FIT3181 Deep Learning

Week 04: Convolutional Neural Networks

Lecturer: Lim Chern Hong

Email: lim.chernhong@monash.edu

GROUP OF EIGHT AUSTRALIA

Department of Data Science and AI
Faculty of Information Technology, Monash University, Australia

Outline

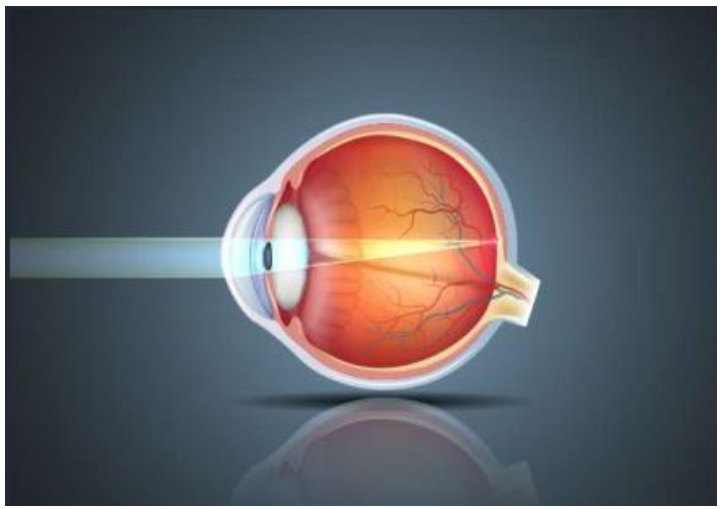
- Origin of human vision
- Convolutional Neural Networks
  - Convolutional operators in 1D, 2D, 3D
  - Standard CNN architecture:
    - Convolutional layer, Pooling layer, Fully-connected layer
  - CNN-based image recognition system
    - Batch norm and dropout layers
    - Popular datasets in vision

- 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.

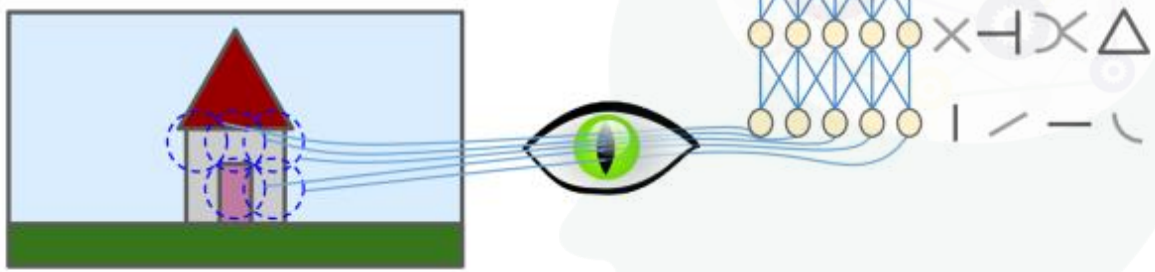



JUST BECAUSE YOU'RE MAD AT SOMEONE
DOESN'T MEAN YOU STOP LOVING THEM

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



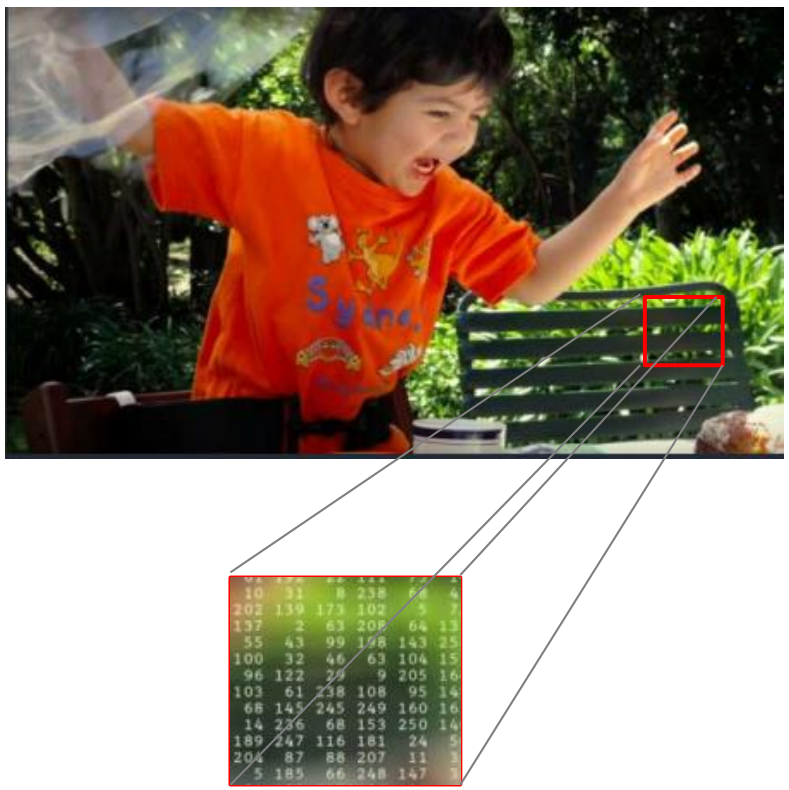
(Source: Hands On, Ch. 14)

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.*

(Source: www.pngwing.com)

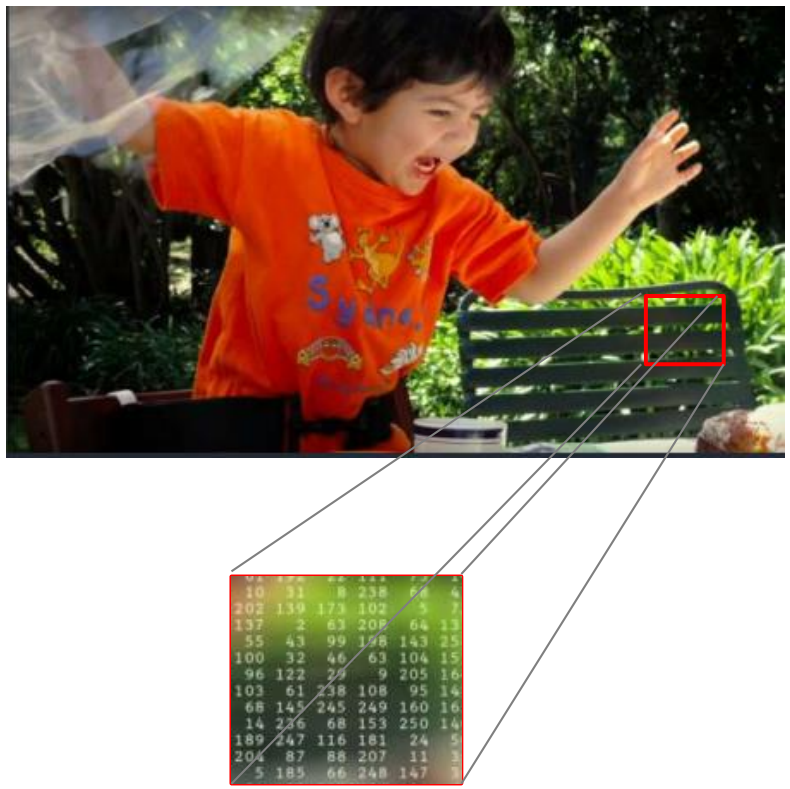
Recognition is much harder for machines

Human vision



Human vision vs computer vision

Human vision



Computer vision

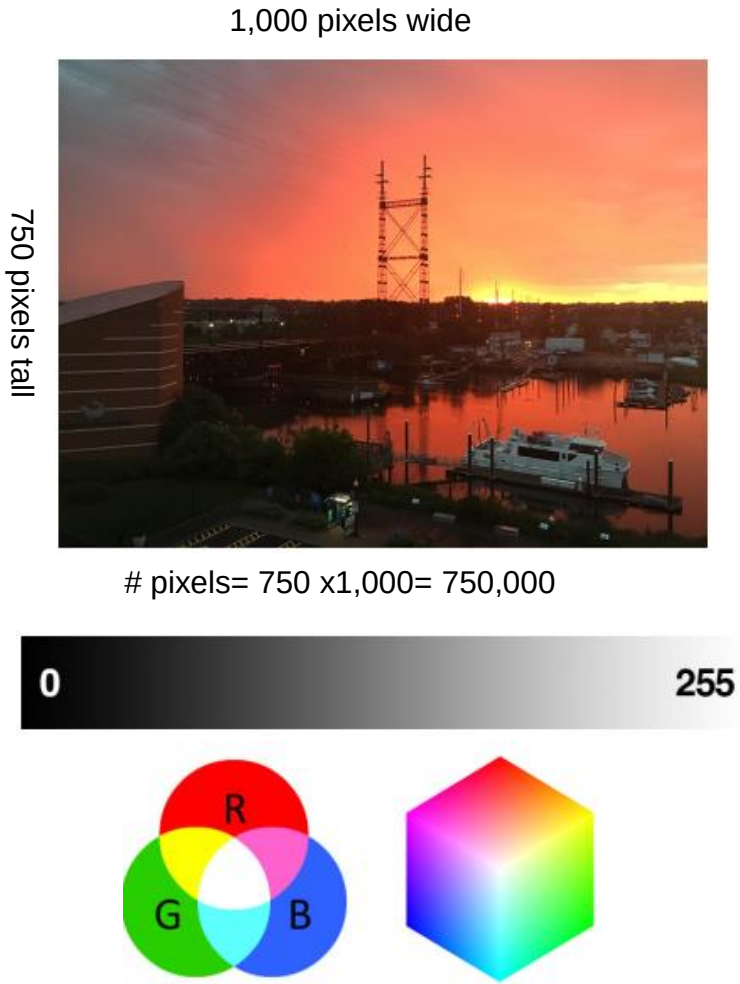
pixel

- Computer ‘see’ in a different way than we do. They are pixels!
- Every image can be represented as a 2-dimensional arrays of numbers, each represents a pixel.

Pixels: the building block of images

- Pixels are the raw building blocks of an image. Every image consists of a **set of pixels**.

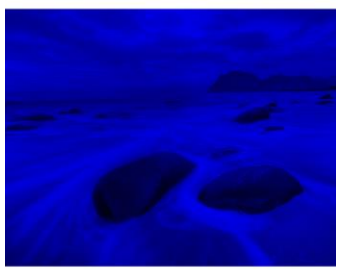
- Most pixels are represented in two ways:
  - Grayscale/single channel
    - A scalar between 0 (black) and 255 (white)
  - Color
    - Color pixels are represented in RGB system
      - Red, green, and blue channels



