FIT3181 Deep Learning

Week 06:Advanced 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

- Deep learning before and after 2012
- Revision of building-blocks, rationales and drawbacks of CNNs
- SOTA Convolutional Neural Networks
- Visualization in CNNs
- Adversarial Machine Learning
  - Questions 3.9 and 3.10 in Assignment 1

- Further Reading Recommendation
  - [FDL, ch05], [HandsOn-2nd, ch14], [Dive Into Deep Learning, ch07]
  - "CS231n: Convolutional Neural Networks for Visual Recognition", Stanford University, 2016

Deep Learning before and after 2012

Deep learning renaissance

Before 2012

- Some **milestones** before 2012
  - Backprop was proposed in 1986
  - CNN was proposed in 1995
- People **did not believe** that NNs work
  - Could not train NNs for real-world datasets
  - Easy to get overfitting
- In machine learning (computer vision), **hand-crafted feature approach** dominated
  - SIFT (scale-invariant feature transform), SURF (speeded up robust features)

After 2012

- **AlexNet** won **ImageNet competition**
- **Missing Ingredient:** Data
  - In 2009, **the ImageNet dataset** was released, challenging researchers to learn models **from 1 million examples, 1000 each** from **1000 distinct categories** of objects
- **Missing Ingredient:** Hardware
  - **Graphical processing units (GPUs)** proved to be a game changer in making deep learning feasible.
  - These chips had long been developed to **optimize** for high throughput **4×4 matrix-vector products**, which are needed for many computer graphics tasks and **calculate convolutional layers**
- The **renaissance** of Deep Learning

Hand-crafted feature

Visual data

Learning algorithm and feature extractor are disconnected !

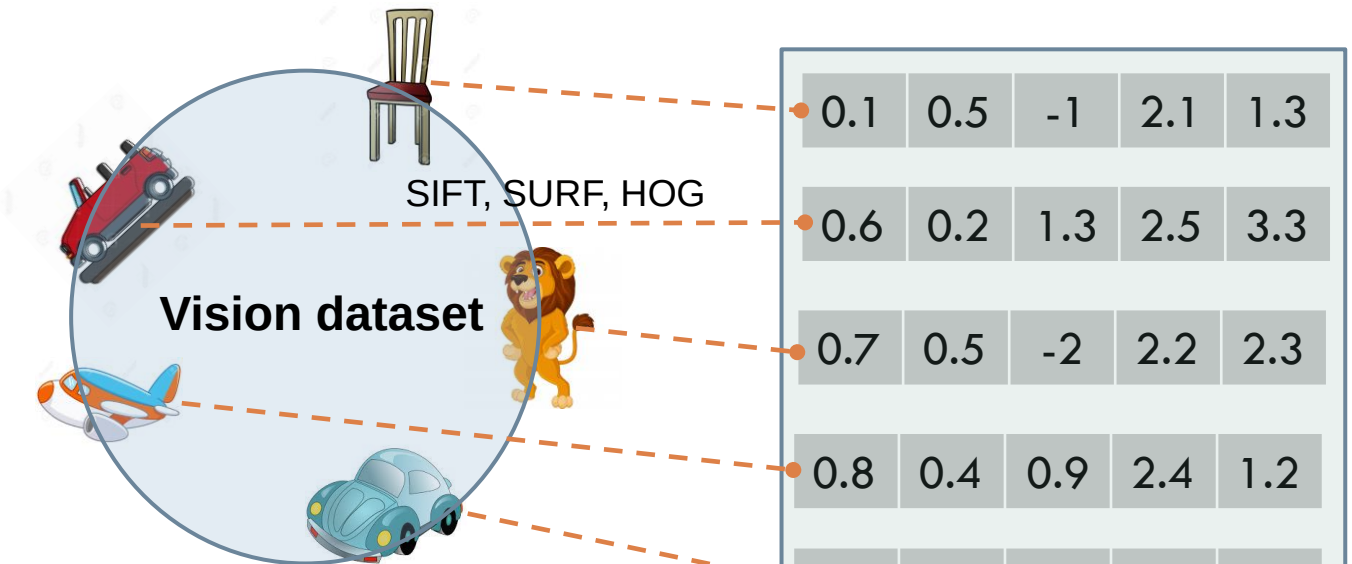
- **Traditional approach** (hand-crafted feature learning)

Input

Feature extraction (SIFT, SURF, HOG)

Hand-crafted features

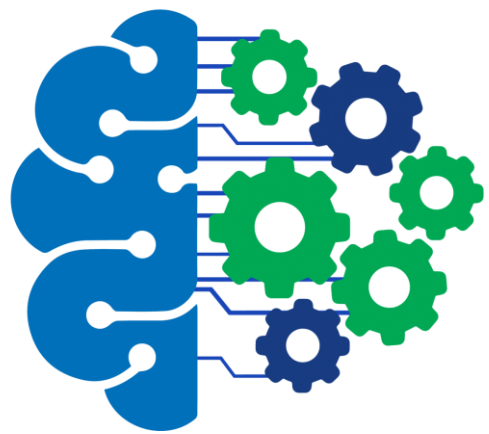
Learning algorithms (SVM, XGBoost)



0.1	0.5	-1	2.1	1.3
0.6	0.2	1.3	2.5	3.3
0.7	0.5	-2	2.2	2.3
0.8	0.4	0.9	2.4	1.2
0.6	0.9	1.5	2.0	1.6

Feature vectors

[Source: Internet]



Prediction

ML model

(SVM, k-NN, decision tree, XG Boost)

Automatic feature extractor

Visual data

- **Deep learning approach** (automatic feature learning)

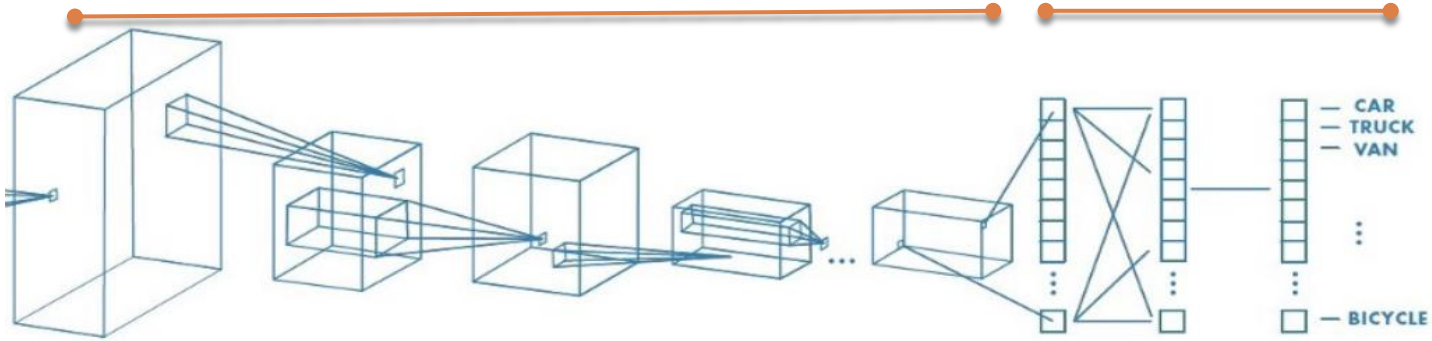
Classification and **feature extractor** are **connected** !

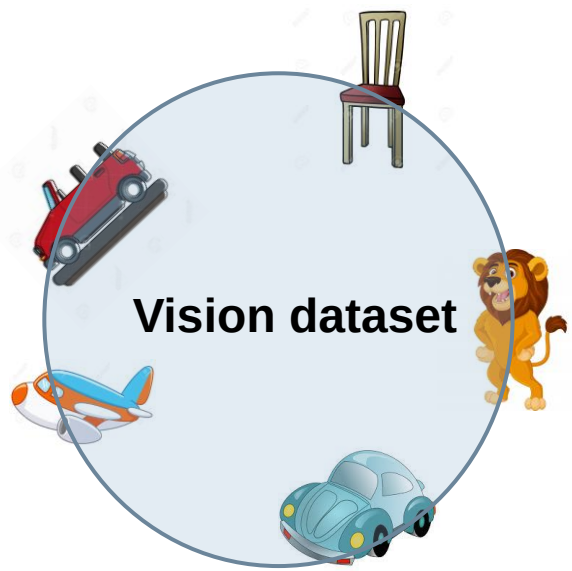
Input

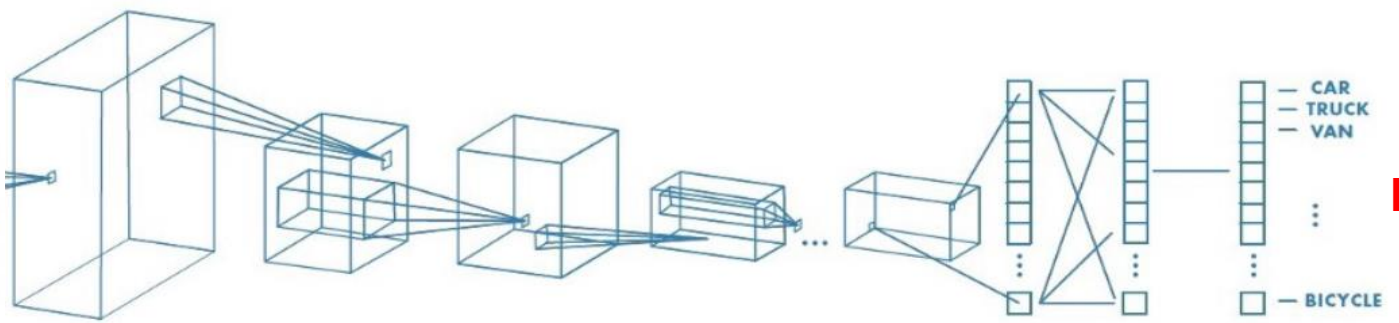
Feature extraction

Automatic features

Classification







Prediction

Hand-crafted and automatic feature extractor

Visual data

Learning algorithm and feature extractor are disconnected !

- **Traditional approach** (hand-crafted feature learning)

Input → **Feature extraction** (SIFT, SURF, HOG) → **Learning algorithms** (SVM, XGBoost)

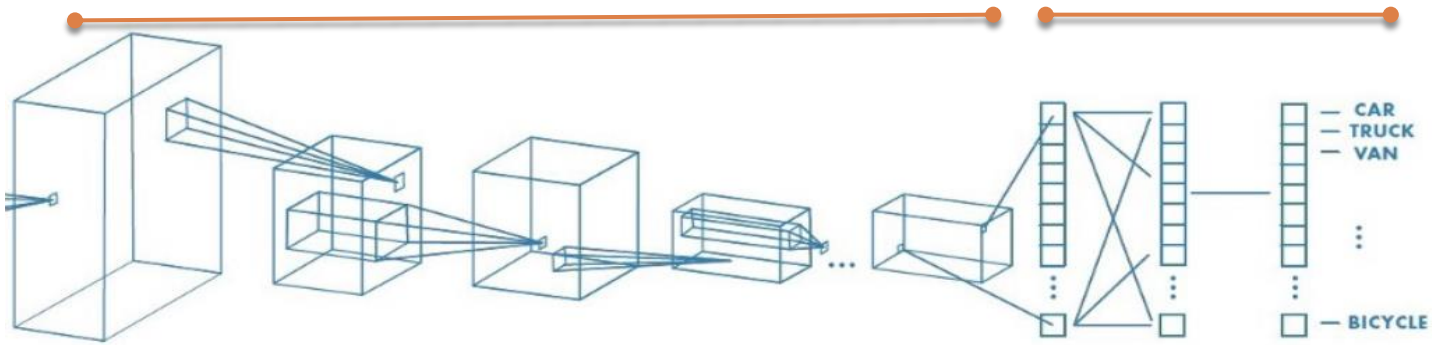
Hand-crafted features

Classification and feature extractor are connected !

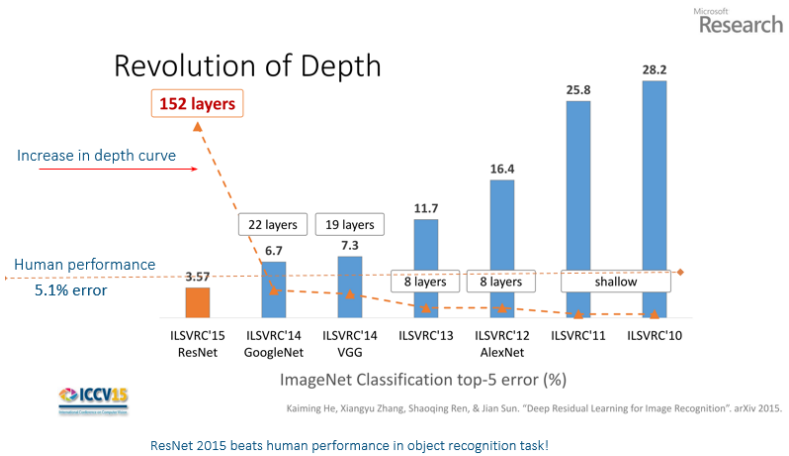
- **Deep learning approach** (automatic feature learning)

Input → **Feature extraction** → **Classification**

Automatic features

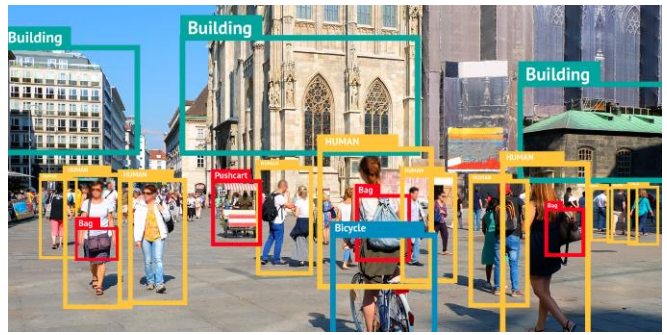


Achievements of CNNs



ResNet 2015 beats human performance in object recognition task!

