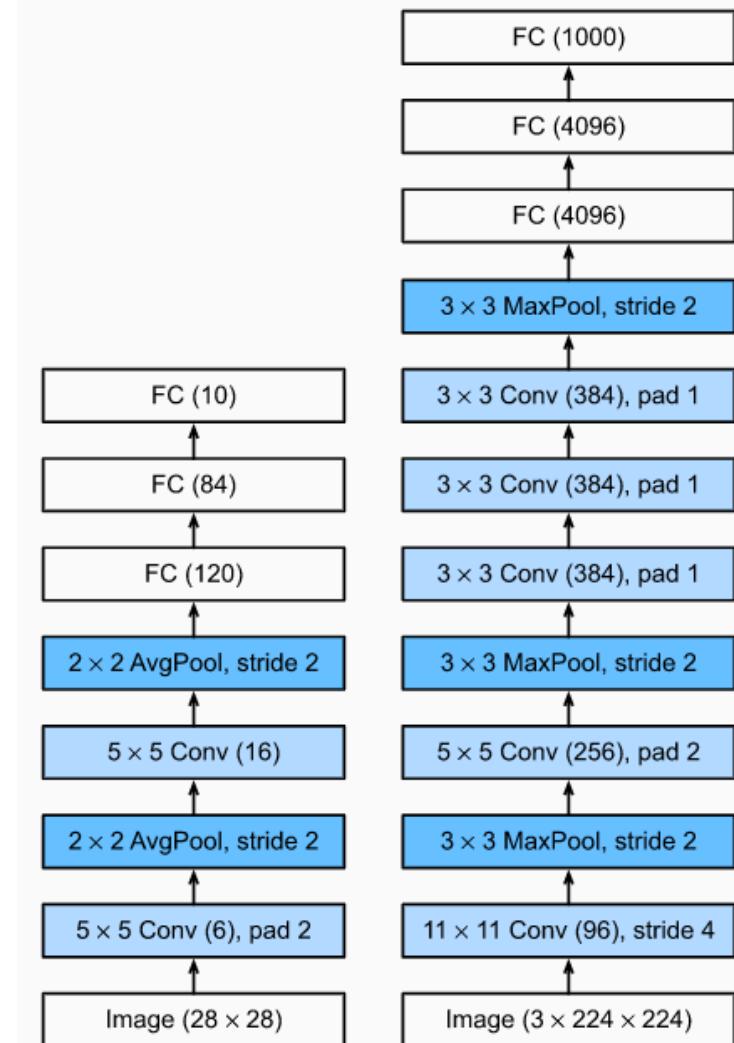
## Some popular CNN architectures

# LeNet and AlexNet

- The **design philosophies** of AlexNet and LeNet are **very similar**, but there are also **significant differences**.

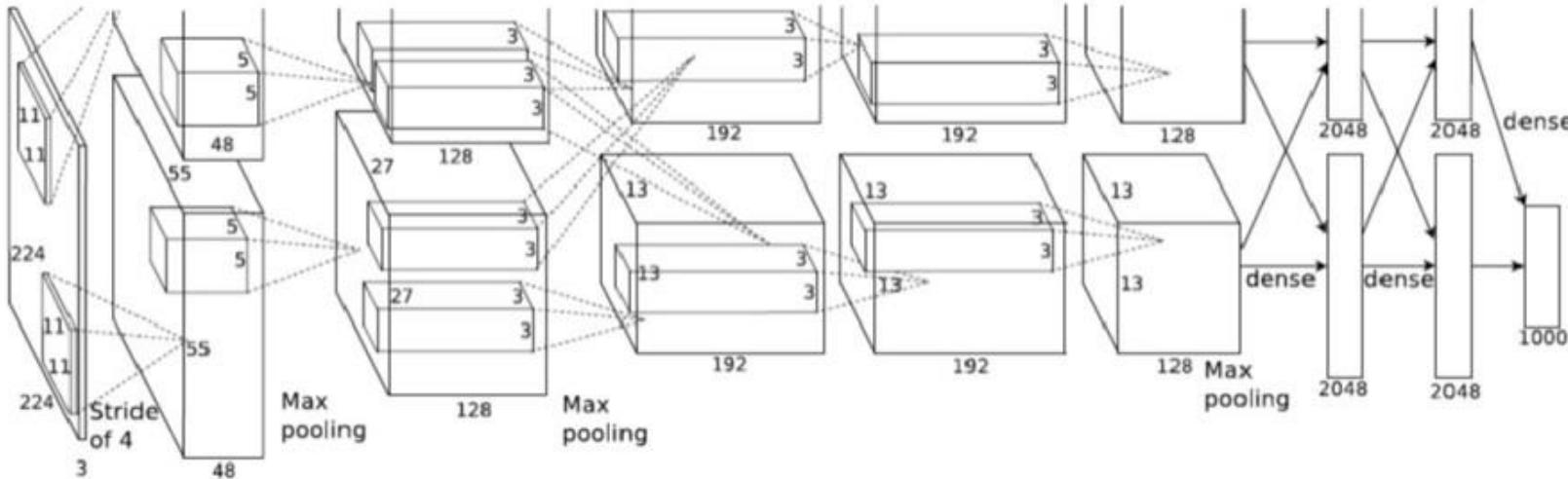
- AlexNet is much **deeper** than the comparatively small LeNet5.
- AlexNet **consists of eight layers**
  - Five convolutional layers, two fully-connected hidden layers, and one fully-connected output layer.
- AlexNet used the **ReLU** instead of the **sigmoid**
- In AlexNet's first layer, the convolution window shape is **11×11** because **objects in ImageNet** data tend to **occupy more pixels**.