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 x 256 x 256=16,777,216

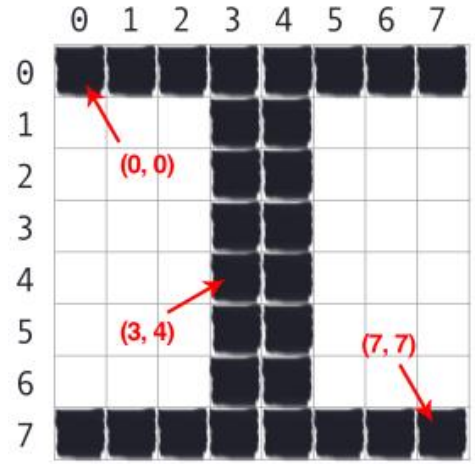
red



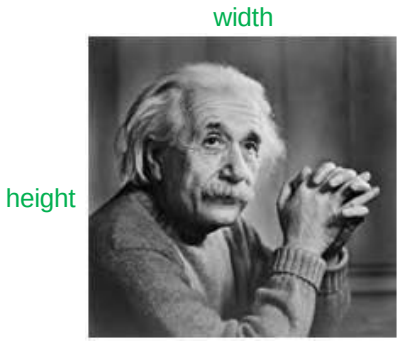
blue

green

Image coordinate system



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



A grayscale image can 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)
```

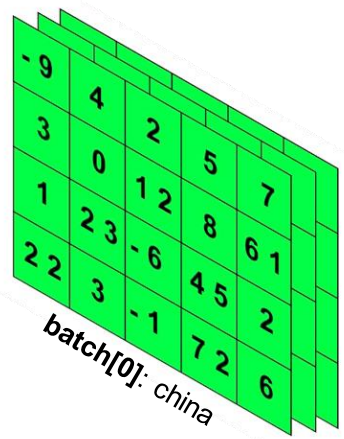
```
(2, 427, 640, 3)
```

2 images
height
width
RGB channels

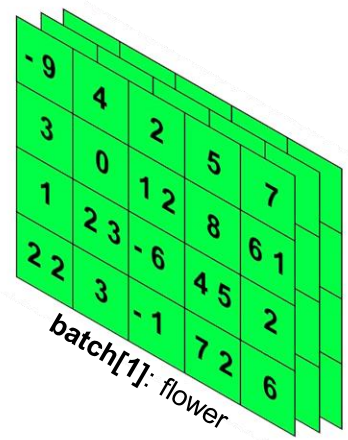
(batch, width, height, depth)

batch[0]: china

batch[1]: flower



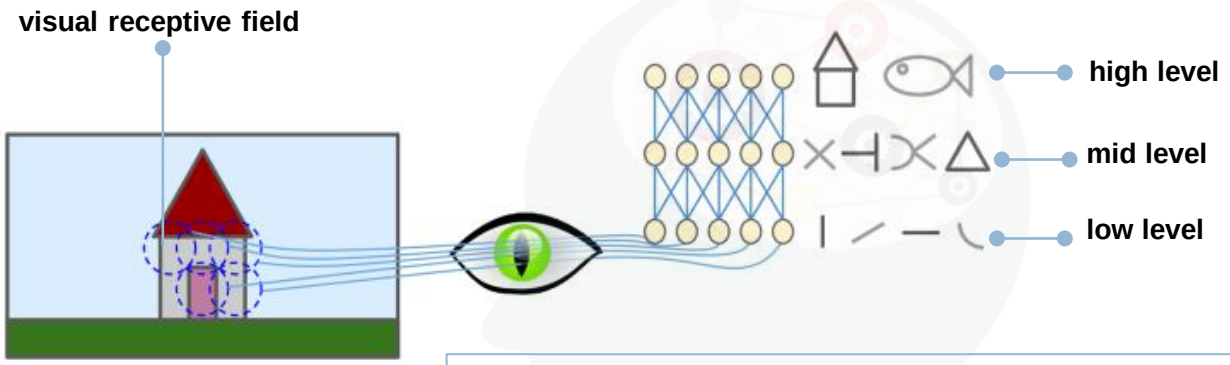
batch[0]: china



batch[1]: flower

Convolutional Neural Networks

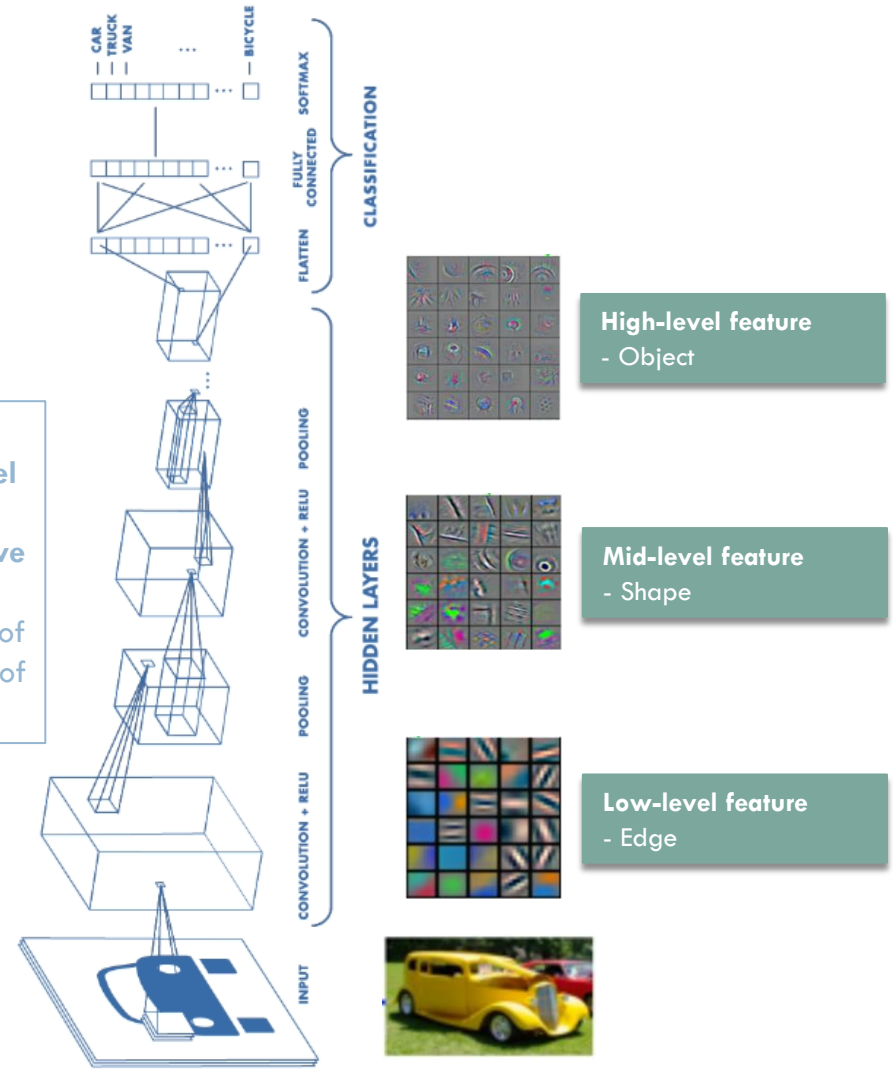
How to simulate human's vision?



(Source: Hands On, Ch. 14)

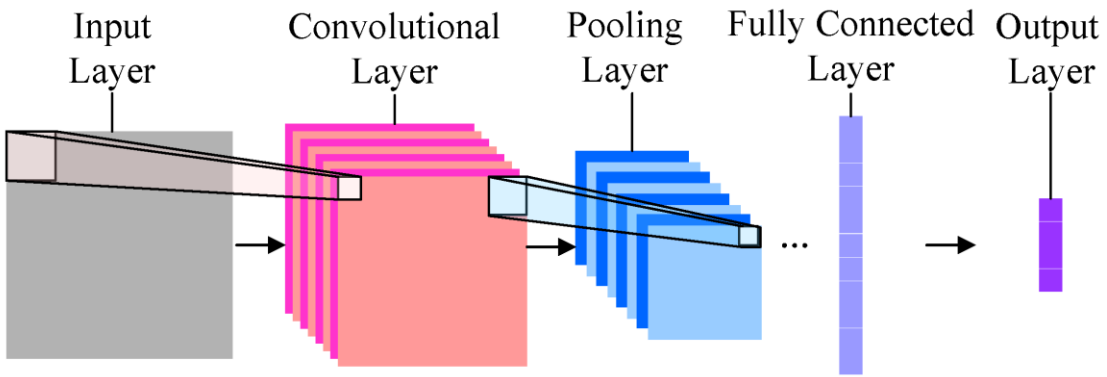
David H. Hubel and **Torsten Wiesel** performed a series of experiments on cats and monkeys in 1958 and 1959 (the **Nobel Prize** in **Physiology** or **Medicine** in **1981**).

- **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

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

$$W * x = sum(W \otimes x)$$

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

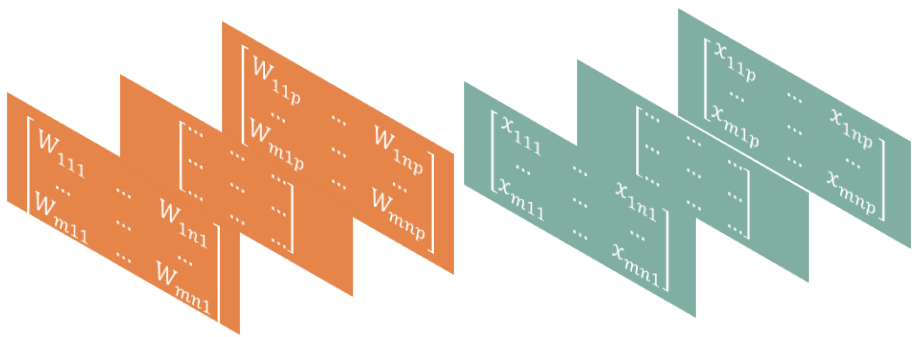
- **1D convolutional operation** $(W, x \in \mathbb{R}^{m \times 1})$
  - $W * x = \sum_{i=1}^{m} W_i x_i$

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

- **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

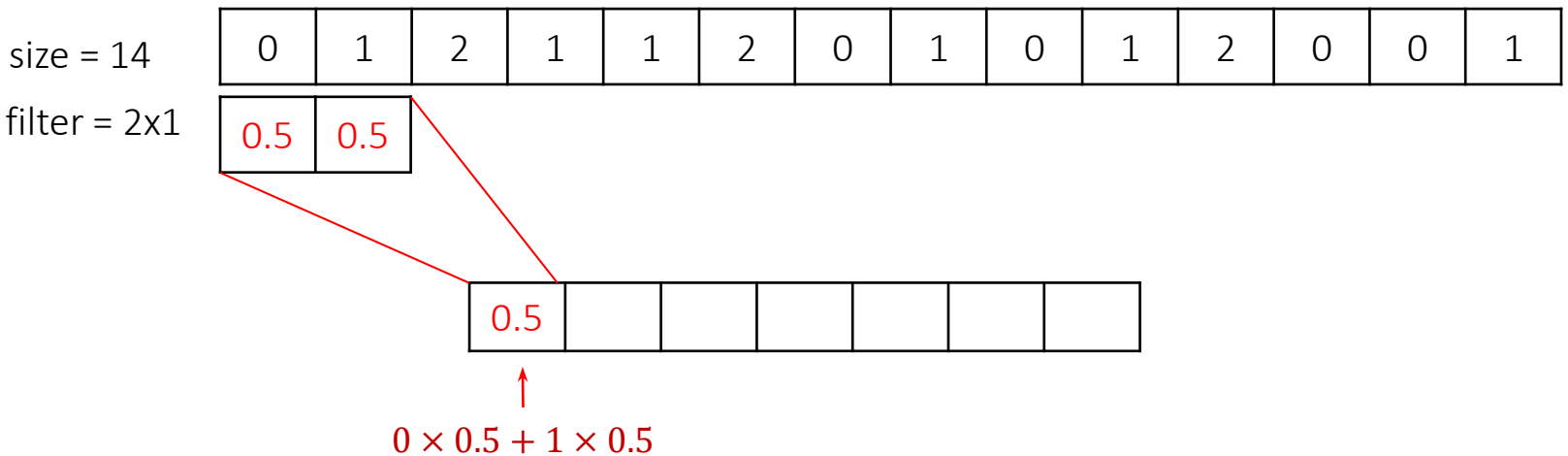
size = 14

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

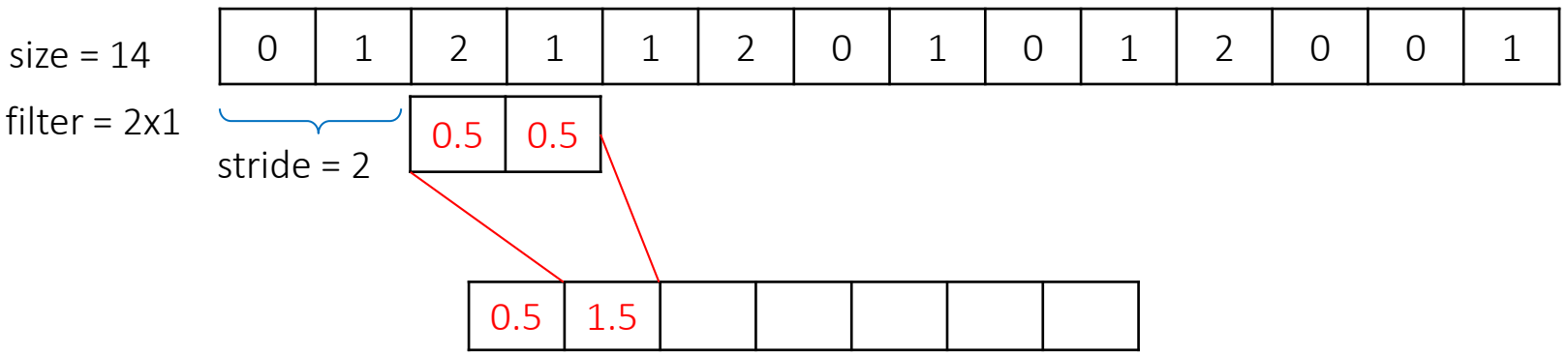
filter = 2x1

0.5	0.5

Convolution operation – 1D filter



Convolution operation – 1D filter

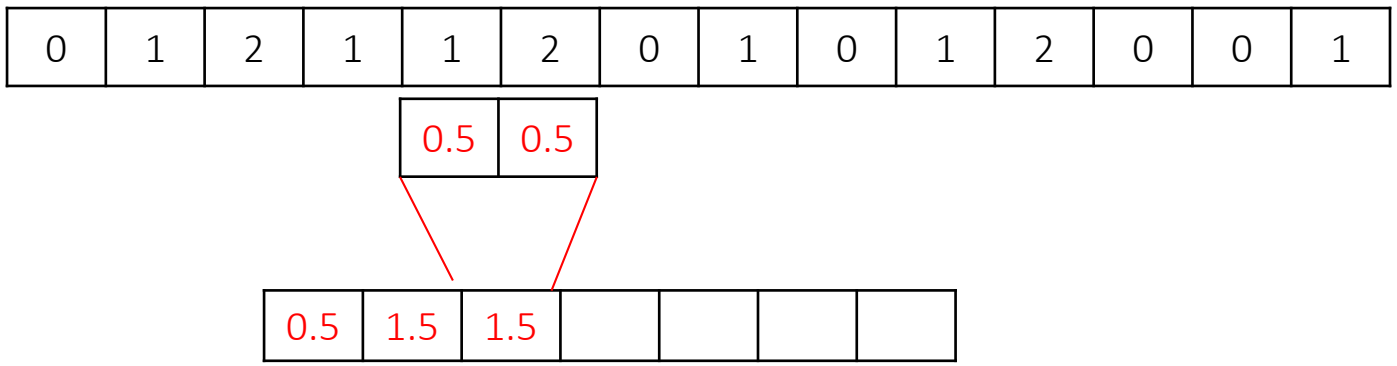


Convolution operation – 1D filter

size = 14

filter = 2x1

stride = 2

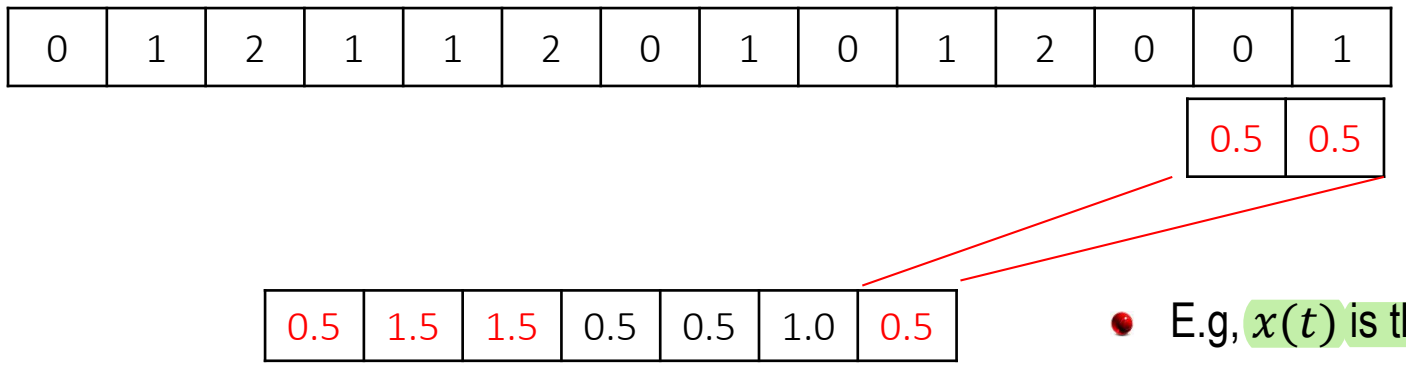


Convolution operation – 1D filter

size = 14

filter = 2x1

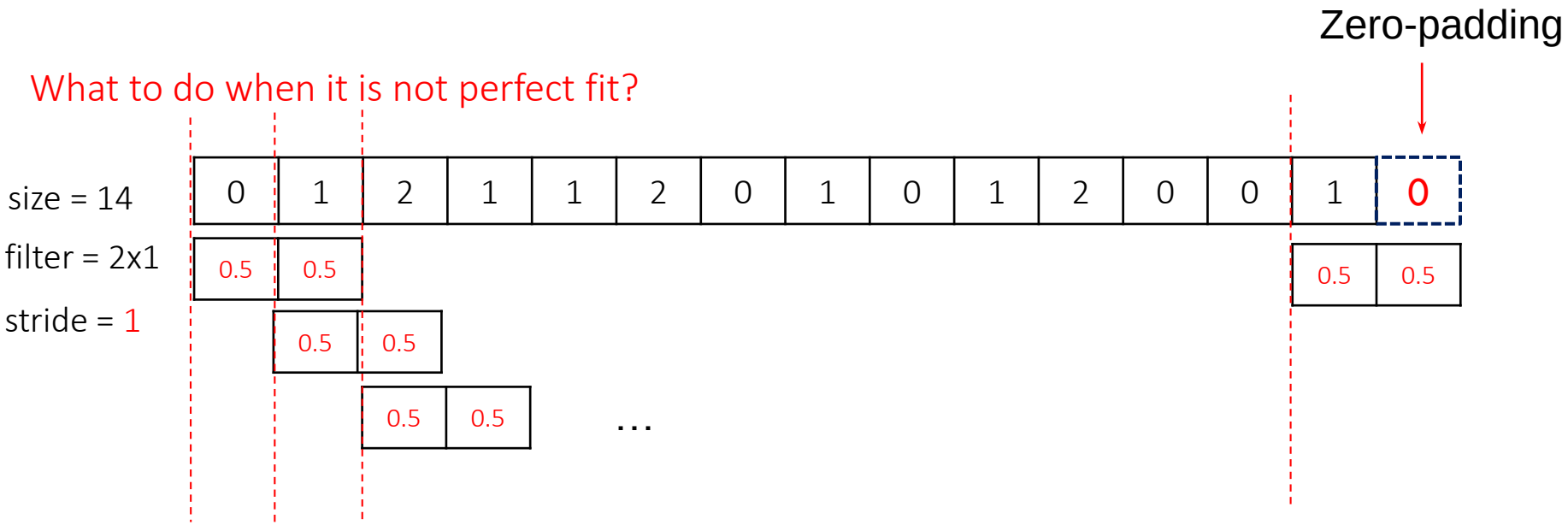