Win human in object recognition on ImageNet dataset

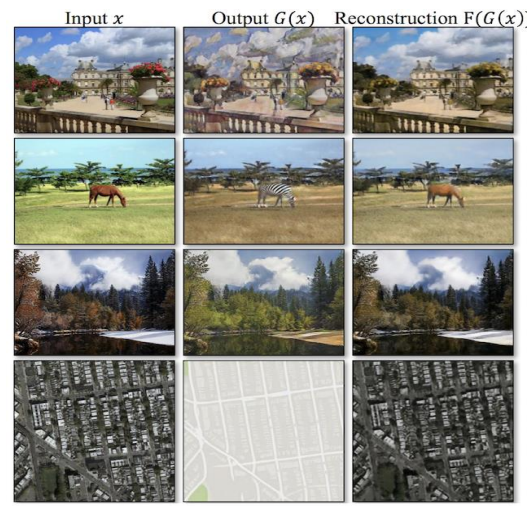


SOTA in object detection and semantic segmentation

Some fiction and imaginary tasks

Image captioning

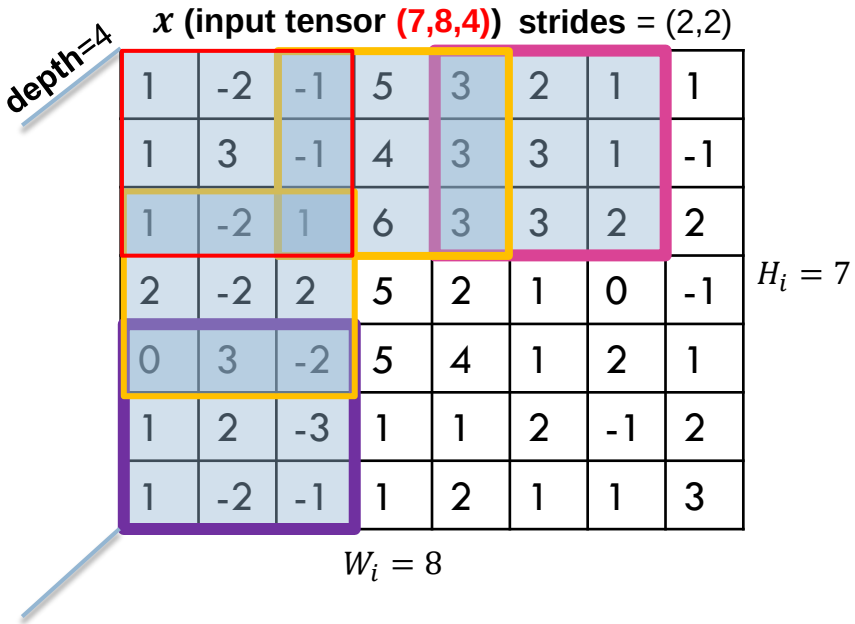
Image generation from Progressive GAN

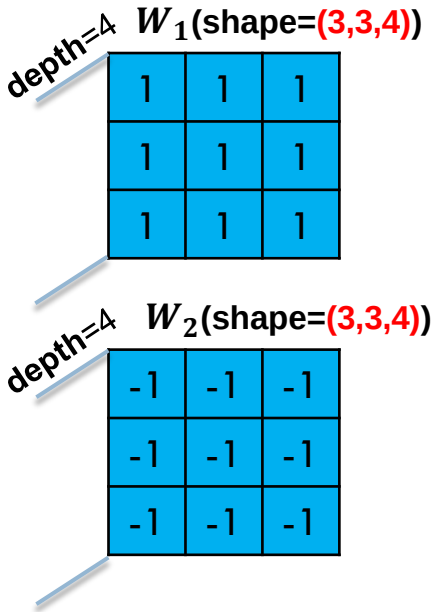


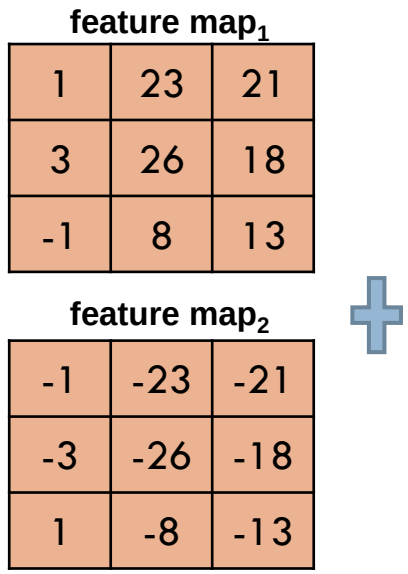
Style transferred with CycleGAN

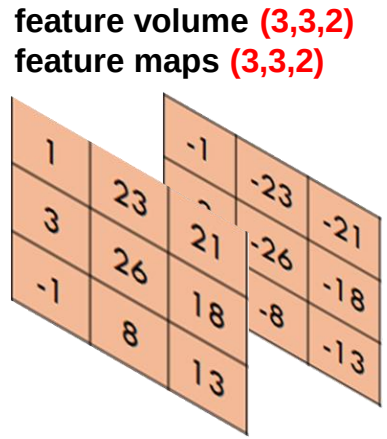
Revision of CNN building-blocks and How CNN works?

Convolution layer with multiple filters and feature maps

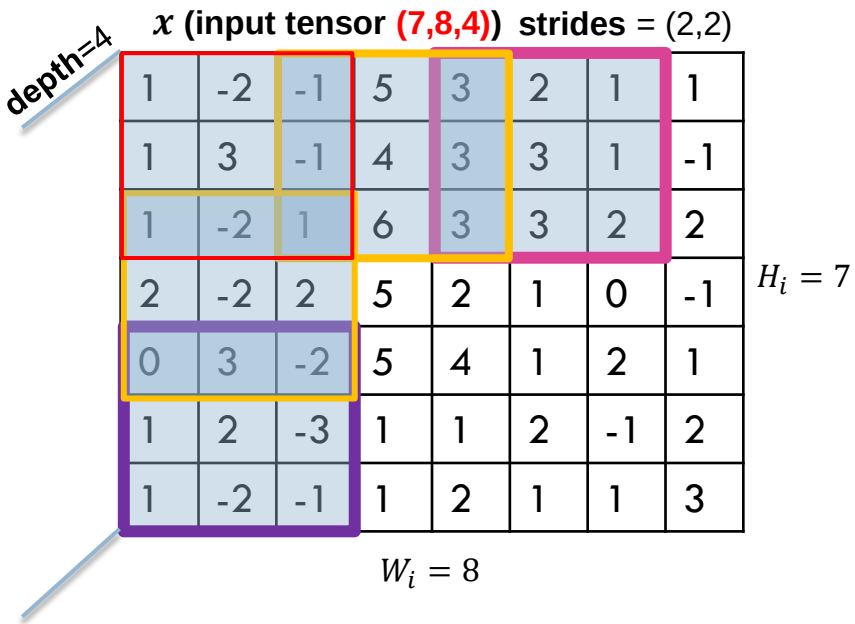


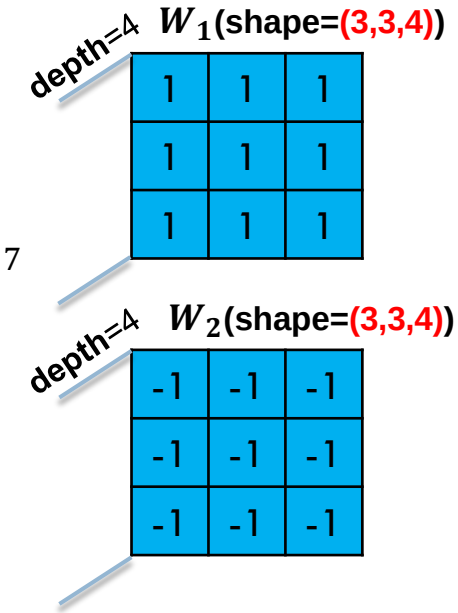


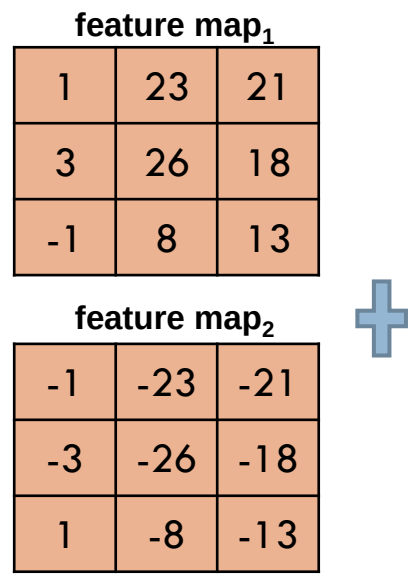


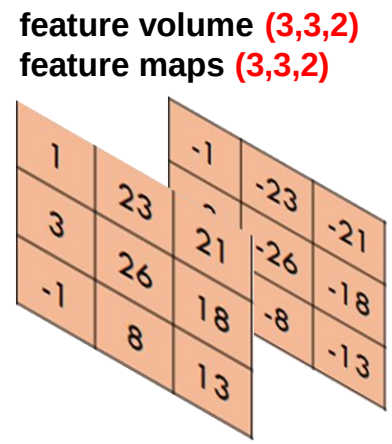


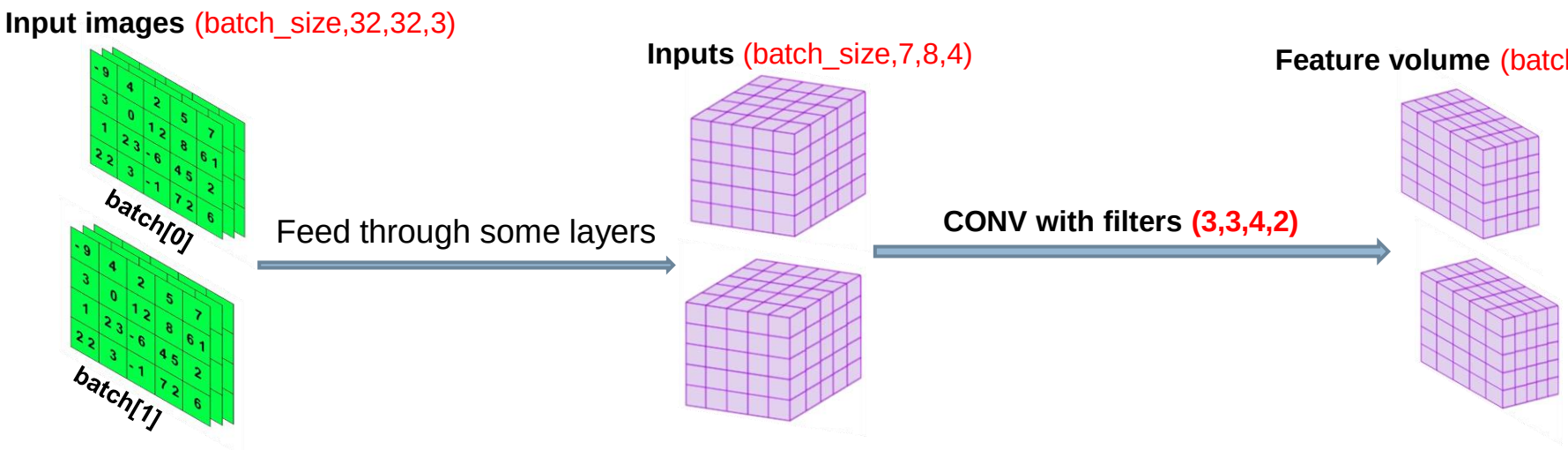
Convolution layer with multiple filters and feature maps



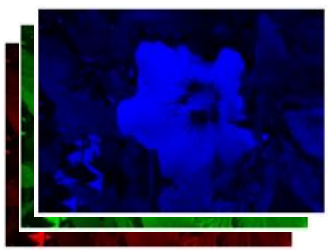




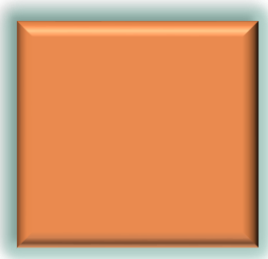




CNN in Operation



[32,32,3]



Input Layer

Filters [4,4,3,4]

Filter 1

[4,4,3]

Filter 2

[4,4,3]

Filter 3

[4,4,3]

Filter 4

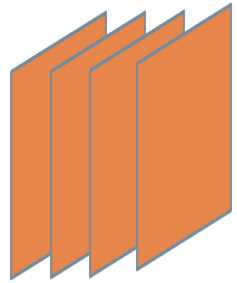
[4,4,3]

Conv2D 1

padding= valid
strides = (2,2)

$$out\_size = \left\lfloor \frac{32-4}{2} \right\rfloor + 1 = 15$$

Feature volume
Feature maps



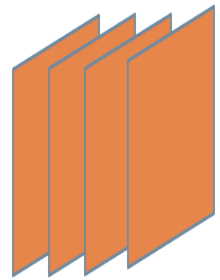
[15,15,4]

Latent representation

Pooling layer

pool-size=(2,2)
strides= (2,2)
padding= same

Feature volume
Feature maps



[8,8,4]

Latent representation

FC layer

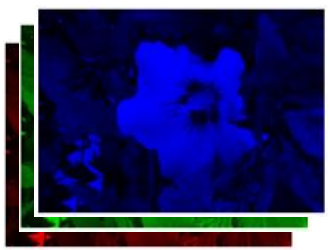
8x8x4 neurons

softmax

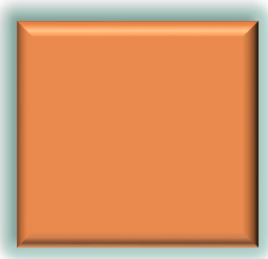
Output layer

10 neurons
for 10 classes

CNN in Operation



[32,32,3]