(Source:Dive into DL)



# AlexNet

Alex Krizhevsky, et al, 2012



### Layer 1

Layer 1 is a Convolution Layer

Input Image size is – 224 x 224 x 3

Number of filters – 96

Filter size – 11 x 11 x 3

Stride – 4

Layer 1 Output

$$224/4 \times 224/4 \times 96 = 55 \times 55 \times 96 \\ (\text{because of stride } 4)$$

### Layer 2

Layer 2 is a Max Pooling Followed by Convolution

Input – 55 x 55 x 96

$$\text{Max pooling} - 55/2 \times 55/2 \times 96 = 27 \times 27 \times 96$$

Number of filters – 256

Filter size – 5 x 5 x 48

Layer 2 Output

$$27 \times 27 \times 256$$

### Layer 3,4,5

Layers 3, 4 & 5 follow on similar lines

### Layer 6

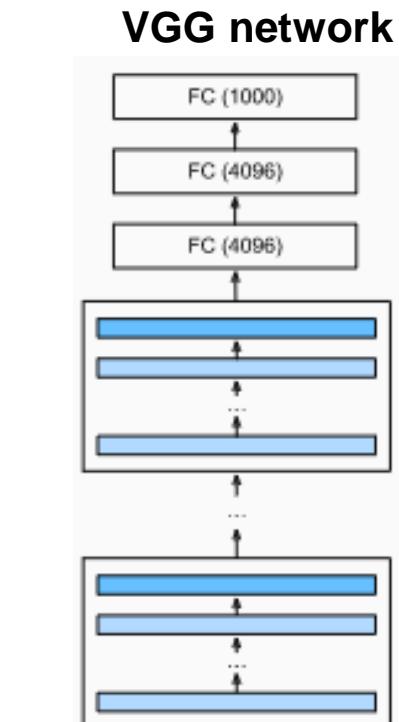
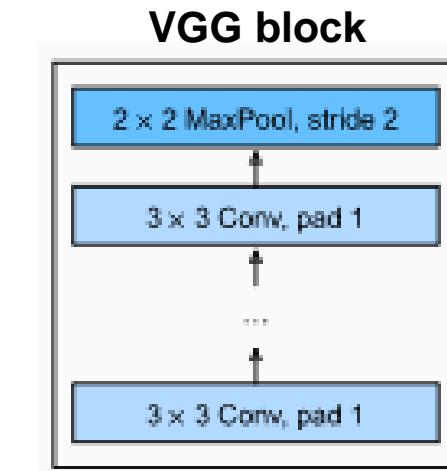
Layer 6 is fully connected

Input – 13 x 13 x 128 –> is transformed into a vector

And multiplied with a matrix of the following dim – (13 x 13 x 128) x 2048

# VGG Network

- Won ImageNet competition in 2014
- Moving from thinking in terms of **individual neurons** → **whole layers** → **blocks** (repeating patterns of layers)
- **VGG block**
  - One VGG block consists of a sequence of convolutional layers, followed by a max pooling layer for spatial downsampling.
- **VGG network**
  - Consists of many VGG blocks, followed by fully-connected layers
  - The original VGG network had 5 convolutional blocks
    - The first two have one convolutional layer each and the latter three contain two convolutional layers each.
    - The first block has 64 output channels and each subsequent block doubles the number of output channels, until that number reaches 512.



(Source: Dive into DL)

# MiniVGG for Cifar10

## Our Tutorial

<b>Layer Type</b>	<b>Output Size</b>	<b>Filter Size / Stride</b>
INPUT IMAGE	$32 \times 32 \times 3$	
CONV	$32 \times 32 \times 32$	$3 \times 3, K = 32$
ACT	$32 \times 32 \times 32$	
BN	$32 \times 32 \times 32$	
CONV	$32 \times 32 \times 32$	$3 \times 3, K = 32$
ACT	$32 \times 32 \times 32$	
BN	$32 \times 32 \times 32$	
POOL	$16 \times 16 \times 32$	$2 \times 2$
DROPOUT	$16 \times 16 \times 32$	
CONV	$16 \times 16 \times 64$	$3 \times 3, K = 64$
ACT	$16 \times 16 \times 64$	
BN	$16 \times 16 \times 64$	
CONV	$16 \times 16 \times 64$	$3 \times 3, K = 64$
ACT	$16 \times 16 \times 64$	
BN	$16 \times 16 \times 64$	
POOL	$8 \times 8 \times 64$	$2 \times 2$
DROPOUT	$8 \times 8 \times 64$	
FC	512	
ACT	512	
BN	512	
DROPOUT	512	
FC	10	
SOFTMAX	10	



50,000 images  
10 classes

Thanks for your attention!