stride = 2



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



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?

yes we do

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

$H_i = 7$

W_i=7 **Zero padding** and $p = 1$
In TF or Keras, padding = 'same'

kernel size =(3,3)
***W* (filter or kernel)**

1	1	1
1	1	1
1	1	1

$f_h = 3$

$f_w = 3$

feature map

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

$H_o = 4$

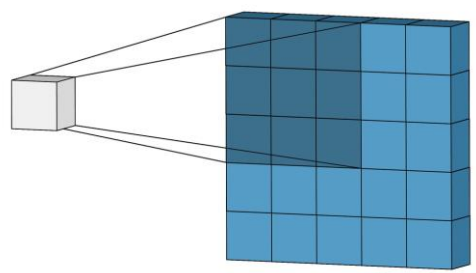
$W_o = 4$

- 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)

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

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



(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

$H_i = 7$

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

In TF or Keras, padding = 'valid'

kernel size =(3,3)

***W* (filter or kernel)**

1	1	1
1	1	1
1	1	1

$f_h = 3$

$f_w = 3$

feature map

1	23	21
3	26	18
-1	8	13

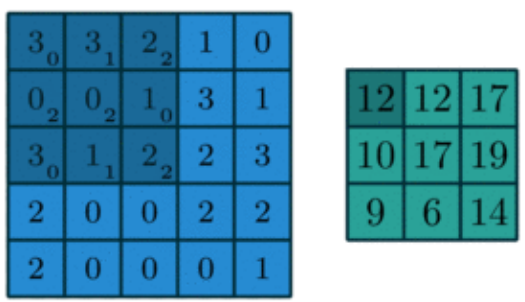
$H_o = 3$

$W_o = 3$

- 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)

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

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

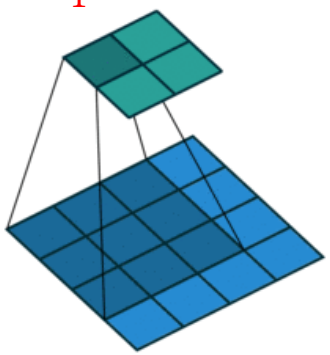


(Source: Internet)

Convolution layer

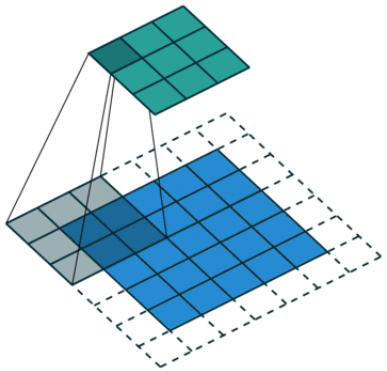
Example

$\frac{4-3}{1}+1=2$



s=1, p=0, f=3

$\frac{5-1}{2}+1=3$



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

depth=4

x (input tensor (7,8,4)) 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

$H_i = 7$

$W_i = 8$

depth=4 W_1(shape=(3,3,4))

1	1	1
1	1	1
1	1	1

depth=4 W_2(shape=(3,3,4))

-1	-1	-1
-1	-1	-1
-1	-1	-1

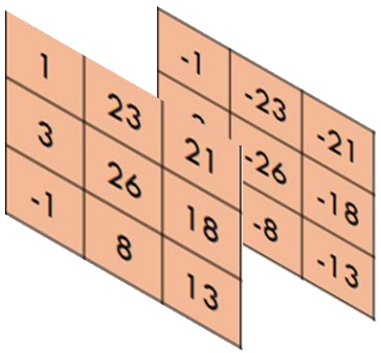
feature map$_1$

1	23	21
3	26	18
-1	8	13

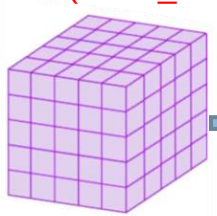
feature map$_2$

-1	-23	-21
-3	-26	-18
1	-8	-13

feature volume (3,3,2)
feature maps (3,3,2)

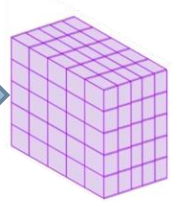


Inputs (batch_size,7,8,4)



CONV with filters (3,3,4,2)