Input Layer

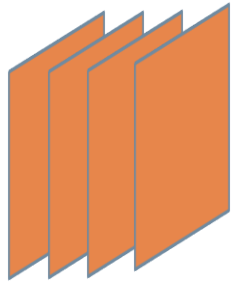
Filters [4,4,3,4]

Conv2D 1

padding= valid
strides = (2,2)

$$out\_size = \left\lfloor \frac{32-4}{2} \right\rfloor + 1 = 15$$

Feature volume
Feature maps



[15,15,4]

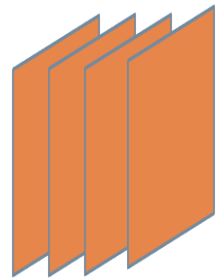
Latent representation

Pooling layer

pool-size=(2,2)
strides= (2,2)
padding= same

$$out\_size = \left\lfloor \frac{15-1}{2} \right\rfloor + 1 = 8$$

Feature volume
Feature maps

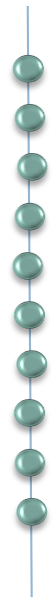


[8,8,4]

Latent representation

FC layer
8x8x4 neurons

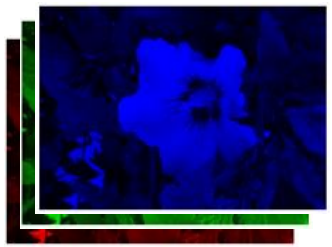
softmax



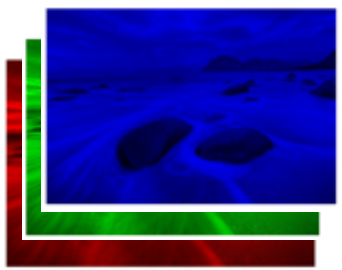
Output layer

10 neurons
for 10 classes

CNN in Operation



[32,32,3]



[32,32,3]

[2,32,32,3]

Input Layer

Filters [4,4,3,4]

Filter 1

[4,4,3]

Filter 2

[4,4,3]

Filter 3

[4,4,3]

Filter 4

[4,4,3]

Conv2D 1

padding= valid
strides = (2,2)

Feature volume
Feature maps
[2,15,15,4]

[15,15,4]

[15,15,4]

Pooling layer

pool-size=(2,2)
strides= (2,2)
padding= same

Feature volume
Feature maps
[2,8,8,4]

[8,8,4]

[8,8,4]

8x8x4

2

10

2

softmax

Output layer

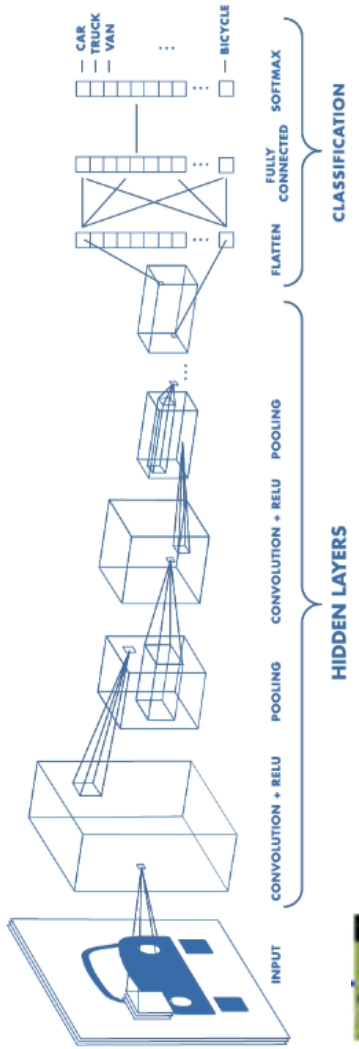
10 neurons
for 10 classes

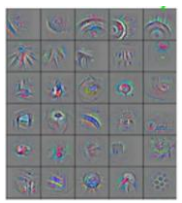
FC layer

8x8x4 neurons

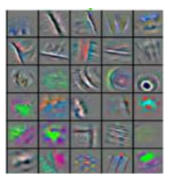
Automatic feature extractor

Deep learning for visual data





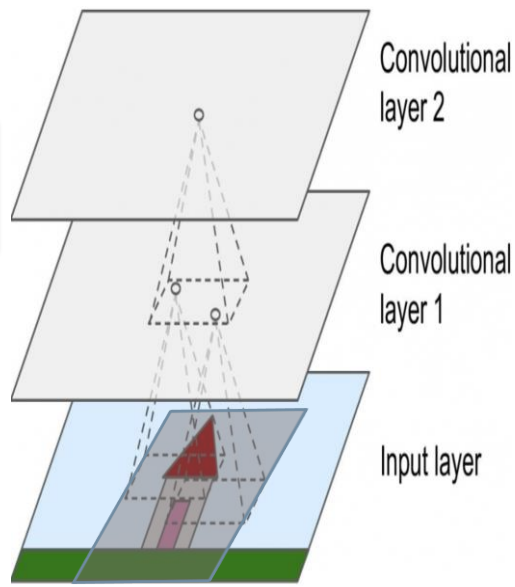
High-level feature/representation
- objects

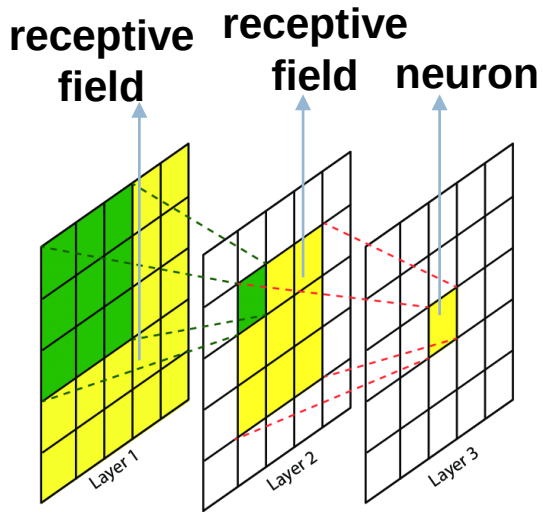


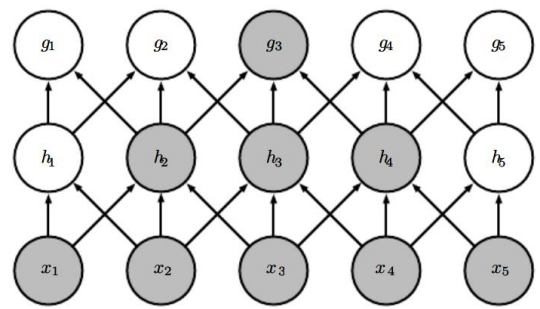
Mid-level feature/representation
- circles, triangles, boxes

Low-level feature/representation
- edges, pixels, corners

Neurons on **higher layers** have **larger receptive fields** (patches) on **input images**

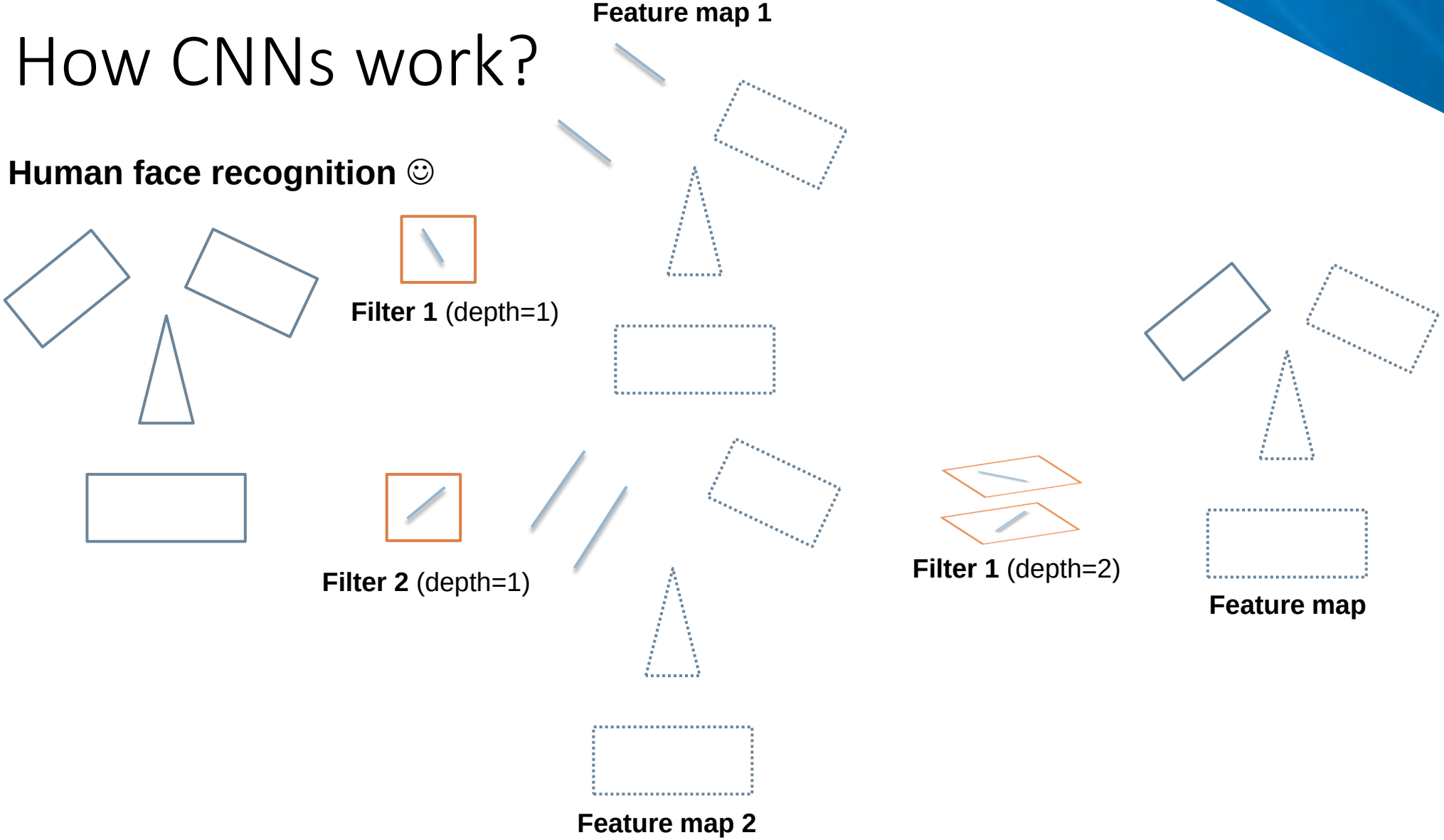






How CNNs work?

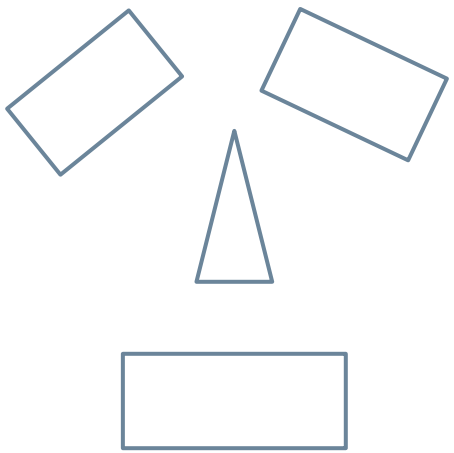
Human face recognition ☺



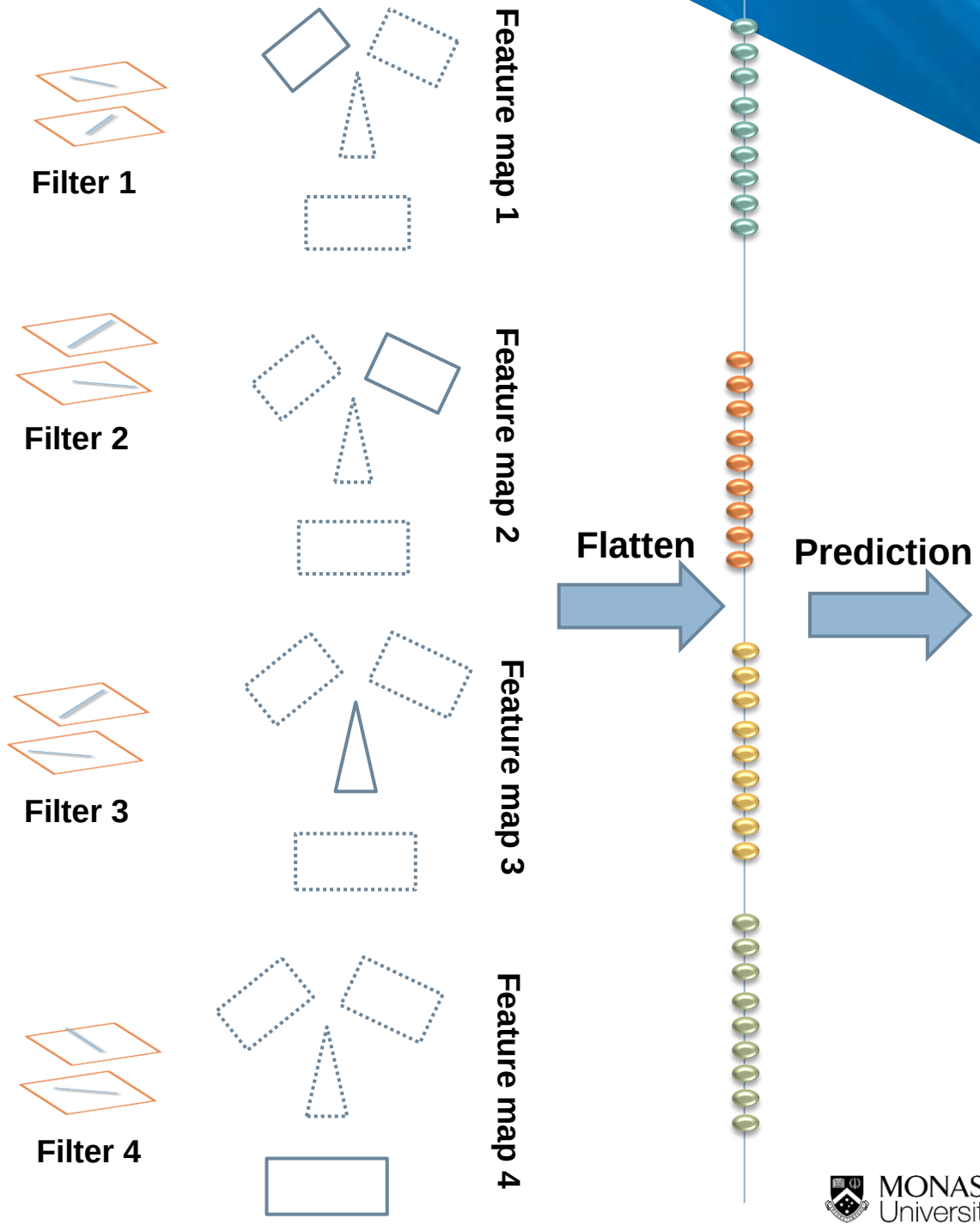
- Higher level filters **combine** lower level filters to represent **more abstract objects**
  - The context is **locally expanded**

How CNNs work?

Human face recognition ☺

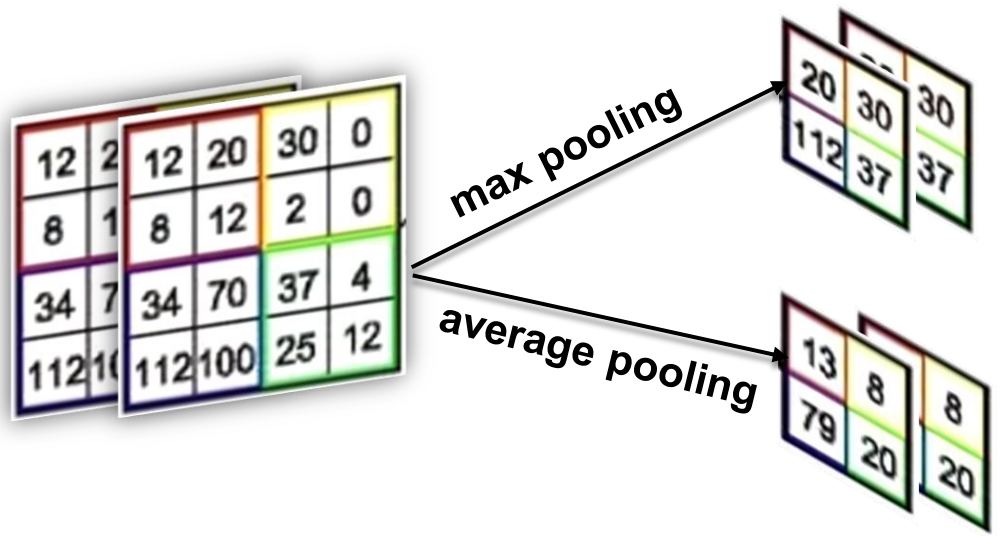


- CNN cannot capture **spatial relationships** among objects
  - How **spatially related** of two eyes, nose and eye, eye and mouth

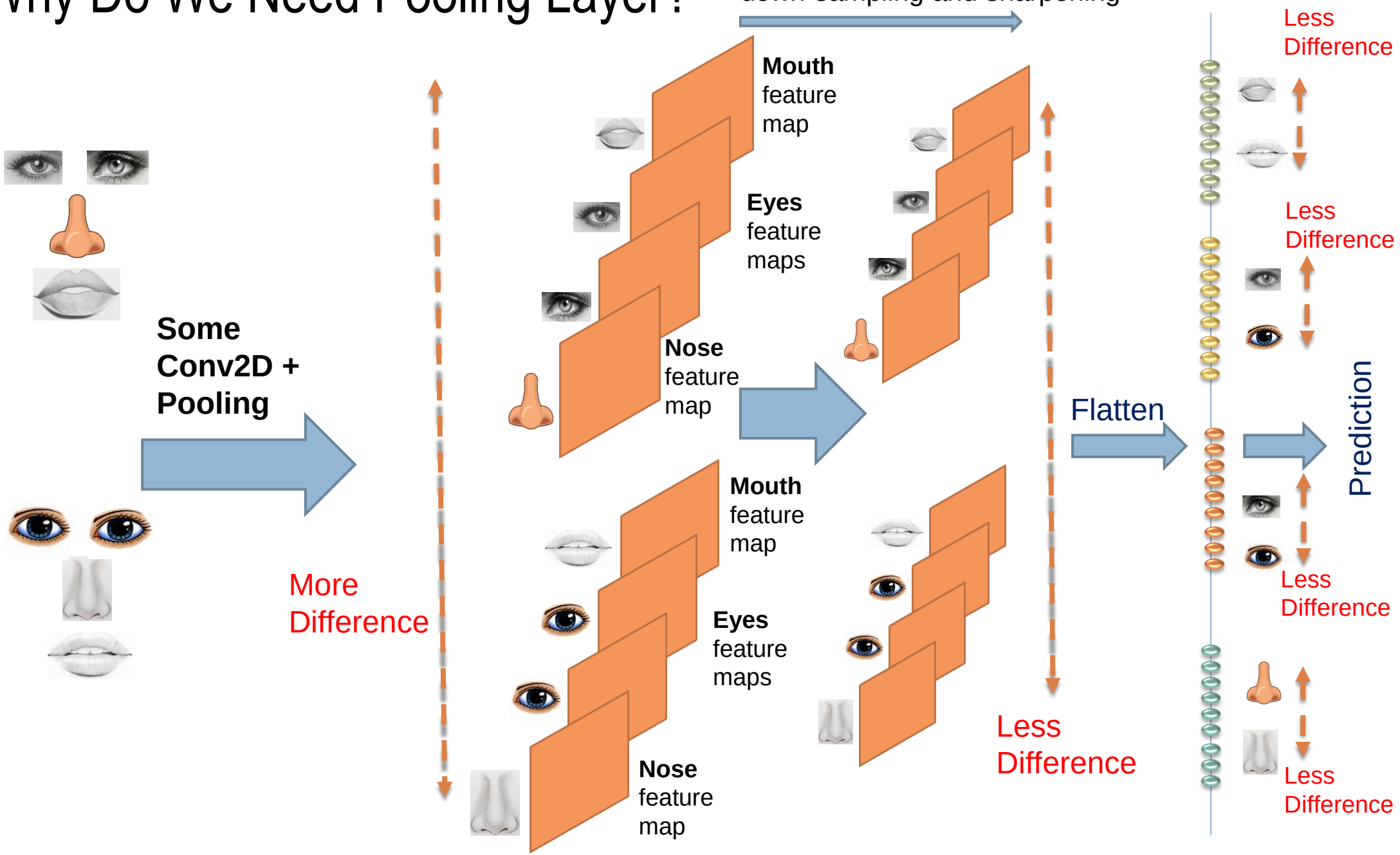


Pooling layer

- Max/average-pooling is **locally invariant** and applied **independently** to **each feature map**
- Reduce the size of feature map to be more manageable
  - Smaller size for flattening
- Reduce the difference of output tensors



Why Do We Need Pooling Layer?



Some comments from Hinton

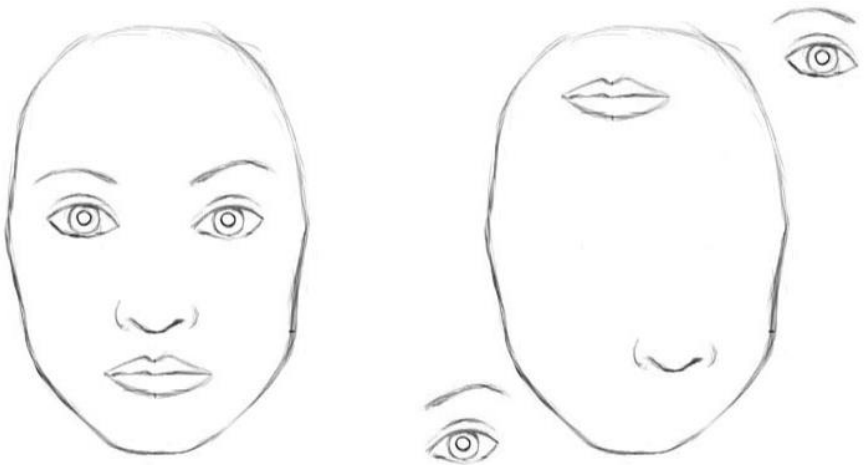
Dynamic Routing Between Capsules

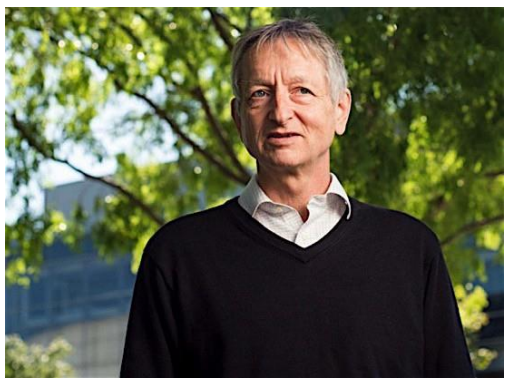
Sara Sabour **Nicholas Frosst**

Geoffrey E. Hinton
Google Brain
Toronto
{sasabour, frosst, geoffhinton}@google.com

Paper link: https://arxiv.org/pdf/1710.09829.pdf

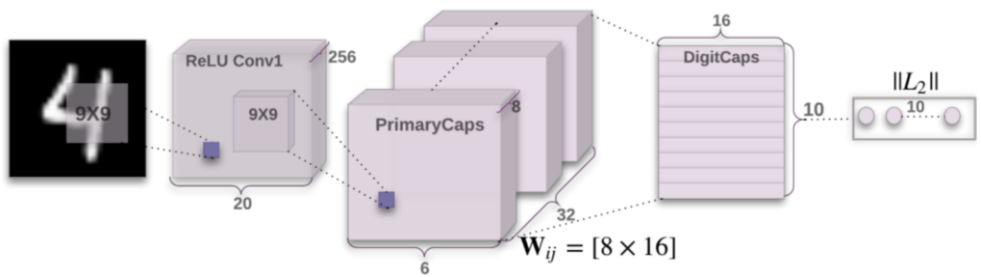
(Source: medium.com)





Hinton: "The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster."

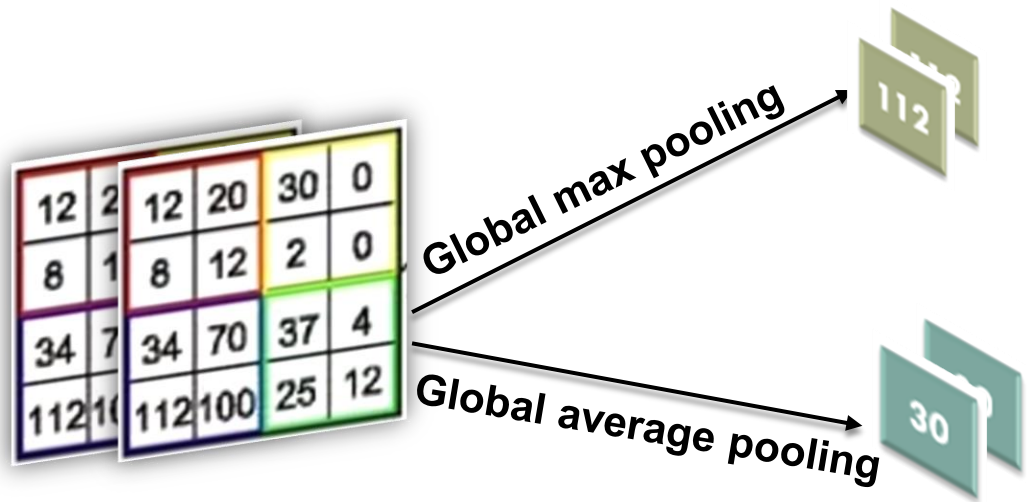
- CNNs cannot capture the **spatial relationships** among objects, hence it **wrongly predicts every face** with **two eyes**, **one nose**, and **one mouth** as a **human face**.