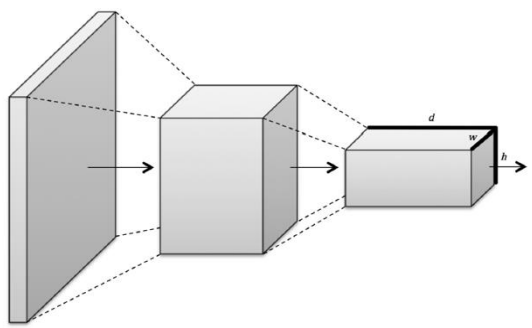
Feature volume (batch_size,3,3,2)



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

batch_size, in_height, in_width, in_channels

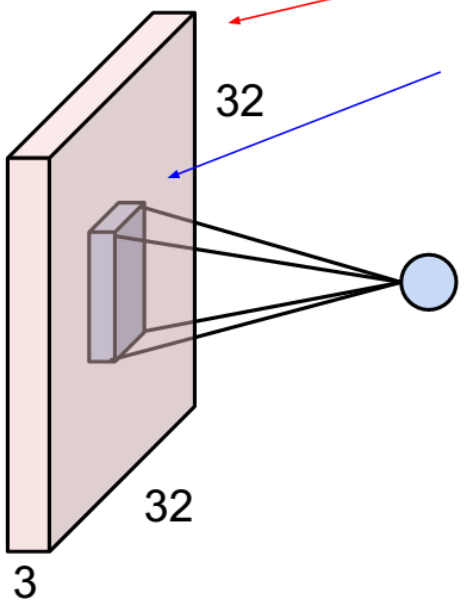
fil_height, fil_width, in_channels, out_channels



(Source: FDL Ch. 5)

Convolution layer: 3D convolution

Example



32x32x3 image

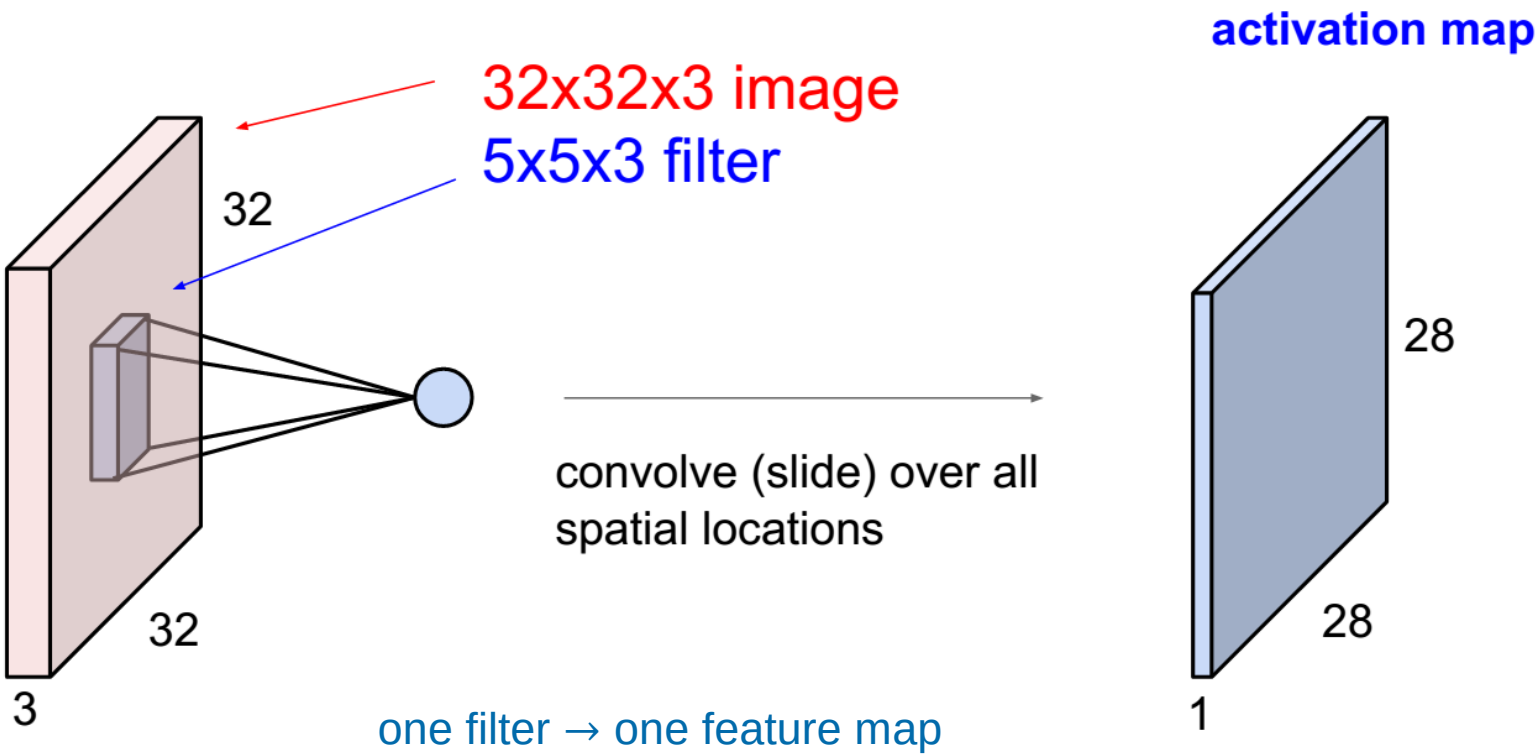
5x5x3 filter w

1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

"CS231n: Convolutional Neural Networks for Visual Recognition", Stanford, 2016

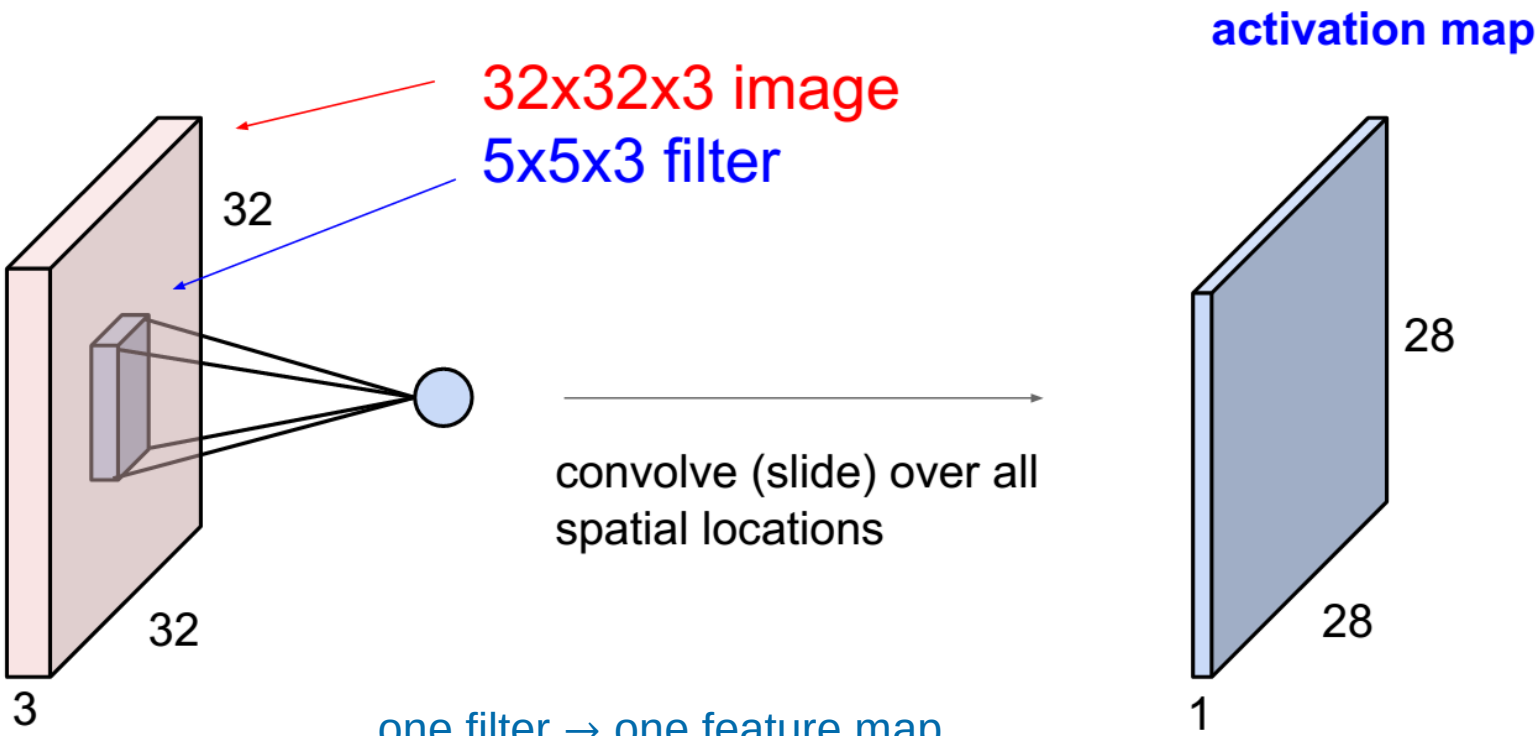
Convolution layer: one feature map



one filter → one feature map

"CS231n: Convolutional Neural Networks for Visual Recognition", Stanford, 2016

Convolution layer: one feature map

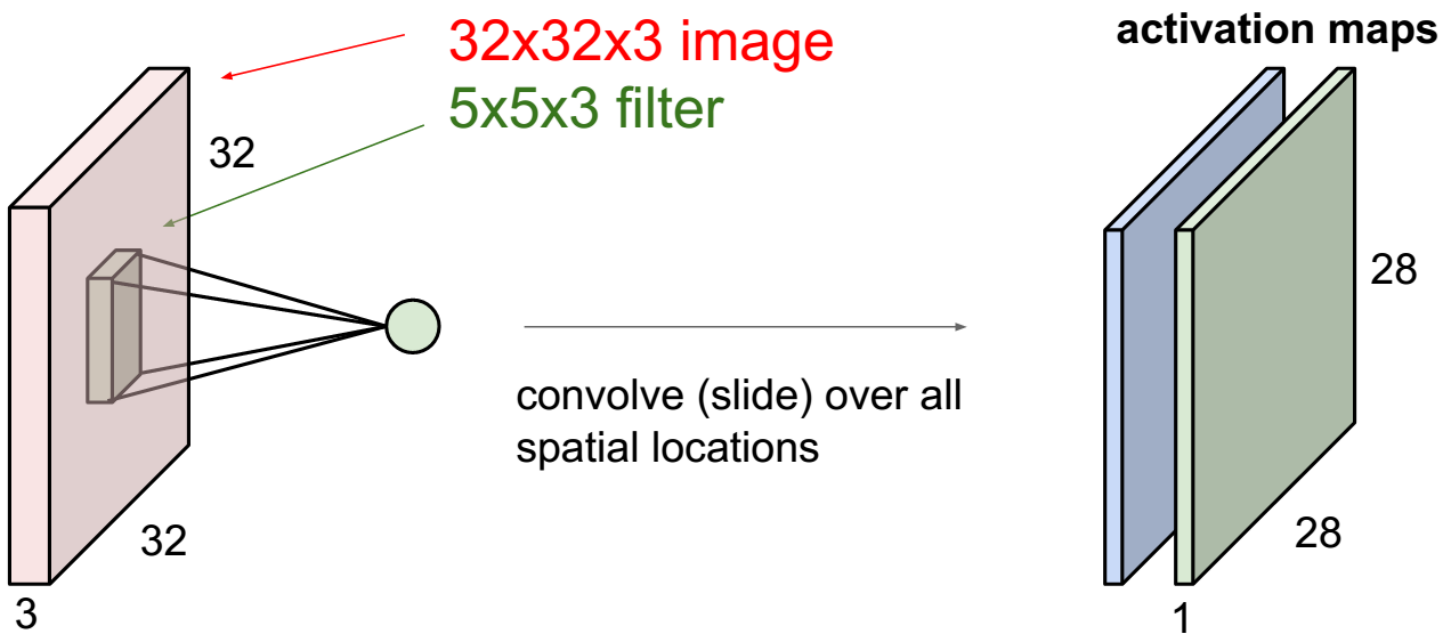


one filter → one feature map

$$28 = \frac{32-5}{1} + 1$$

"CS231n: Convolutional Neural Networks for Visual Recognition", Stanford, 2016

Convolution layer: two feature maps

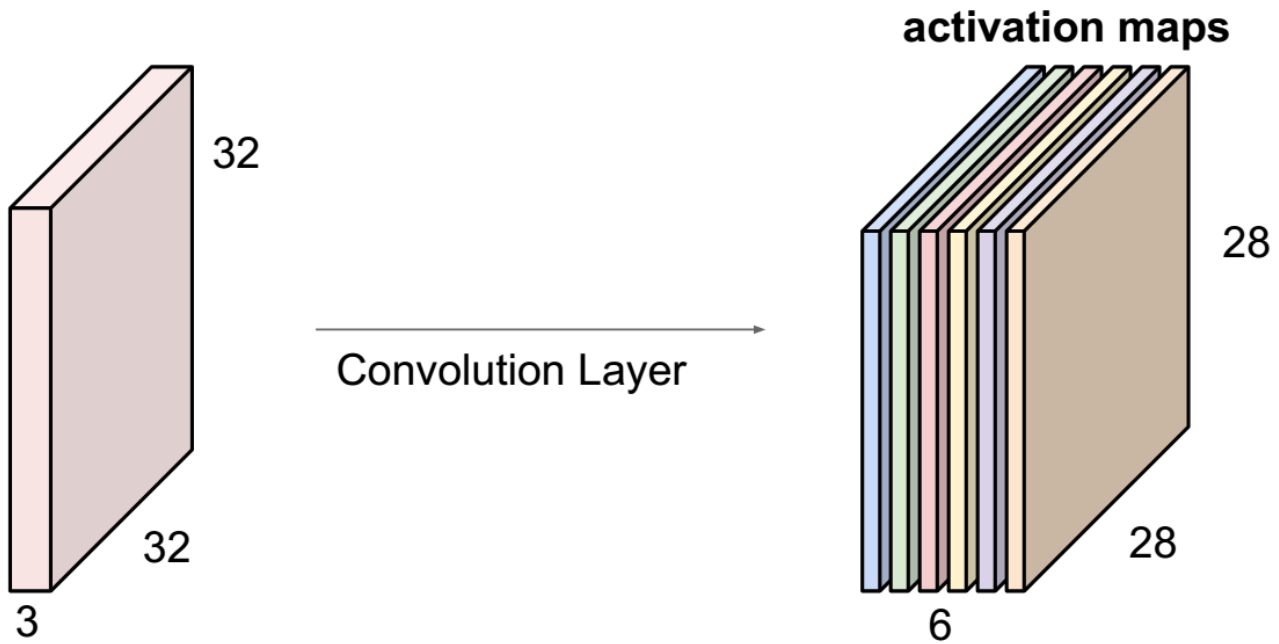


another filter → another feature map

"CS231n: Convolutional Neural Networks for Visual Recognition", Stanford, 2016

Convolution layer: multiple feature maps

Example

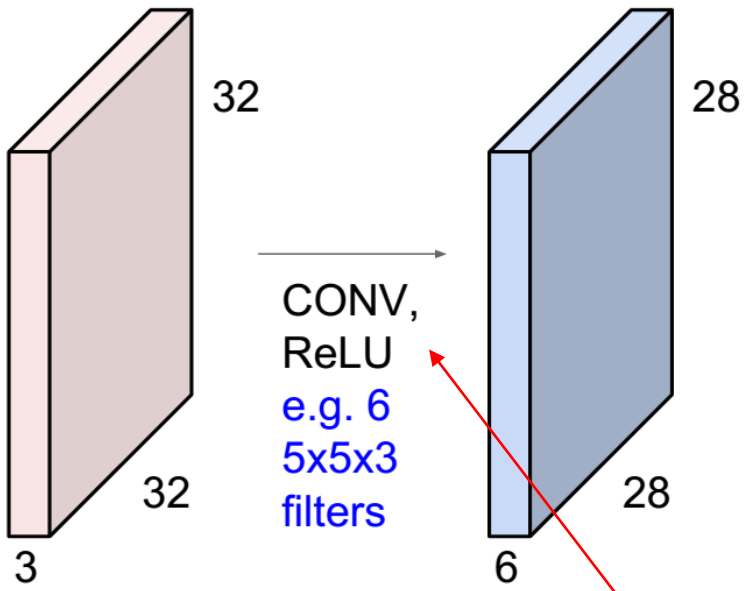


stacking up feature maps to form a new 3D volume = ‘new’ image

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

Convolution Layer with activation function

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

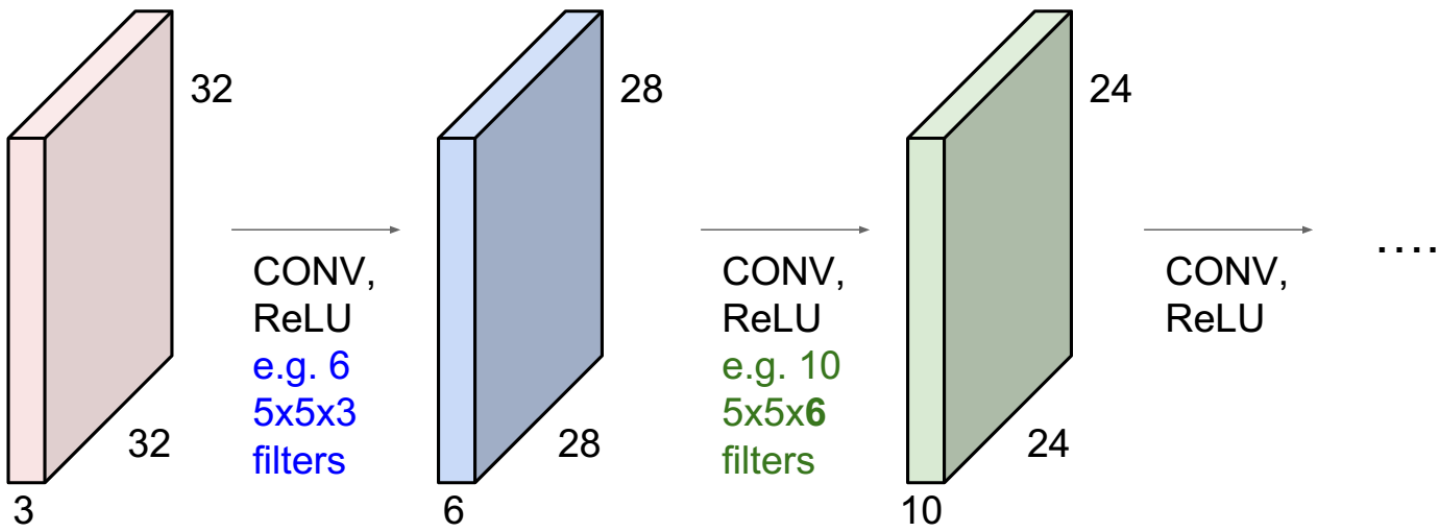


After convolution step, we further apply a ReLU activation function

"CS231n: Convolutional Neural Networks for Visual Recognition", Stanford, 2016

Convolution Layer with activation function

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



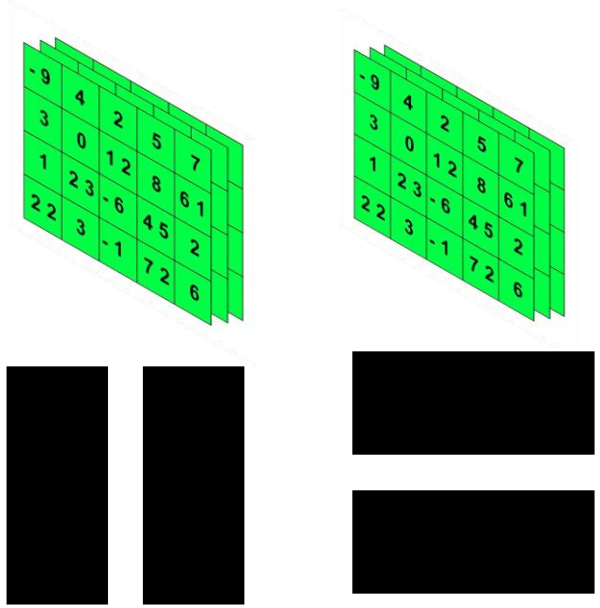
"CS231n: Convolutional Neural Networks for Visual Recognition", Stanford, 2016

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)
```