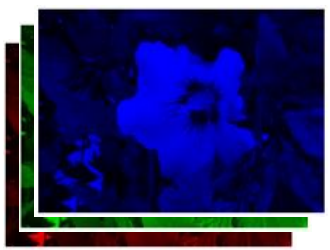
(Source: original paper of **Capsule Net**)

Global pooling layer

- Similar to **pooling layer**, but the **kernel size** is set to **input size**
  - **Each feature map** becomes a **single neuron**
  - **Global average pooling** (GAP)
  - **Global max pooling** (GMP)
  - **4D input tensor** [batch_size, in_height, in_width, in_depth] → **2D output tensor** [batch_size, in_depth]



Global Pooling Layer



[32,32,3]

Input Layer

Filters [4,4,3,4]

Filter 1

[4,4,3]

Filter 2

[4,4,3]

Filter 3

[4,4,3]

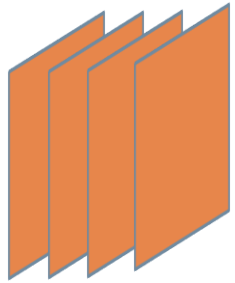
Filter 4

[4,4,3]

Conv2D 1

padding= valid
strides = (2,2)

Feature volume
Feature maps



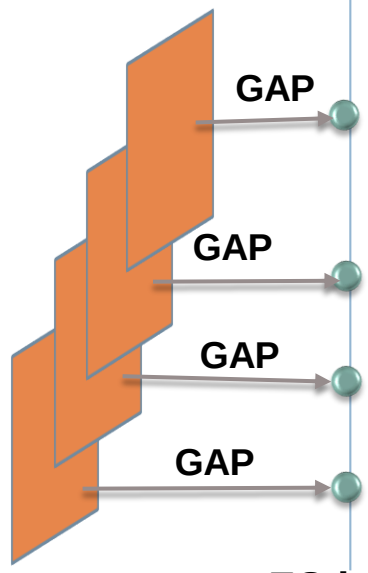
[15,15,4]

Latent representation

Pooling layer

pool-size=(2,2)
strides= (2,2)
padding= same

Feature volume
Feature maps



[8,8,4]

Latent representation

FC layer

4 neurons

softmax

Output layer

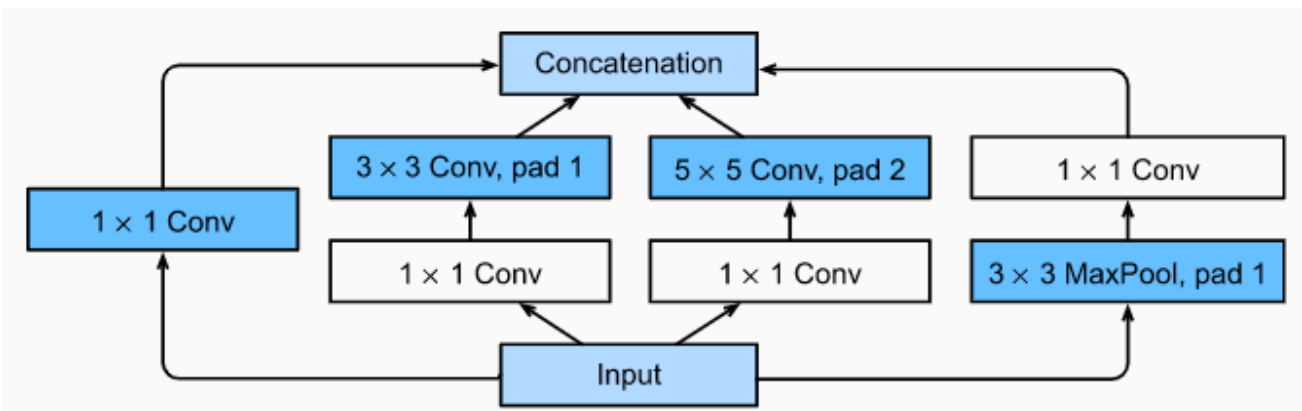
10 neurons
for 10 classes

SOTA CNNs

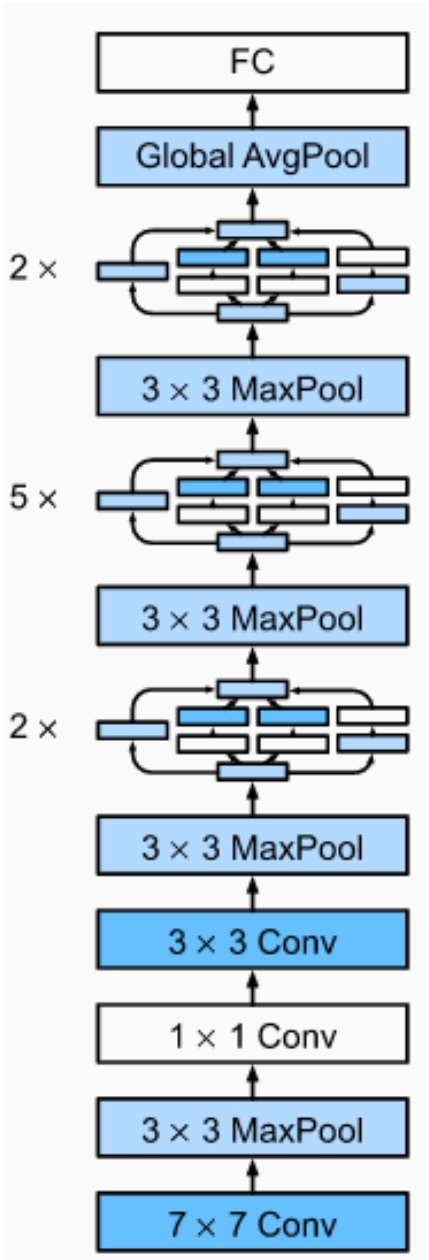
GoogLeNet

Szegedy et al. '14

- **Won ImageNet** competition **in 2014**.
- Address the question of **which sized** convolution kernels are **best**.
  - Previous popular networks employed choices as **small as 1×1** and as **large as 7×7**.
  - Sometimes it can be **advantageous** to employ a **combination** of **variously-sized kernels**.
- The building-block is **Inception block**



Inception block

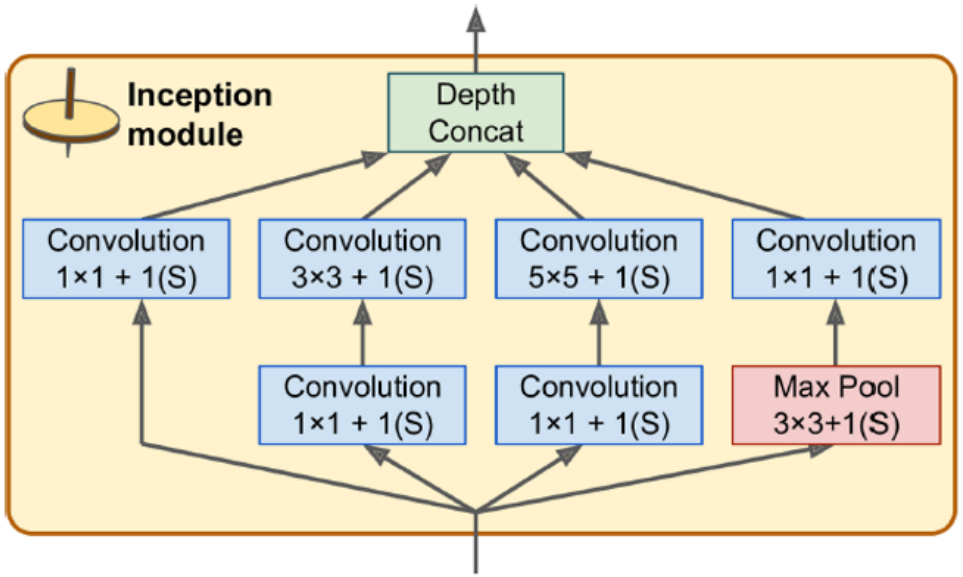


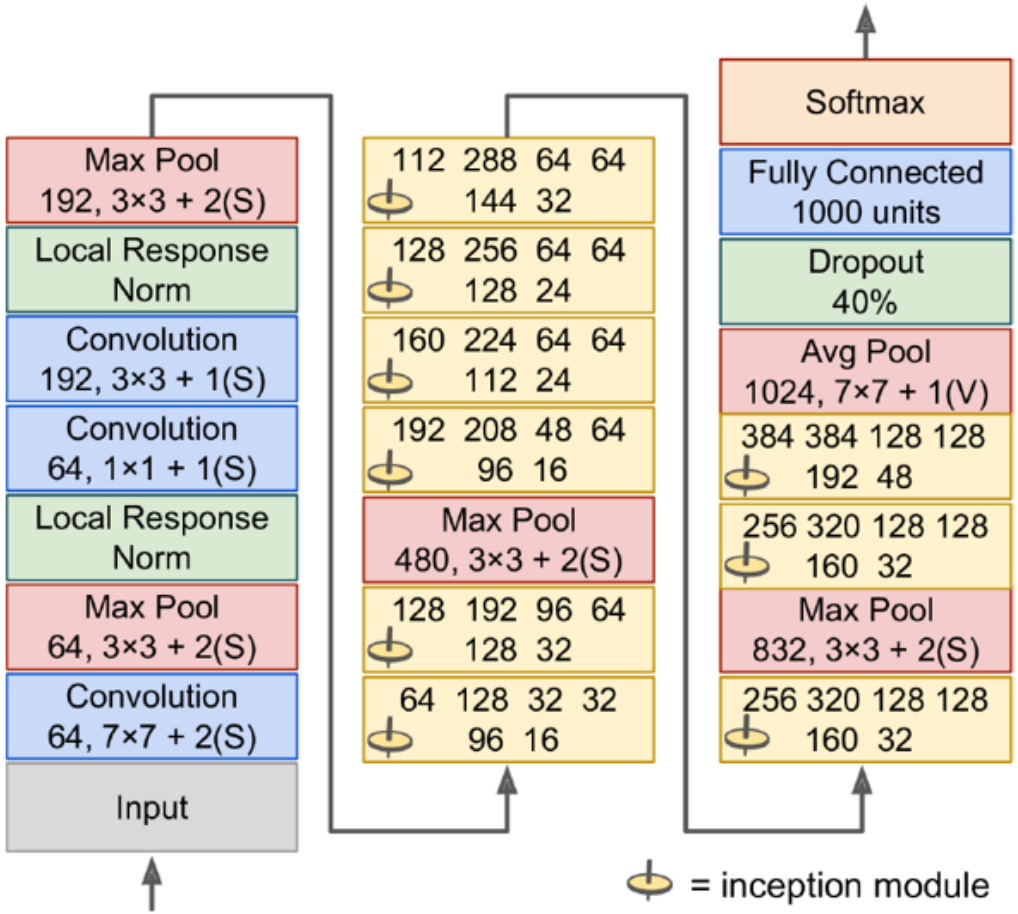
(Source:Dive into DL)

GoogLeNet

Szegedy et al. ‘14

22 layers

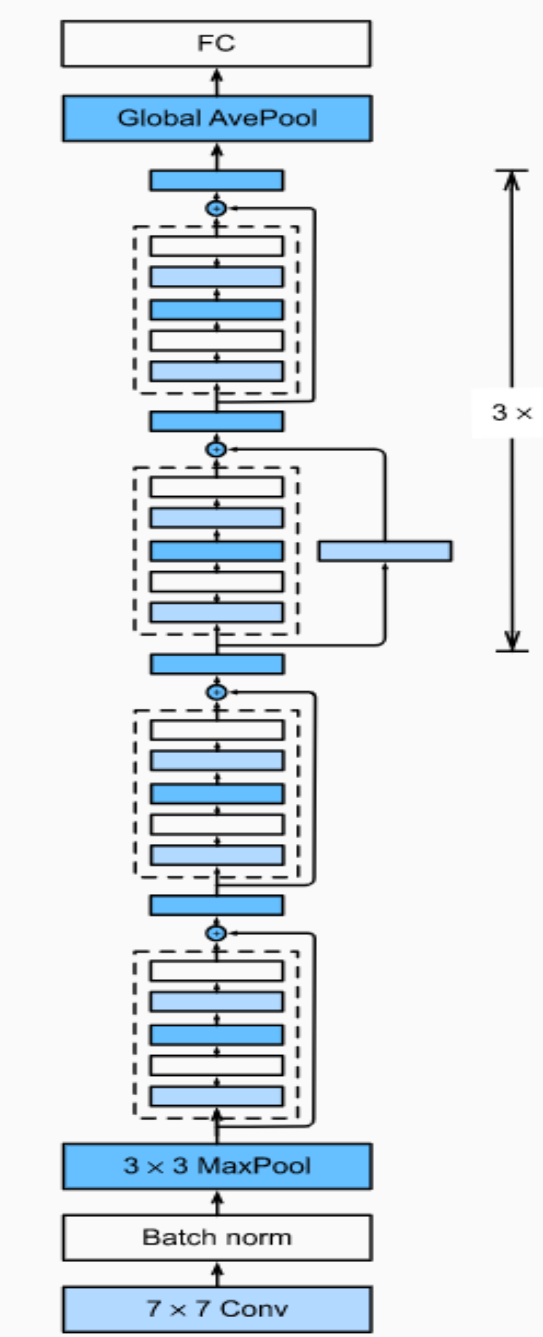




(Source: Hands On, Ch. 15)

ResNet

- **Won ImageNet** competition **in 2015**
- Many ResNet blocks
- Each ResNet block contains **many residual blocks**
  - Number of residual blocks
  - Number of channels for residual blocks
- Each ResNet block
  - From the **second block**, the first **residual block**
    - Use **1x1 Conv skip connection**
  - Otherwise
    - Use **standard skip connection**

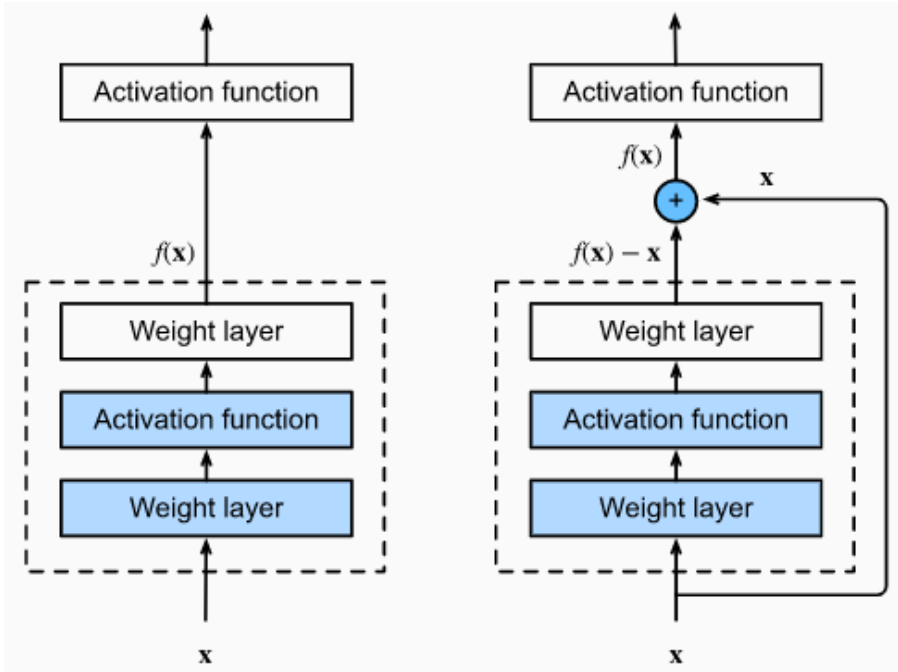


ResNet-18 architecture

ResNet

Residual Block

(Source:Dive into DL)

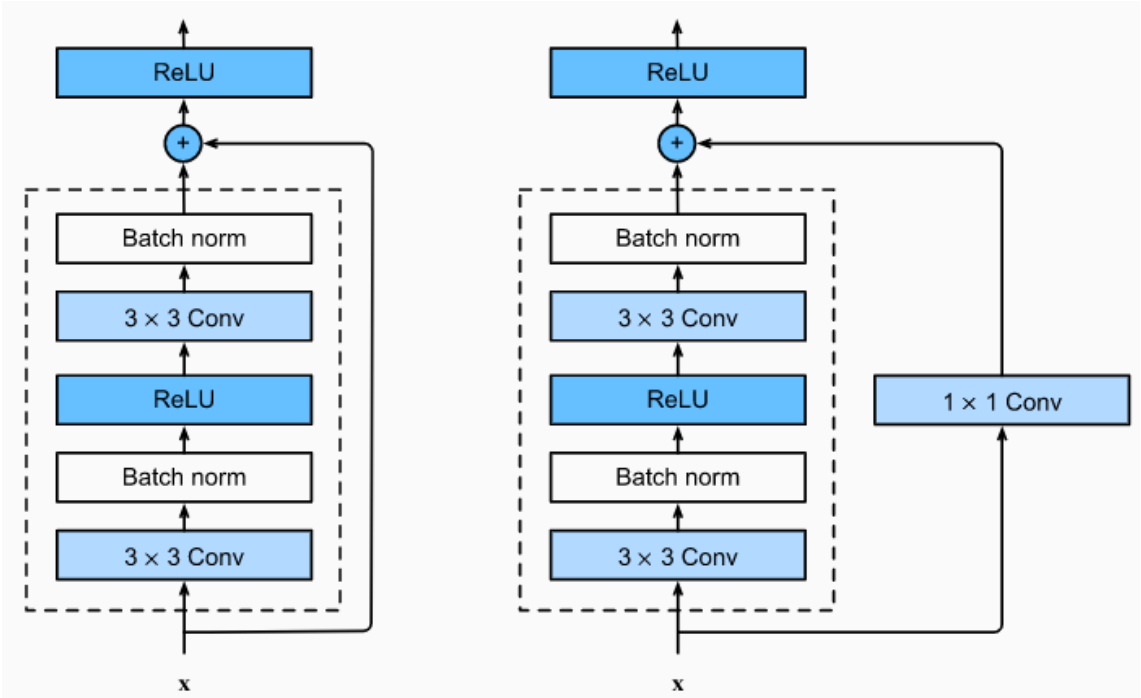


Without skip connection **With skip connection**

- Learn $f(x) = g(x) + x$ where $g(x) = f(x) - x$
  - The **model expressiveness** is the **same** as previous
  - The **gradient** $\nabla f(x) = \nabla g(x) + \mathbf{1}$ **looks better** where $\mathbf{1}$ is the vector of **all** 1 with the same shape as x

ResNet

Residual Block



Not use 1x1 Conv **Use 1x1 Conv**

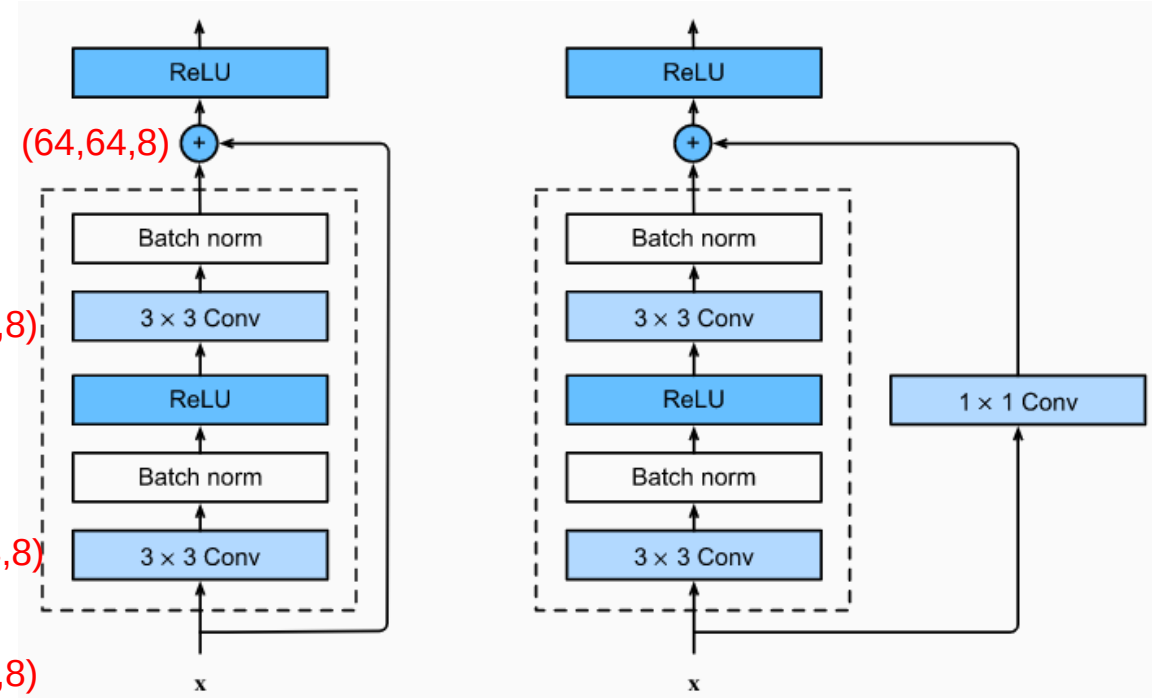
- ResNet follows VGG's full **3×3 convolutional layer** design.
  - The residual block has two **3×3 convolutional layers** with the **same number of output channels**.
  - Each **convolutional layer** is followed by a **batch normalization layer** and a **ReLU activation function**.
  - **Skip these two convolution operations** and **add the input directly** before the final ReLU activation function.
  - Requires that the output of **the two convolutional layers** must be of **the same shape** as the input, so that they can be added together.
    - If we want to change the number of channels, we need to introduce an additional **1×1 convolutional layer** to transform the input into the **desired shape** for the addition operation.

ResNet

Residual Block

padding=same
strides= (1,1)
#filters=8

padding=same
strides= (1,1)
#filters=8

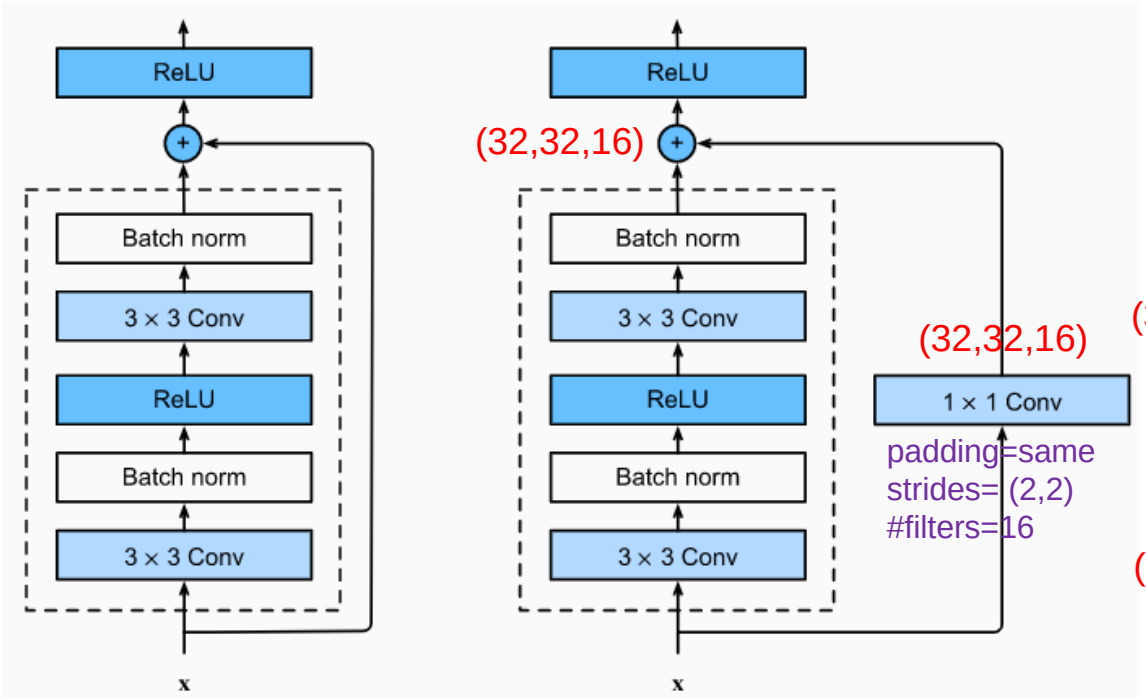


Not use 1x1 Conv **Use 1x1 Conv**

- ResNet follows VGG's full **3×3 convolutional layer** design.
  - The residual block has two **3×3 convolutional layers** with the **same number of output channels**.
  - Each **convolutional layer** is followed by a **batch normalization layer** and a **ReLU activation function**.
  - **Skip these two convolution operations** and **add the input directly** before the final ReLU activation function.
  - Requires that the output of **the two convolutional layers** must be of **the same shape** as the input, so that they can be added together.
    - If we want to change the number of channels, we need to introduce an additional **1×1 convolutional layer** to transform the input into the **desired shape** for the addition operation.

ResNet

Residual Block



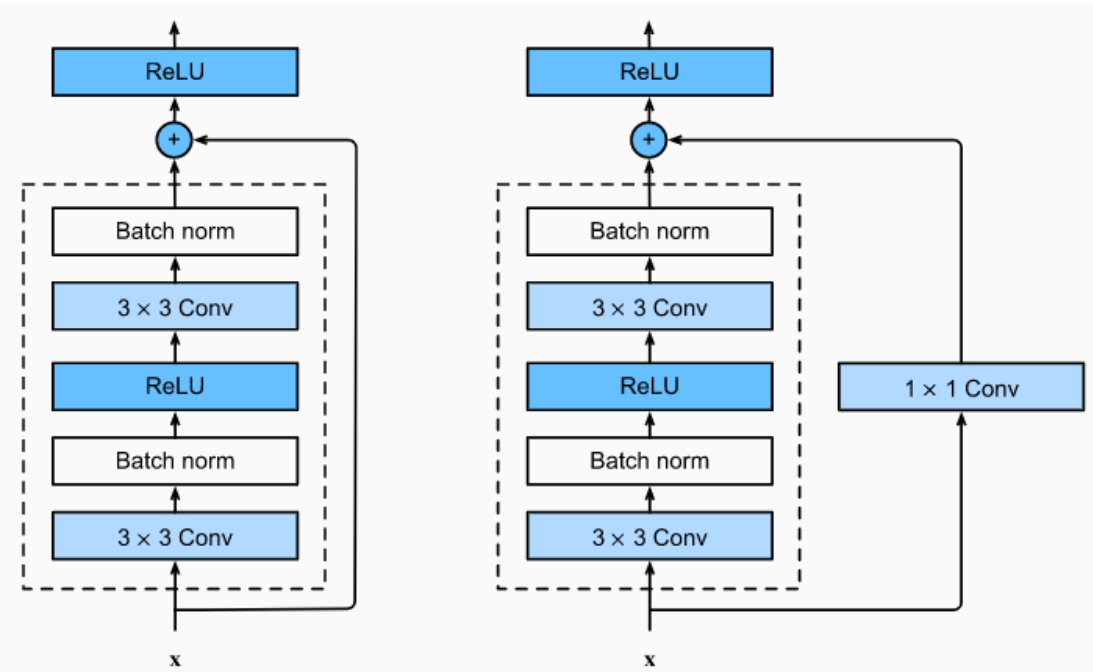
Not use 1x1 Conv **Use 1x1 Conv**

- ResNet follows VGG's full **3×3 convolutional layer** design.
  - The residual block has two **3×3 convolutional layers** with the **same number of output channels**.
  - Each **convolutional layer** is followed by a **batch normalization layer** and a **ReLU activation function**.
  - **Skip these two convolution operations** and **add the input directly** before the final ReLU activation function.
  - Requires that the output of **the two convolutional layers** must be of **the same shape** as the input, so that they can be added together.
    - If we want to change the number of channels, we need to introduce an additional **1×1 convolutional layer** to transform the input into the **desired shape** for the addition operation.