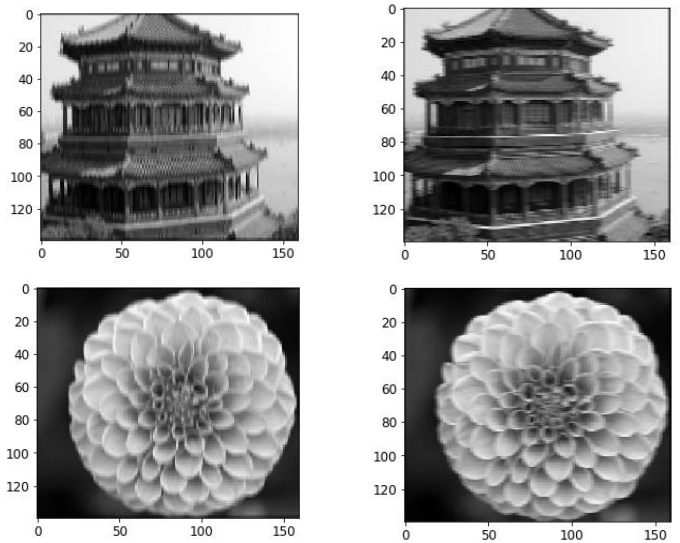
```
(2, 280, 320, 3)
```

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



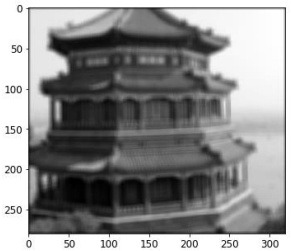
```
output = tf.nn.conv2d(batch, filters, strides=[1,2,2,1], padding="SAME")
print("Output shape:" + str(output.shape))
plt.imshow(output[0, :, :, 0], cmap='gray') # plot 1st image's 1nd feature map, channel 0
plt.show()
plt.imshow(output[0, :, :, 1], cmap='gray') # plot 1st image's 2nd feature map,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

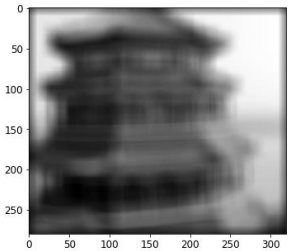


Effects of filters/kernels to images

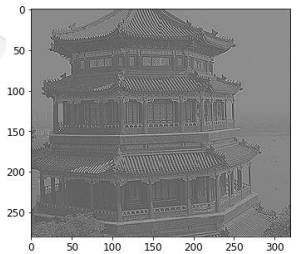
```
smallBlur = np.ones((7, 7), dtype="float") * (1.0 / (7 * 7))
```



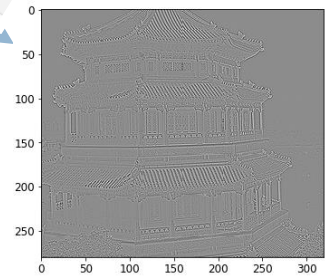
```
largeBlur = np.ones((21, 21), dtype="float") * (1.0 / (21 * 21))
```



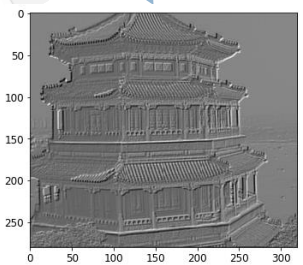
```
# construct a sharpening filter
sharpen = np.array(([0, -1, 0],
                    [-1, 5, -1],
                    [0, -1, 0]), dtype="int")
```



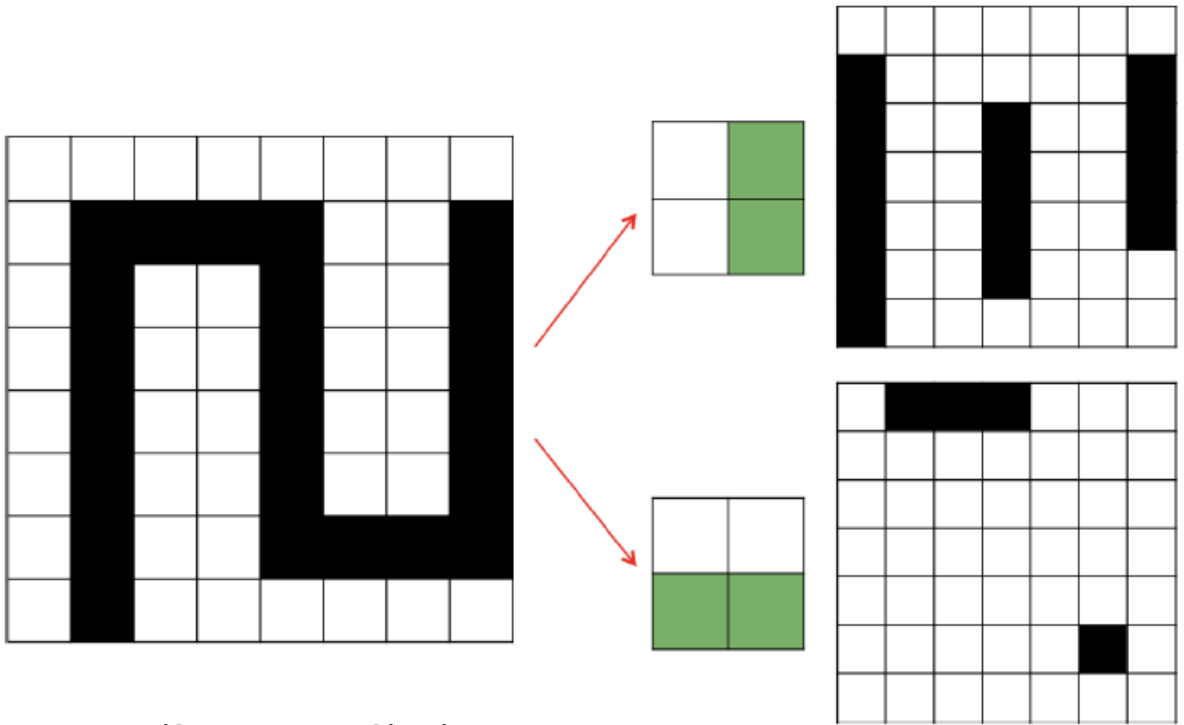
```
# construct the Laplacian kernel used to detect edge-like
# regions of an image
laplacian = np.array(([0, 1, 0],
                      [1, -4, 1],
                      [0, 1, 0]), dtype="int")
```



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



Convolution – 2D filter



Detect vertical lines

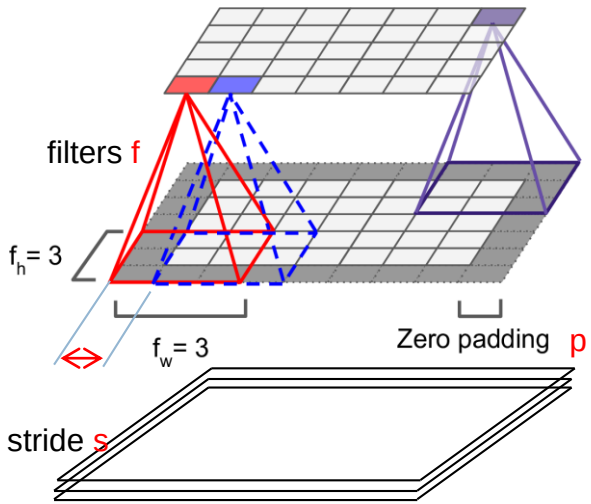
Detect horizontal lines

(Source: FDL Ch. 5)

Summary: full convolutional layer

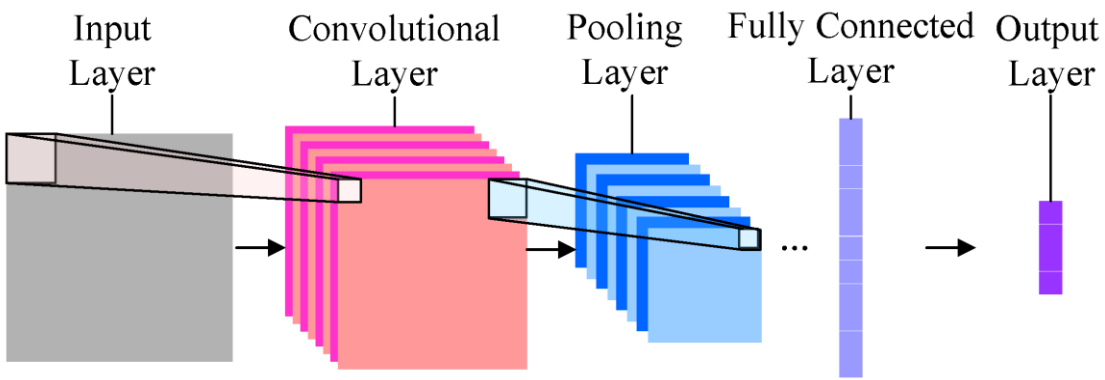
- Input is a volume, e.g., an RGB image:
  - Width W_i
  - Height H_i
  - Depth D
  - Zero padding
- Output is another volume
  - Width W_o
  - Height W_o
  - Depth K (= number of filters)
- W_i, H_i: The width and height of the input image
- W_o, H_o: The width and height of the output image

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



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

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

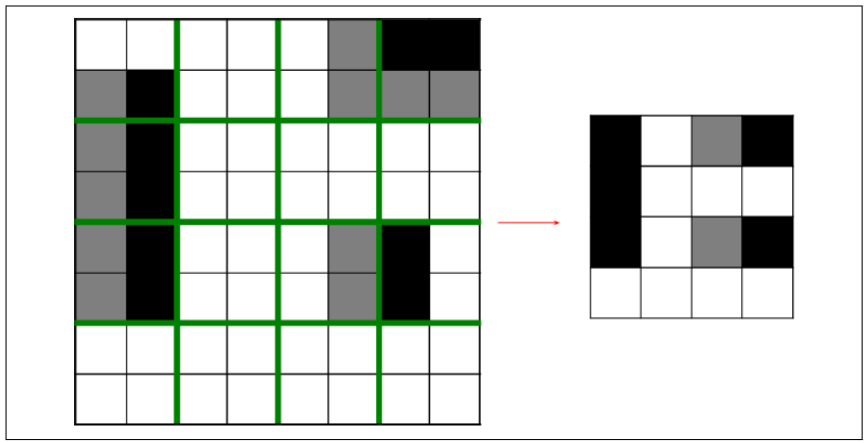
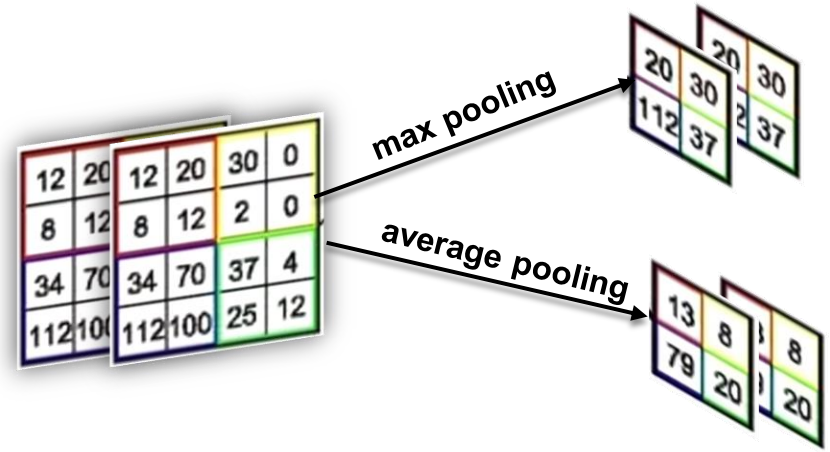


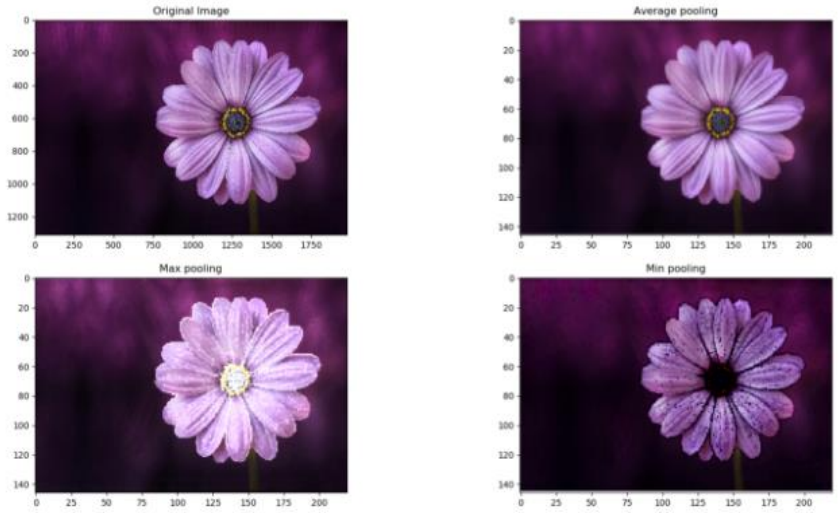
Figure 5-13. An illustration of how max pooling significantly reduces parameters as we move up the network

(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**

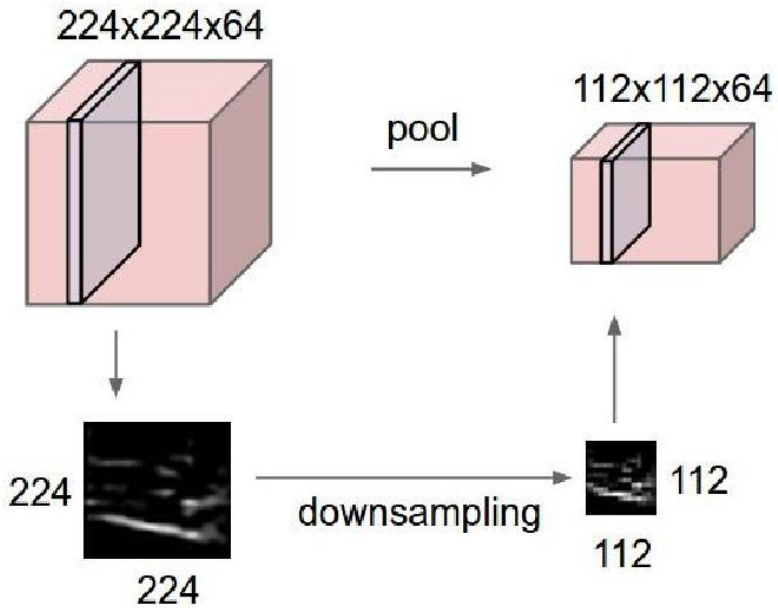




(Source: medium.com)

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)
```

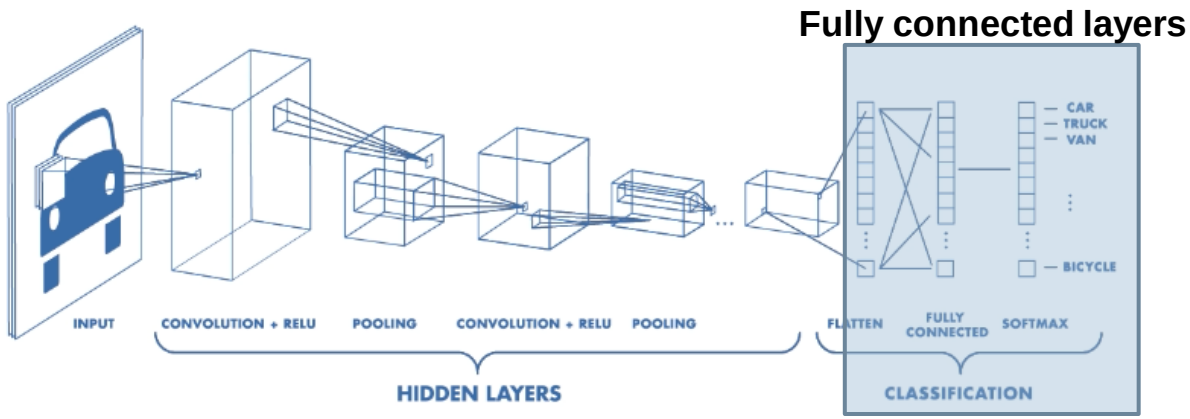
Average pooling

```
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] → 1 layer with 5x5x10= 250 neurons

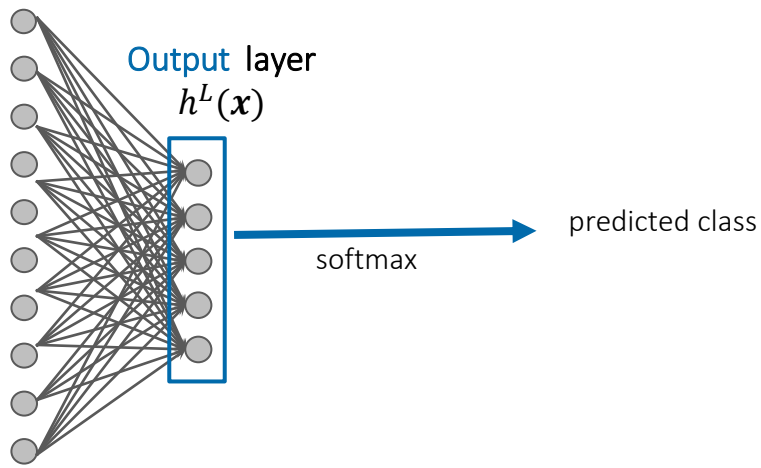
250 is the total number of features that ONE x sample has (we dont account for batch size here)

What is a softmax layer?

Classification setting

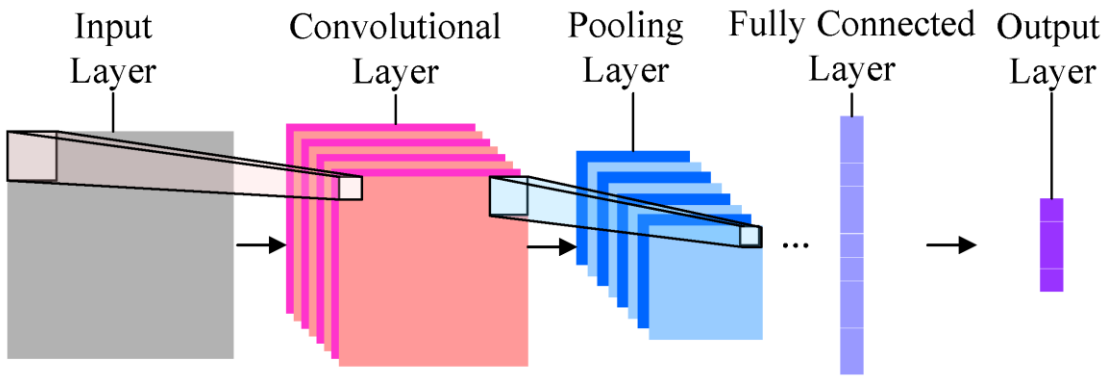
- Input: vector of dimension M
  - e.g., $h = [1, 2, 3]$, $\mathrm{M} = 3$
- Output: a discrete distribution of dimension M
  - e.g., $y = [0.09, 0.24, 0.67]$
- How to calculate prediction probability p?

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

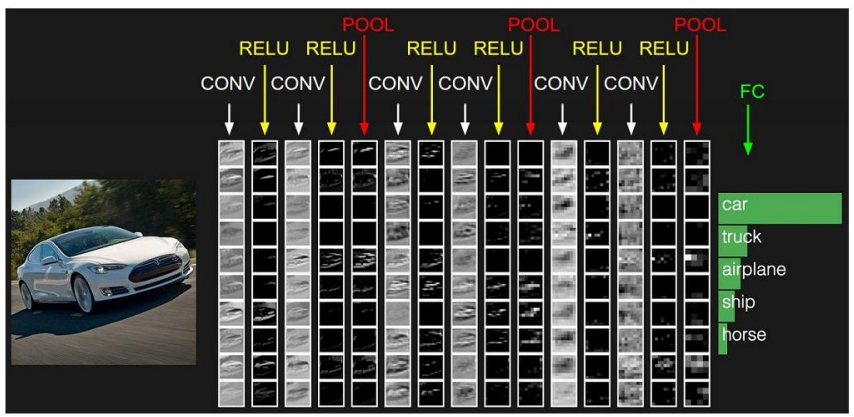


Put them all together

General architecture of CNNs



- Convolution stage
- Pooling stage
- Fully connected stage

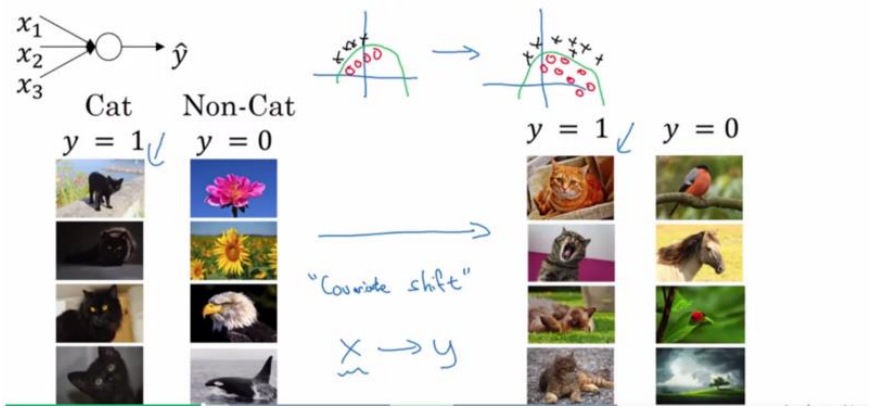


"CS231n: Convolutional Neural Networks for Visual Recognition", Stanford, 2016

Batch Normalization Layer

Covariate shift vs internal covariate shift

Covariate shift problem

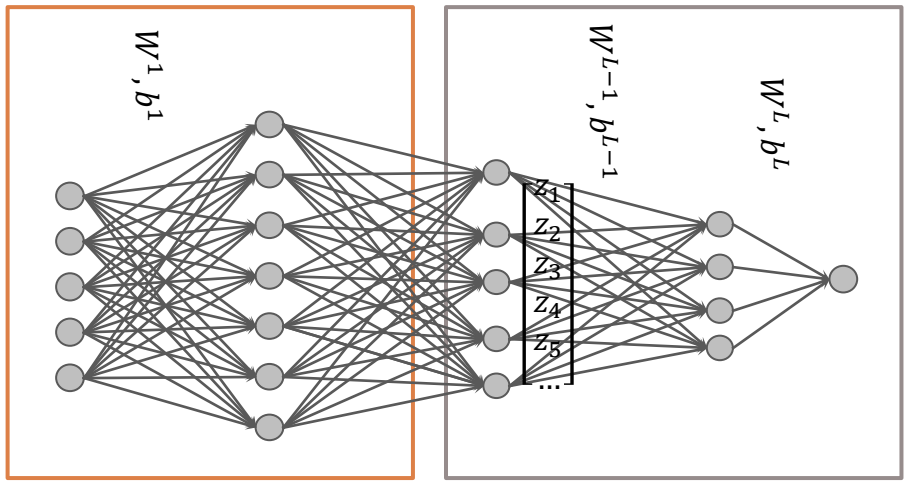


(Source: DL course Andrew Ng)

Covariate shift problem

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

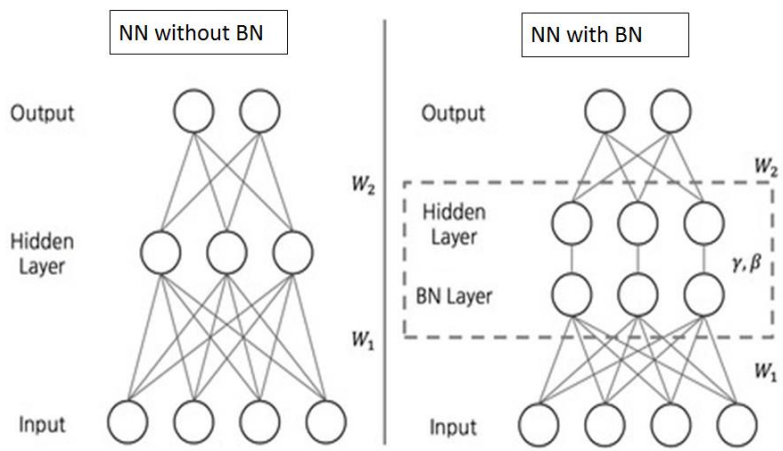
Internal covariate shift



- 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



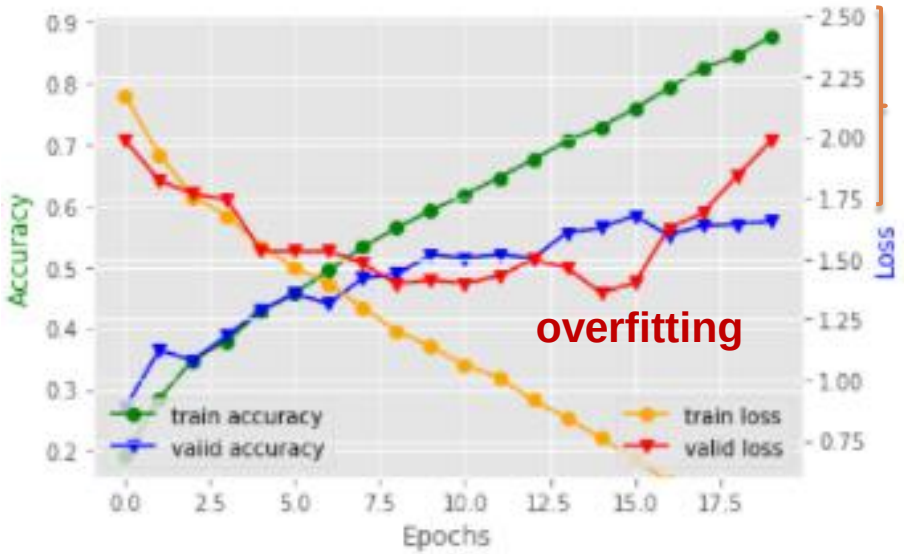
(Source: medium.com)

- Let $z = W^k h^k + b^k$ be the mini-batch before activation. We compute the normalized $\hat{z}$ as
  - $\hat{\boldsymbol{z}} = \frac{\boldsymbol{z}-\boldsymbol{\mu_B}}{\sqrt{\boldsymbol{\sigma_B^2}+\epsilon}}$ where ϵ is a small value such as $1e^{-7}$
  - $\boldsymbol{\mu_B} = \frac{1}{b}\sum_{i=1}^{b} \boldsymbol{z_i}$ is the empirical mean
  - $\boldsymbol{\sigma_B^2} = \frac{1}{b}\sum_{i=1}^{b}(\boldsymbol{z_i} - \boldsymbol{\mu_B})^2$ is the empirical variance
- We scale the normalized $\hat{z}$
  - $\boldsymbol{z_{BN}} = \boldsymbol{\gamma}\hat{\boldsymbol{z}} + \boldsymbol{\beta}$ 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
  - $\boldsymbol{h^{k+1}} = \boldsymbol{\sigma(z_{BN})}$ **batch normalize before activation**