Implementation of Residual Block

```python
class Residual(tf.keras.Model):  #@save
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = tf.keras.layers.Conv2D(
            num_channels, padding='same', kernel_size=3, strides=strides)
        self.conv2 = tf.keras.layers.Conv2D(
            num_channels, kernel_size=3, padding='same')
        self.conv3 = None
        if use_1x1conv:
            self.conv3 = tf.keras.layers.Conv2D(
                num_channels, kernel_size=1, strides=strides)
        self.bn1 = tf.keras.layers.BatchNormalization()
        self.bn2 = tf.keras.layers.BatchNormalization()

    def call(self, X):
        Y = tf.keras.activations.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3 is not None:
            X = self.conv3(X)
        Y += X
        return tf.keras.activations.relu(Y)
```




```python
blk = Residual(3)
X = tf.random.uniform((4, 6, 6, 3))
Y = blk(X)
Y.shape
```

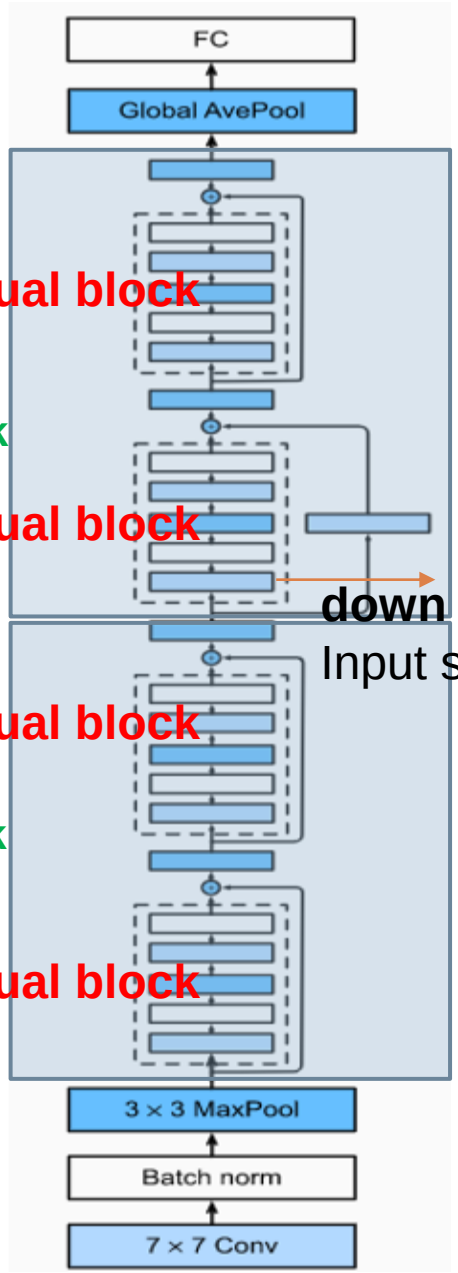
```
TensorShape([4, 6, 6, 3])
```

```python
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
TensorShape([4, 3, 3, 6])
```

(Source:Dive into DL)

Implementation of ResNet



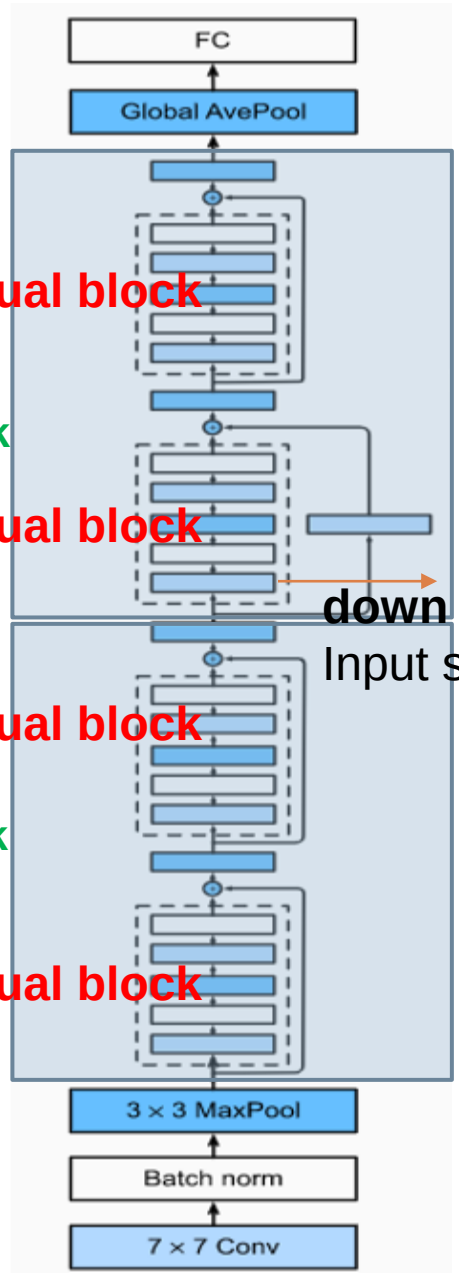
(Source:Dive into DL)

```python
class ResnetBlock(tf.keras.layers.Layer):
    def __init__(self, num_channels, num_residuals, first_block=False,
                 **kwargs):
        super(ResnetBlock, self).__init__(**kwargs)
        self.residual_layers = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                self.residual_layers.append(
                    Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                self.residual_layers.append(Residual(num_channels))

    def call(self, X):
        for layer in self.residual_layers.layers:
            X = layer(X)
        return X
```

```python
def net():
    return tf.keras.Sequential([
        # The following layers are the same as b1 that we created earlier
        tf.keras.layers.Conv2D(64, kernel_size=7, strides=2, padding='same'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Activation('relu'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same'),
        # The following layers are the same as b2, b3, b4, and b5 that we
        # created earlier
        ResnetBlock(64, 2, first_block=True),
        ResnetBlock(128, 2),
        ResnetBlock(256, 2),
        ResnetBlock(512, 2),
        tf.keras.layers.GlobalAvgPool2D(),
        tf.keras.layers.Dense(units=10)])
```

Implementation of ResNet



(Source:Dive into DL)

```
class ResnetBlock(tf.keras.layers.Layer):
    def __init__(self, num_channels, num_residuals, first_block=False,
                 **kwargs):
        super(ResnetBlock, self).__init__(**kwargs)
        self.residual_layers = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                self.residual_layers.append(
                    Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                self.residual_layers.append(Residual(num_channels))

    def call(self, X):
        for layer in self.residual_layers.layers:
            X = layer(X)
        return X
```

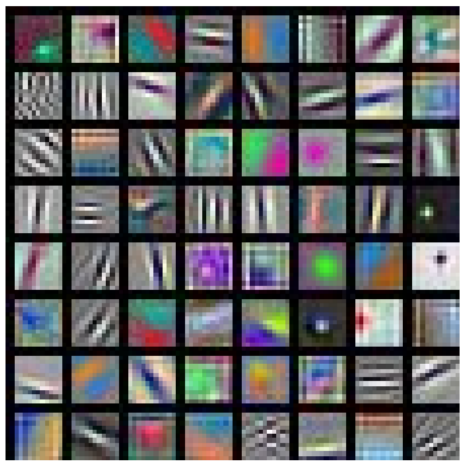
```
def net():
    return tf.keras.Sequential([
(1,224,224,3)    # The following layers are the same as b1 that we created earlier
(1,112,112,64) ⇐ tf.keras.layers.Conv2D(64, kernel_size=7, strides=2, padding='same'),
(1,112,112,64) ⇐ tf.keras.layers.BatchNormalization(),
(1,112,112,64) ⇐ tf.keras.layers.Activation('relu'),
(1,56,56,64) ⇐ tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same'),
        # The following layers are the same as b2, b3, b4, and b5 that we
        # created earlier
(1,56,56,64) ⇐ ResnetBlock(64, 2, first_block=True),
(1,28,28,128) ⇐ ResnetBlock(128, 2),
(1,14,14,256) ⇐ ResnetBlock(256, 2),
(1,7,7,512) ⇐ ResnetBlock(512, 2),
(1,512) ⇐ tf.keras.layers.GlobalAvgPool2D(),
(1,10) ⇐ tf.keras.layers.Dense(units=10)])
```


Q&A
WHO?
WHERE?
WHAT?
WHEN?
HOW?
Why?
When?
What?
Where?