✓ At testing time, we replace the mini-batch μ_B and σ_B with running averages of $\tilde{\mu}_B$ and $\tilde{\sigma}_B$ computed during the training process.

- ✓ $\tilde{\boldsymbol{\mu}}_{\boldsymbol{B}} = (\boldsymbol{1}-\boldsymbol{\theta})\tilde{\boldsymbol{\mu}}_{\boldsymbol{B}} + \boldsymbol{\theta}\boldsymbol{\mu}_{\boldsymbol{B}}$
- ✓ $\tilde{\boldsymbol{\sigma}}_{\boldsymbol{B}} = (\boldsymbol{1}-\boldsymbol{\theta})\tilde{\boldsymbol{\sigma}}_{\boldsymbol{B}} + \boldsymbol{\theta}\boldsymbol{\sigma}_{\boldsymbol{B}}$
- ✓ $0 < \theta < 1$ is the momentum decay

Training with versus without batch norm

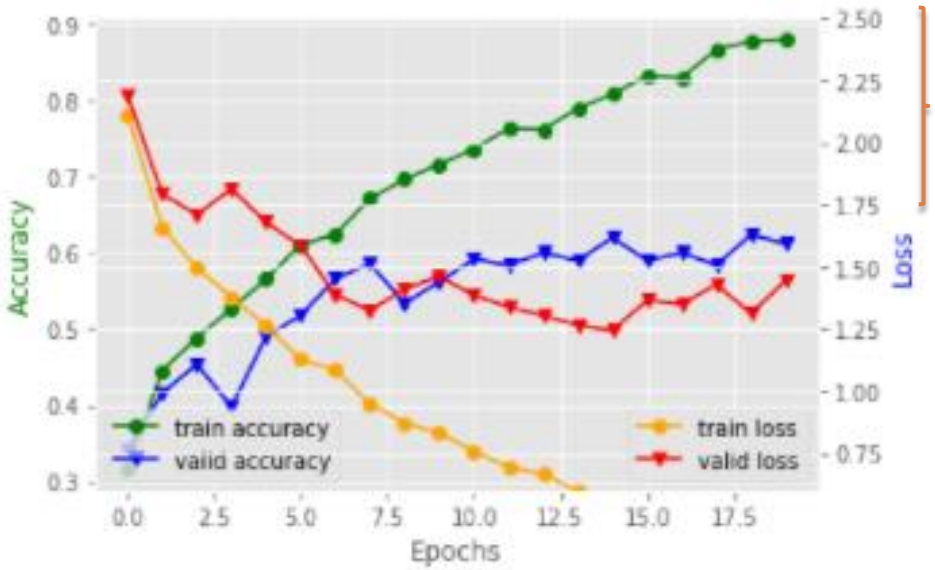


87%

Bigger gap

57%

Without batch norm



88%

Smaller gap

62%

With batch norm

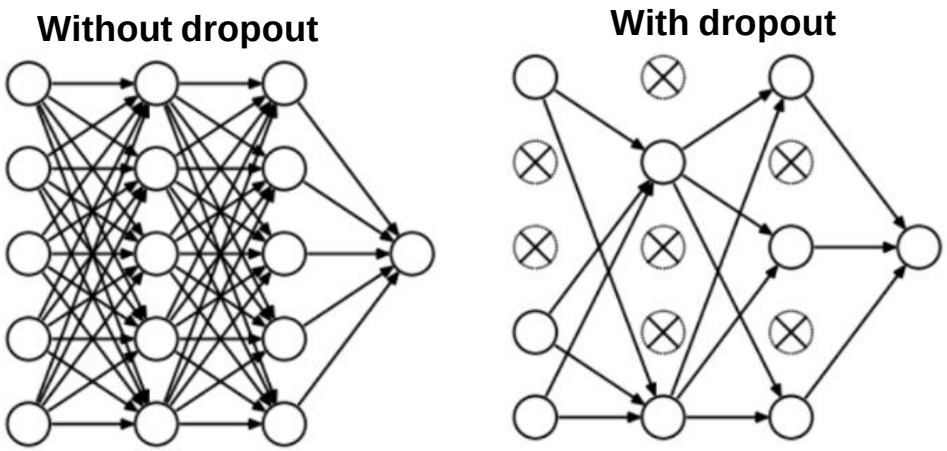
```
Epoch 1/20
63/63 [==============================] - 12s 191ms/step - loss: 2.1712 - accuracy: 0.1912 - val_loss: 1.9883 - val_accuracy: 0.2700
Epoch 2/20
63/63 [==============================] - 12s 186ms/step - loss: 1.9299 - accuracy: 0.2853 - val_loss: 1.8219 - val_accuracy: 0.3640
Epoch 3/20
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
```

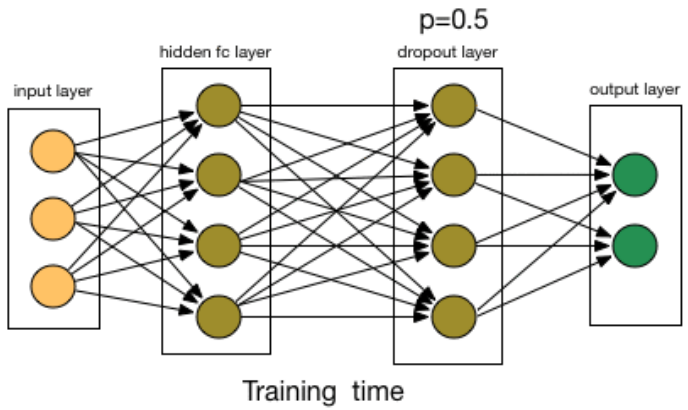
Dropout Layer

Dropout

Reduce Overfitting



(Source: Analytics Vidhya)



(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

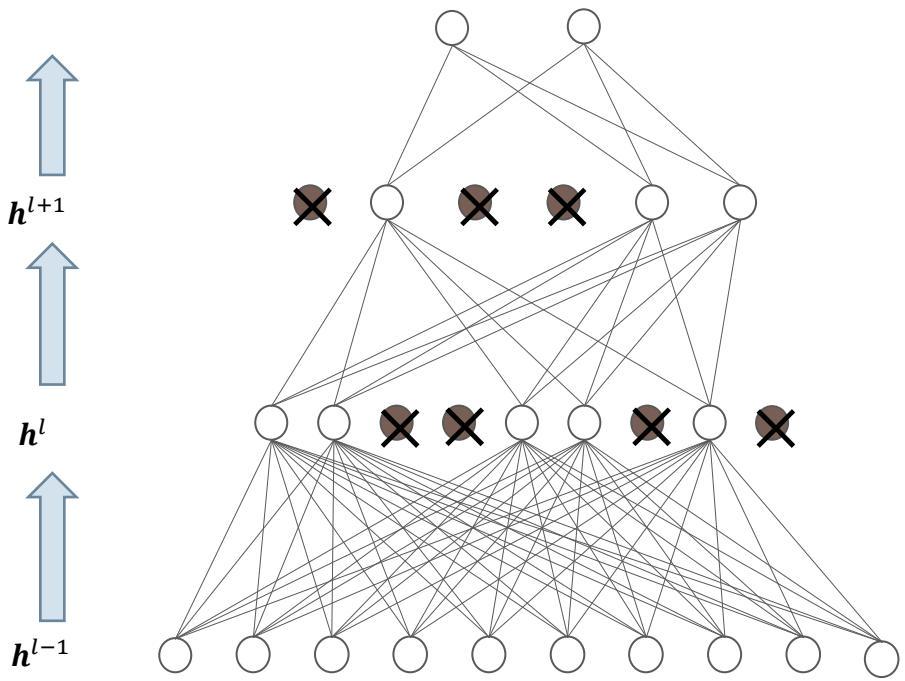
Reduce Overfitting

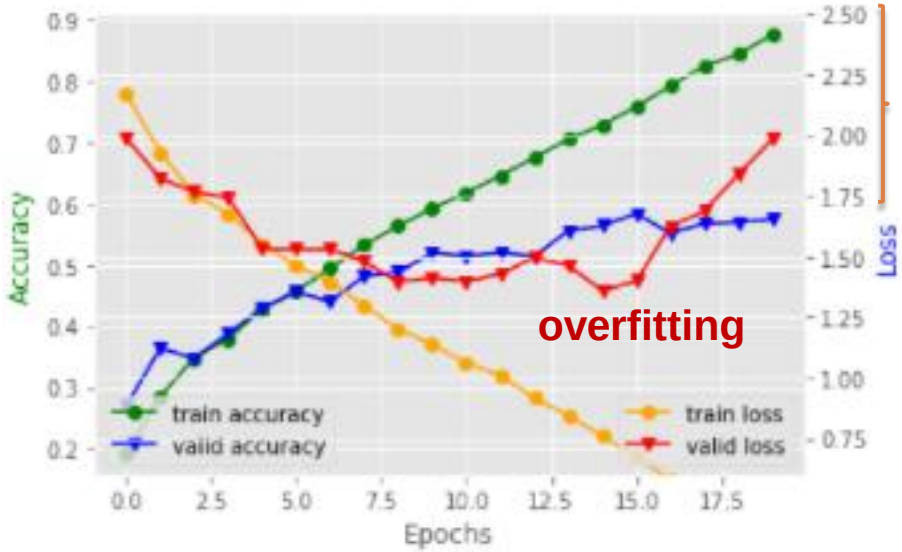
- Computationally efficiently
- Can be considered a **bagged ensemble** of an exponential number (2^N) of neural networks.
- **Training**

$$\boldsymbol{r} \sim \text{Bernoulli}(\mu)$$
$$\widetilde{\boldsymbol{h}}^l = \boldsymbol{h}^l \odot \boldsymbol{r}$$
$$\boldsymbol{h}^{l+1} = \sigma\left(W^{(l)^\top} \widetilde{\boldsymbol{h}}^l + \boldsymbol{b}^l\right)$$

- **Testing**

$$\widetilde{\boldsymbol{h}}^l = \boldsymbol{h}^l \times (1 - \mu)$$
$$\boldsymbol{h}^{l+1} = \sigma\left(W^{l^\top} \widetilde{\boldsymbol{h}}^l + \boldsymbol{b}^l\right)$$





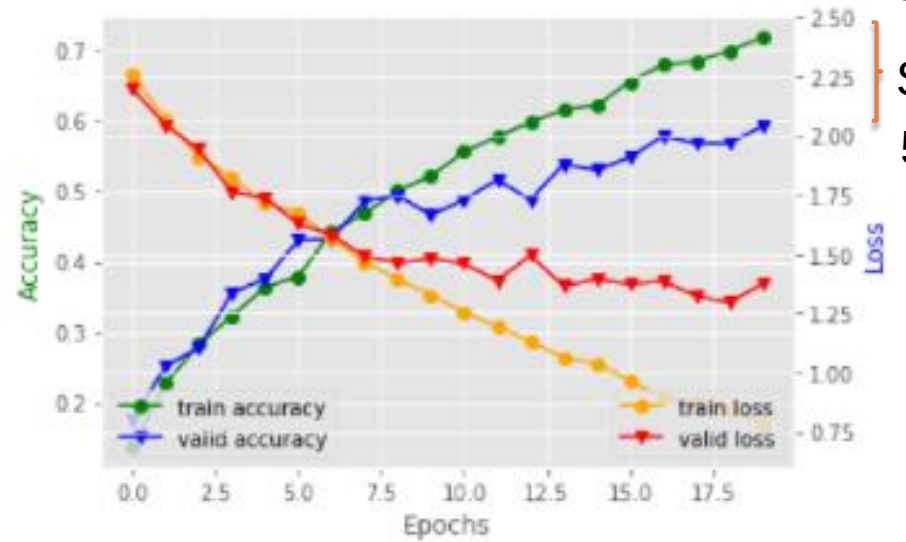
87%

Bigger gap

57%

Without Dropout

```
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
```



71%

Smaller gap

59%

With Dropout

```
Epoch 1/20
63/63 [==============================] - 15s 241ms/step - loss: 2.2590 - accuracy: 0.1375 - val_loss: 2.1962 - val_accuracy: 0.1760
Epoch 2/20
63/63 [==============================] - 15s 237ms/step - loss: 2.0654 - accuracy: 0.2278 - val_loss: 2.0436 - val_accuracy: 0.2520
Epoch 3/20
63/63 [==============================] - 15s 237ms/step - loss: 1.9132 - accuracy: 0.2828 - val_loss: 1.9403 - val_accuracy: 0.2780
Epoch 4/20
63/63 [==============================] - 15s 240ms/step - loss: 1.8224 - accuracy: 0.3232 - val_loss: 1.7622 - val_accuracy: 0.3560
Epoch 5/20
63/63 [==============================] - 15s 242ms/step - loss: 1.7136 - accuracy: 0.3640 - val_loss: 1.7369 - val_accuracy: 0.3760
Epoch 6/20
63/63 [==============================] - 15s 238ms/step - loss: 1.6742 - accuracy: 0.3790 - val_loss: 1.6274 - val_accuracy: 0.4300
Epoch 7/20
63/63 [==============================] - 15s 239ms/step - loss: 1.5645 - accuracy: 0.4430 - val_loss: 1.5805 - val_accuracy: 0.4320
Epoch 8/20
63/63 [==============================] - 15s 239ms/step - loss: 1.4669 - accuracy: 0.4685 - val_loss: 1.4904 - val_accuracy: 0.4860
Epoch 9/20
63/63 [==============================] - 15s 238ms/step - loss: 1.3943 - accuracy: 0.5008 - val_loss: 1.4624 - val_accuracy: 0.4940
Epoch 10/20
63/63 [==============================] - 15s 240ms/step - loss: 1.3266 - accuracy: 0.5215 - val_loss: 1.4785 - val_accuracy: 0.4660
Epoch 11/20
63/63 [==============================] - 15s 239ms/step - loss: 1.2562 - accuracy: 0.5558 - val_loss: 1.4598 - val_accuracy: 0.4880
Epoch 12/20
63/63 [==============================] - 15s 239ms/step - loss: 1.1920 - accuracy: 0.5775 - val_loss: 1.3882 - val_accuracy: 0.5160
Epoch 13/20
63/63 [==============================] - 15s 239ms/step - loss: 1.1324 - accuracy: 0.5975 - val_loss: 1.4941 - val_accuracy: 0.4880
Epoch 14/20
63/63 [==============================] - 15s 239ms/step - loss: 1.0612 - accuracy: 0.6158 - val_loss: 1.3655 - val_accuracy: 0.5380
Epoch 15/20
63/63 [==============================] - 15s 239ms/step - loss: 1.0385 - accuracy: 0.6215 - val_loss: 1.3924 - val_accuracy: 0.5300
Epoch 16/20
63/63 [==============================] - 15s 244ms/step - loss: 0.9641 - accuracy: 0.6555 - val_loss: 1.3736 - val_accuracy: 0.5480
Epoch 17/20
63/63 [==============================] - 15s 245ms/step - loss: 0.8937 - accuracy: 0.6793 - val_loss: 1.3853 - val_accuracy: 0.5780
Epoch 18/20
63/63 [==============================] - 15s 240ms/step - loss: 0.8748 - accuracy: 0.6845 - val_loss: 1.3227 - val_accuracy: 0.5680
Epoch 19/20
63/63 [==============================] - 15s 241ms/step - loss: 0.8398 - accuracy: 0.6990 - val_loss: 1.2952 - val_accuracy: 0.5680
Epoch 20/20
63/63 [==============================] - 15s 240ms/step - loss: 0.7836 - accuracy: 0.7180 - val_loss: 1.3732 - val_accuracy: 0.5920
```

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.

Fully connected

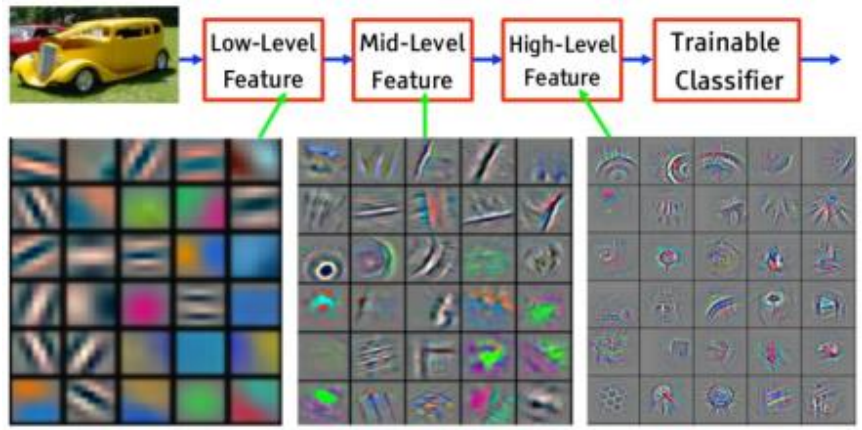
1000x1000
1m hidden units
#param = 10^{12}

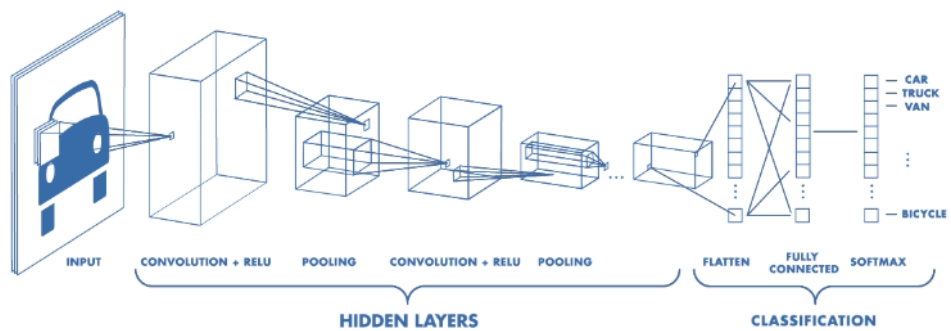
Locally connected

1000x1000
1m hidden units
100x100 filter
#param =10^{10}

Convolution neural networks

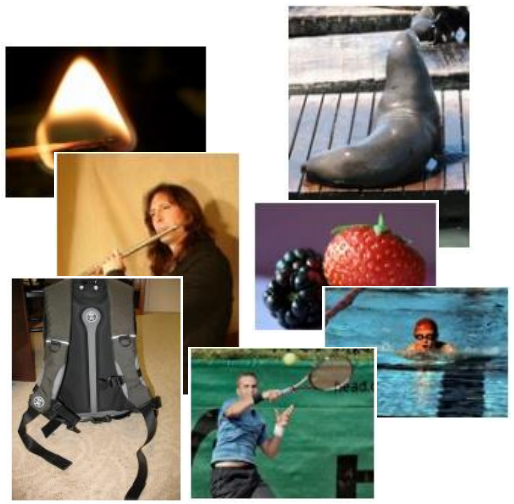
- **Motivation**: vision processing in the brain is fast
  - Simple cells detect local features
  - Complex cells pool local features
- **Technical**: sparse interactions and sparse weights reduce #params.
  - Parameter sharing: using a kernel with same set of weights while applying onto different location
  - 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

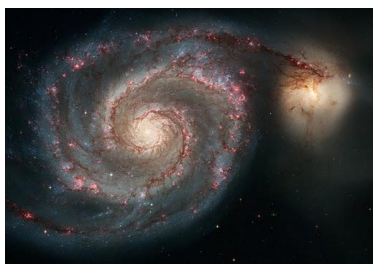




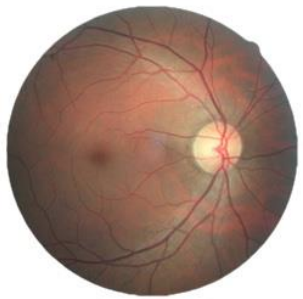
Factors for successful CNNs

- 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



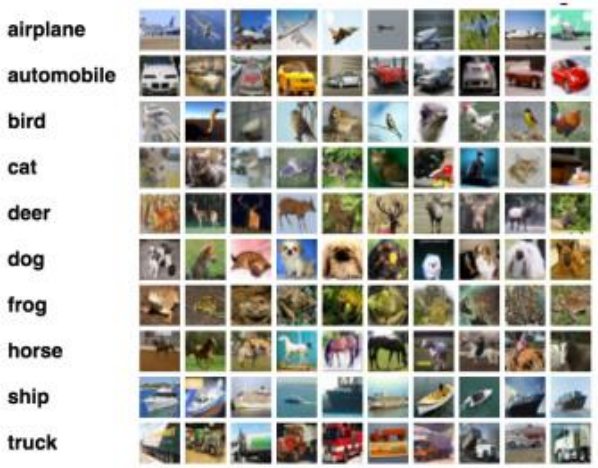
Diabetes recognition from retina image

Datasets for CNNs

Some datasets for computer vision

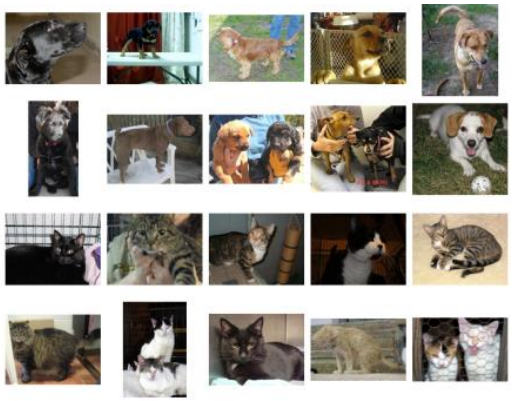
MNIST dataset
#training: 50,000, #testing: 10,000, #classes: 10

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

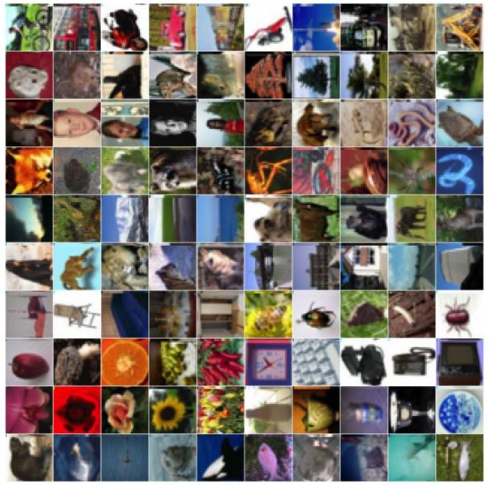


Animals: Cat, Dog dataset
Kaggle competition
#examples: 25,000, #classes: 2

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

IMAGENET 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.

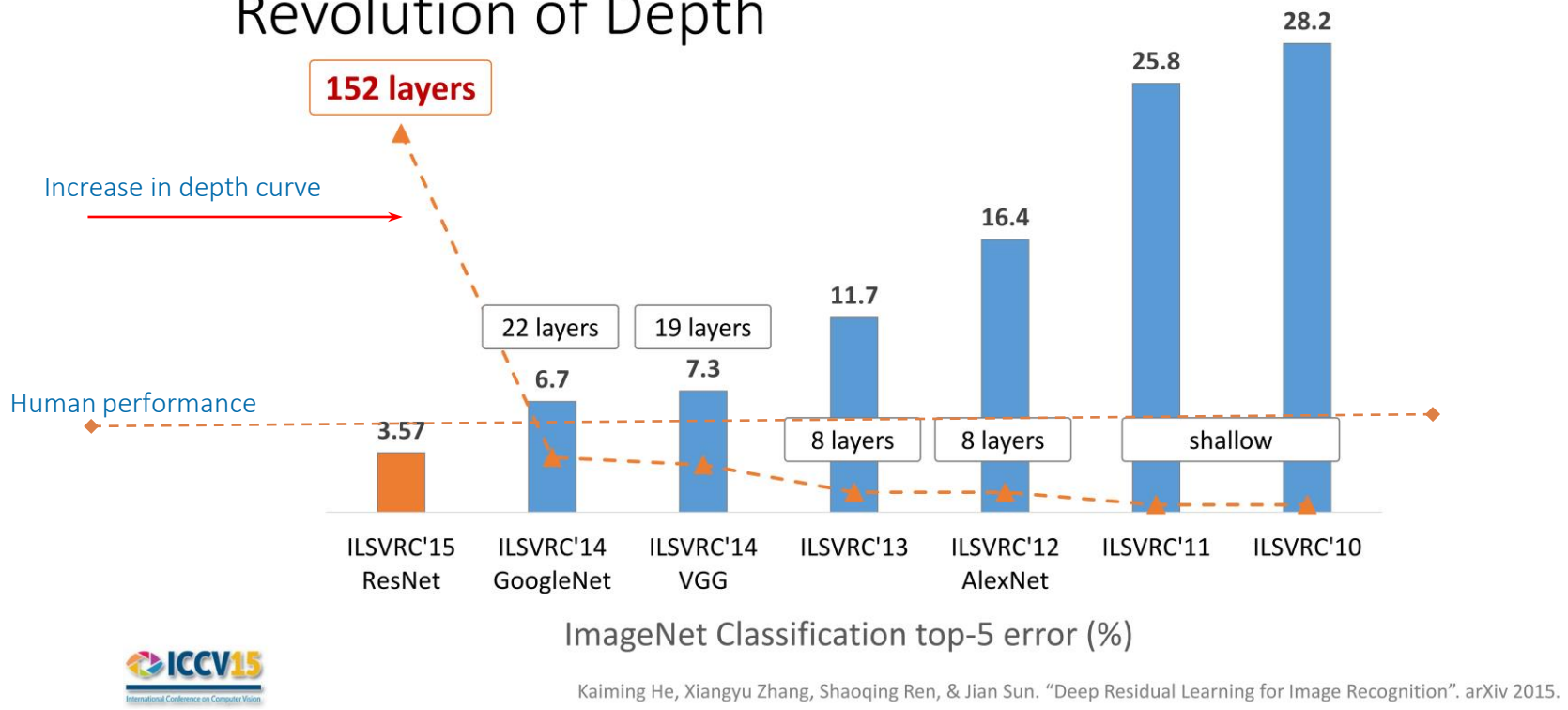
Introduction

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 workshoR-

1. Object localization for 1000 categories.
2. Object detection for 200 fully labeled categories.
3. Object detection from video for 30 fully labeled categories.

Microsoft Research

Revolution of Depth



ICCV15 International Conference on Computer Vision

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

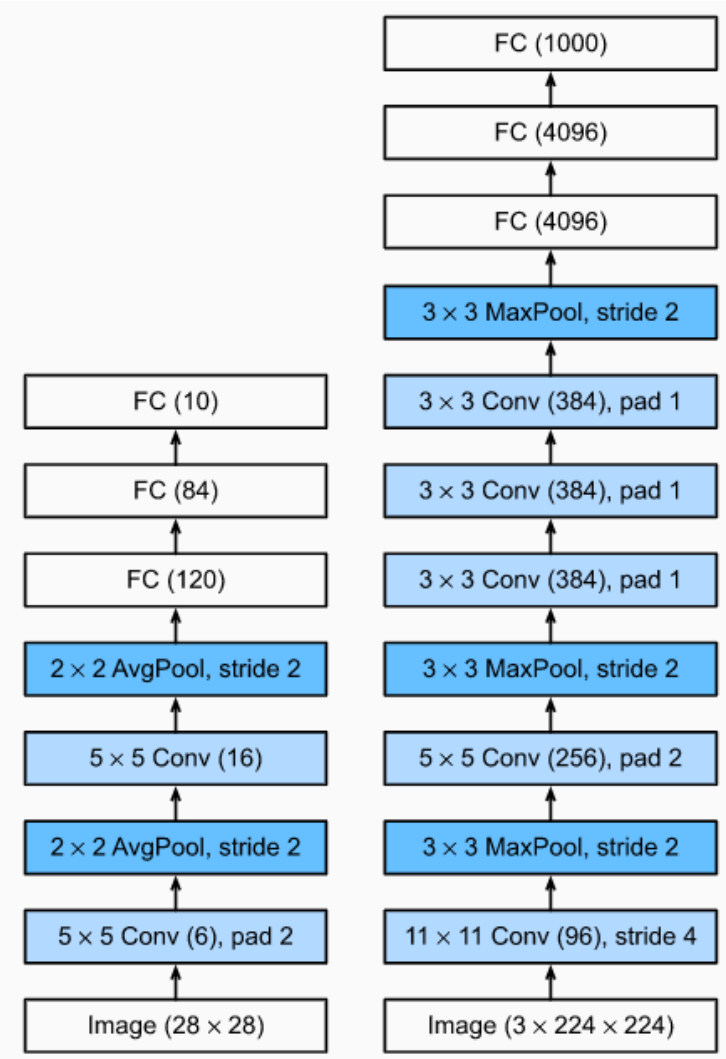
And … ResNet 2015 beats human performance (5.1%)!

Some popular CNN architectures

LeNet and AlexNet

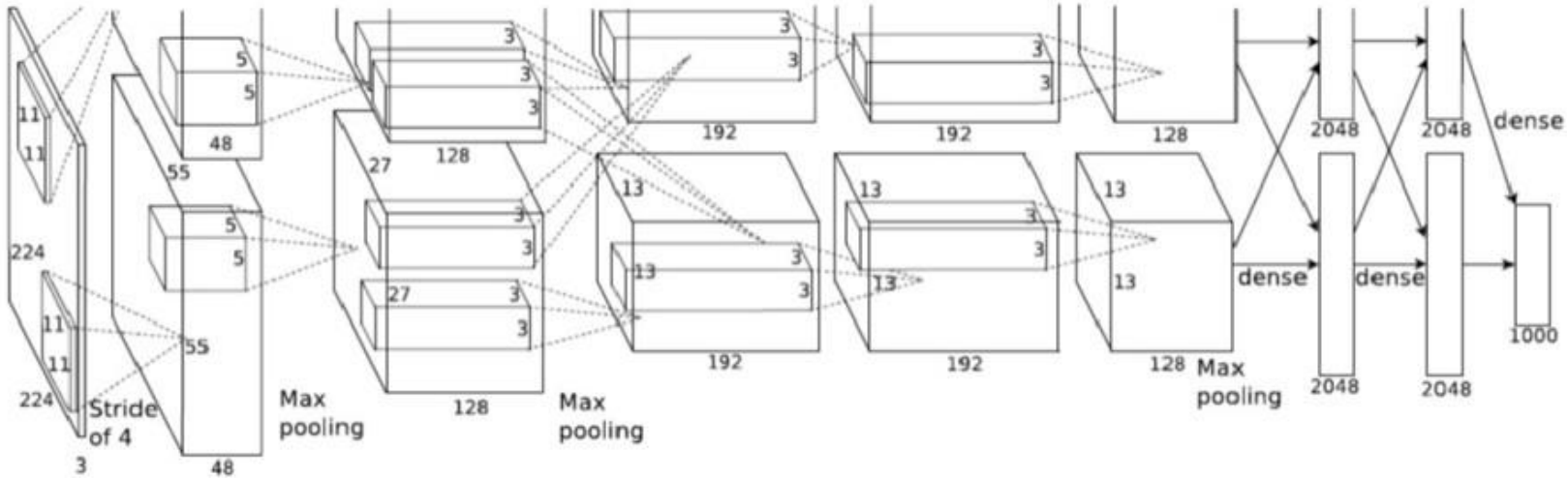
- The **design philosophies** of AlexNet and LeNet are **very similar,** but there are also **significant differences**.
  - AlexNet is much **deeper than** the comparatively small LeNet5.
  - AlexNet consists of **eight layers**
    - Five convolutional layers, two fully-connected hidden layers, and 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 x 224/4 x 96 = 55 x 55 x 96 (because of stride 4)

Layer 2

Layer 2 is a Max Pooling Followed by Convolution

Input – 55 x 55 x 96

Max pooling – 55/2 x 55/2 x 96 = 27 x 27 x 96

Number of filters – 256

Filter size – 5 x 5 x 48

Layer 2 Output

27 x 27 x 256

Layer 3,4,5

Layers 3, 4 & 5 follow on similar lines

Layer 6

Layer 6 is fully connected

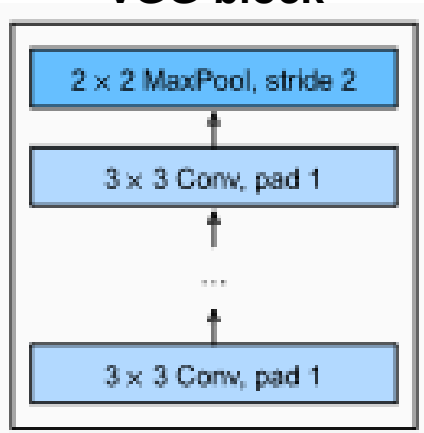
Input – 13 x 13 x 128 – > is transformed into a vector

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

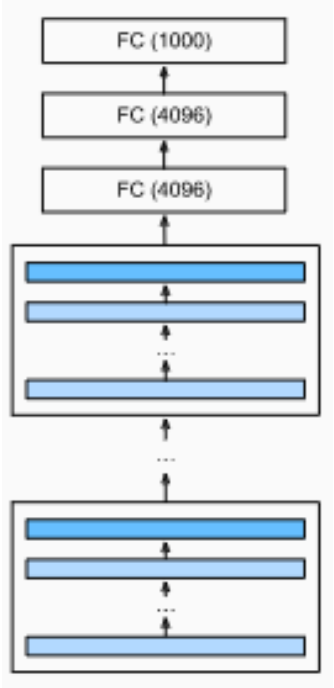
VGG Network

- **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.

VGG block



VGG network



(Source: Dive into DL)

MiniVGG for Cifar10

Our Tutorial

Layer Type	Output Size	Filter Size / Stride
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	$16 \times 16 \times 32$	2×2
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	$8 \times 8 \times 64$	2×2
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!