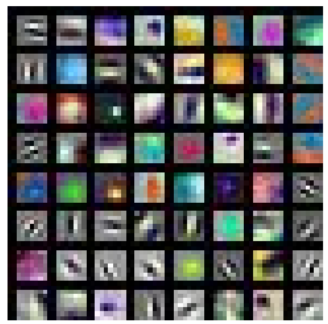
Visualizing CNNs

First layer visualization

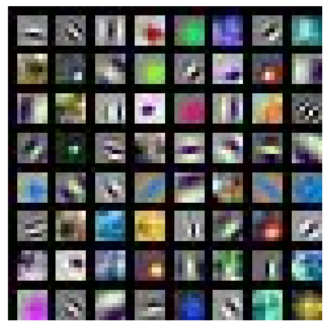
First layer visualization



AlexNet:
64 x 3 x 11 x 11



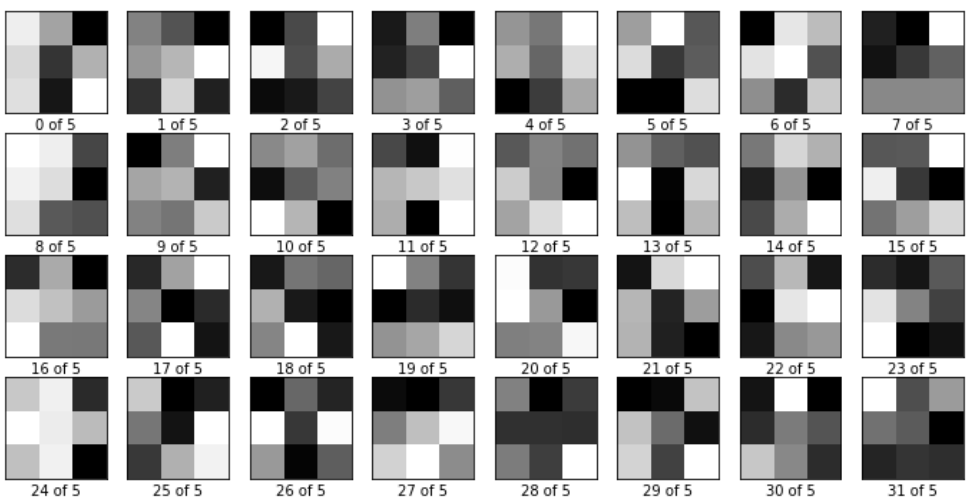
ResNet-18:
64 x 3 x 7 x 7



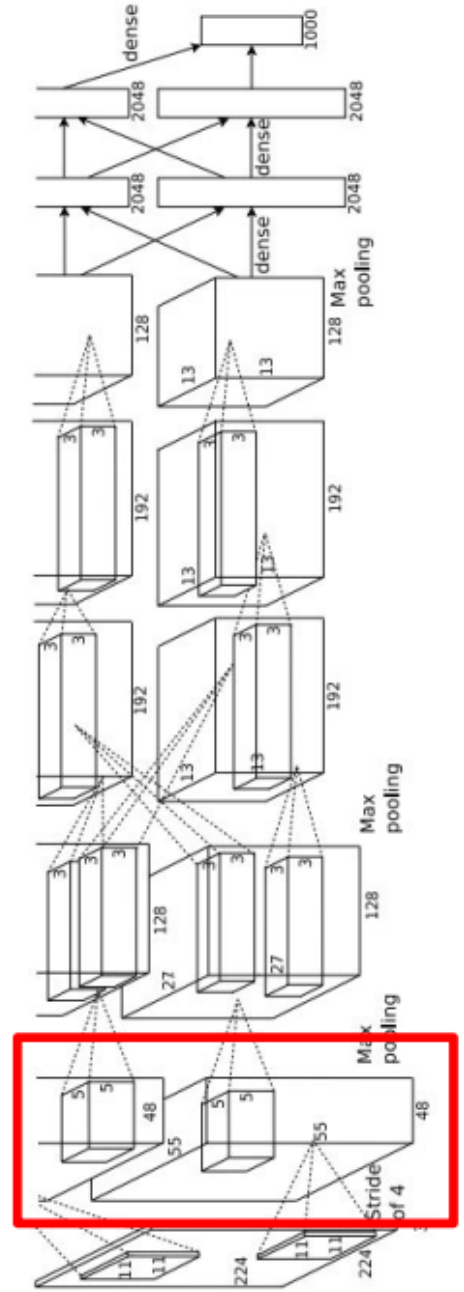
ResNet-101:
64 x 3 x 7 x 7

DenseNet-121:
64 x 3 x 7 x 7

Visualization of filters at higher layers is not that interesting



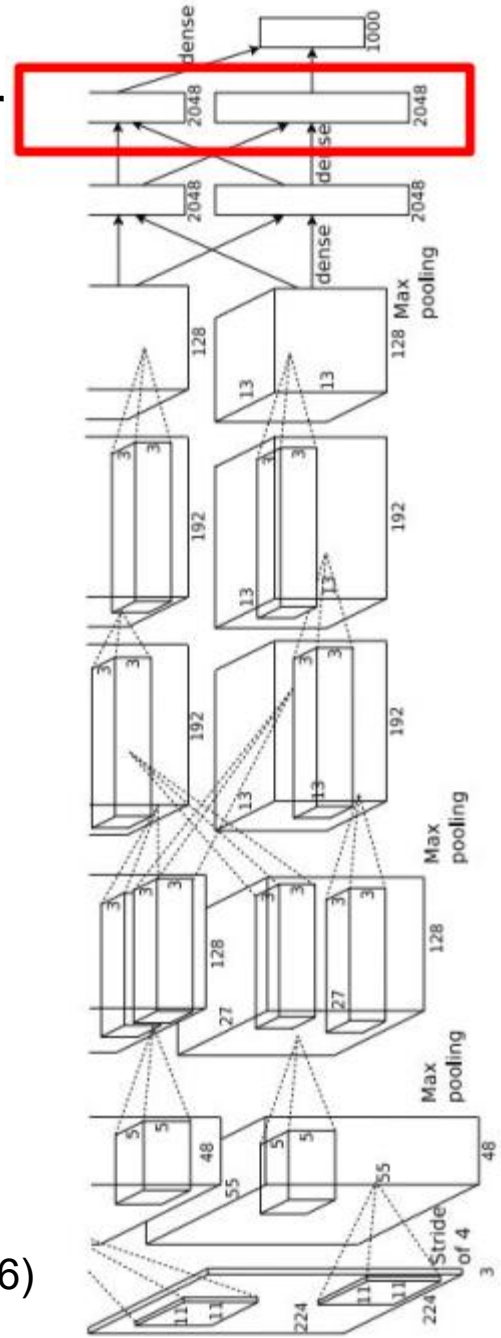
Tiny **3x3** filters



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

Last Layer: Nearest Neighbours

4096 dimensional vector

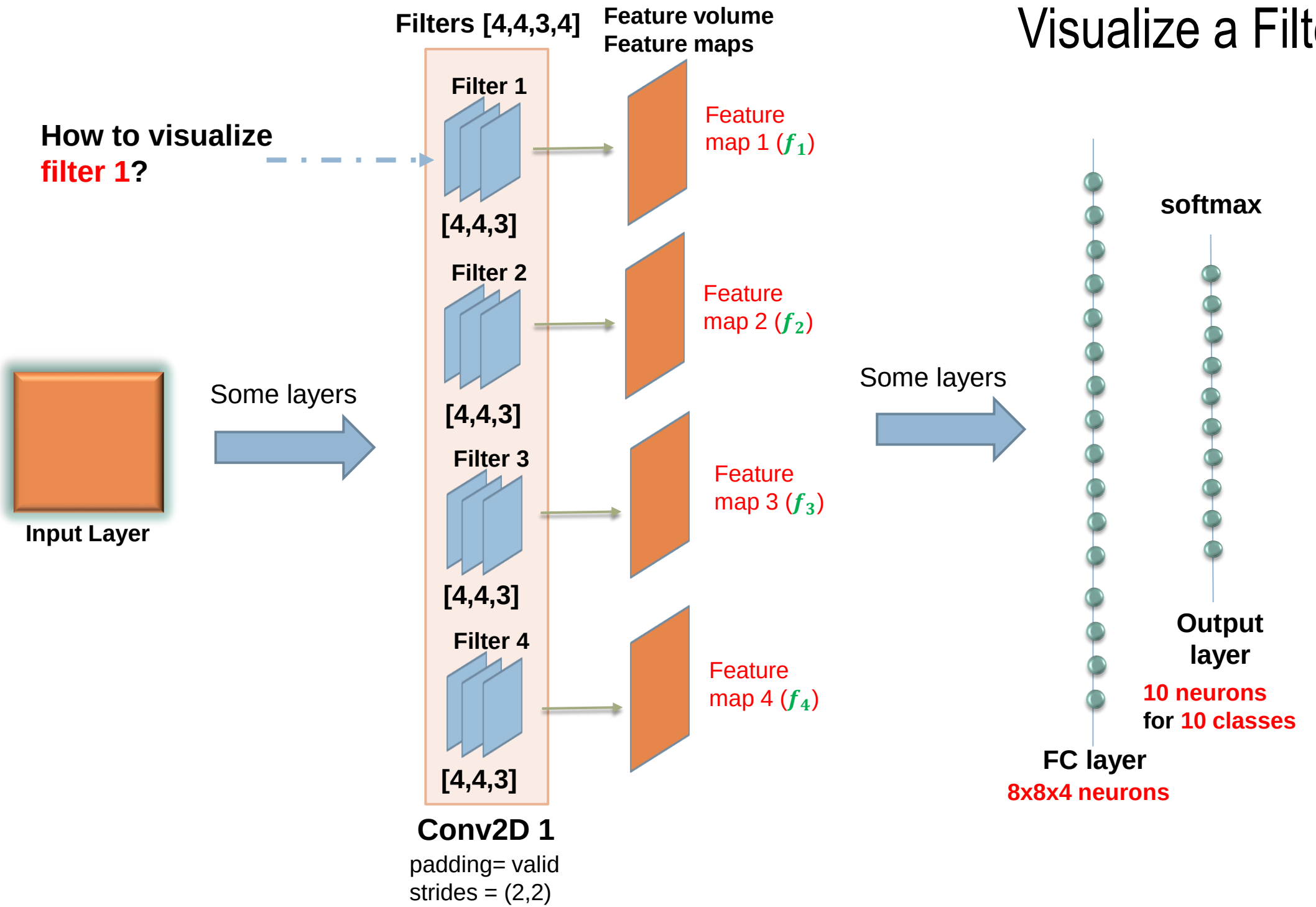


Nearest neighbours in pixel space

Test images **L2 Nearest neighbors in feature space**

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

Visualize a Filter



Visualize a Filter

- How to visualize **filter 1**?
- Find input I^* that **maximally fires filter 1**.
- $I^* = argmax_I\ mean(f_1)$

Input Layer

Some layers

Filters [4,4,3,4]

Feature volume
Feature maps

Filter 1

[4,4,3]

Feature map 1 (f_1)

Filter 2

[4,4,3]

Feature map 2 (f_2)

Filter 3

[4,4,3]

Feature map 3 (f_3)

Filter 4

[4,4,3]

Feature map 4 (f_4)

Conv2D 1

padding= valid
strides = (2,2)

Some layers

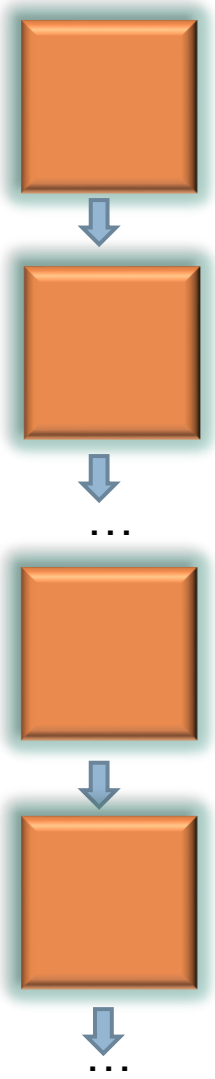
FC layer
8x8x4 neurons

softmax

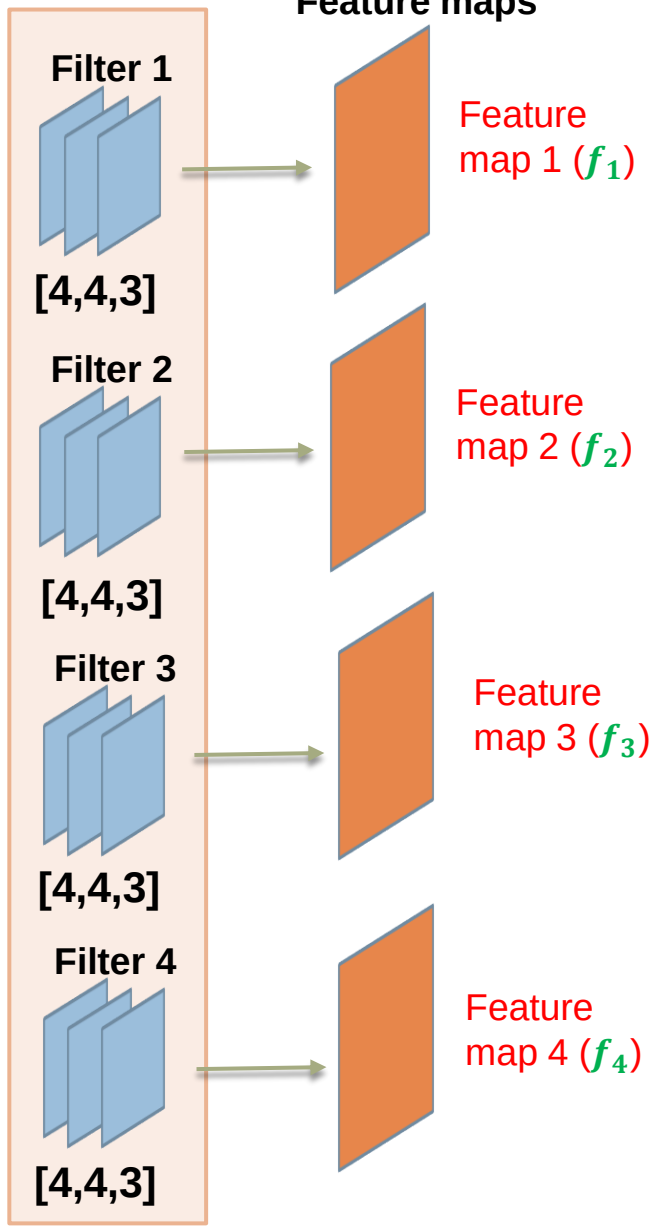
Output layer
10 neurons for 10 classes

Visualize a Filter

- How to visualize **filter 1**?
- Find input I^* that **maximally fires filter 1.**
- $I^* = argmax_I\ mean(f_1)$



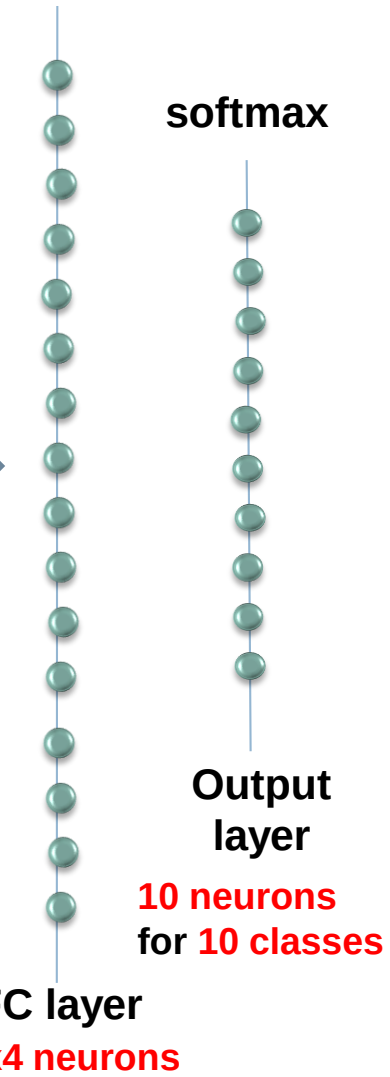
Some layers



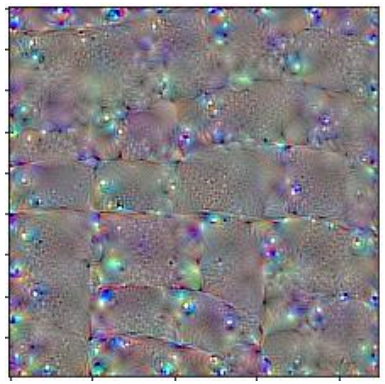
Conv2D 1

padding= valid
strides = (2,2)

Some layers



Activation map visualization



Input image **I**

Feed to our CNN

Filter 1

1	1	1
1	1	1
1	1	1

Feature map 1 (f_1)

-1	23	21
1	26	17
-1	8	13

Filter 2

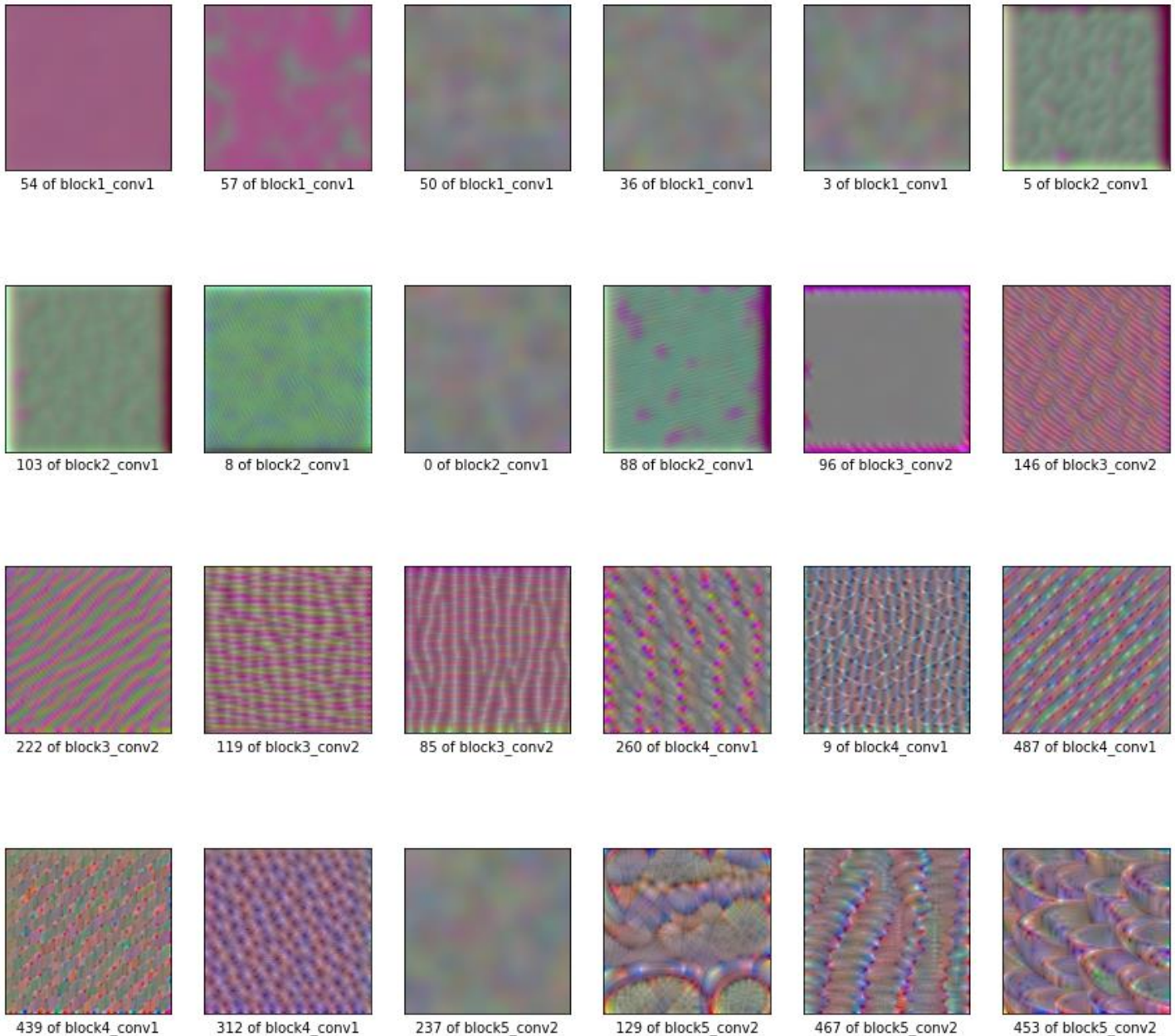
-1	-1	-1
-1	-1	-1
-1	-1	-1

Feature map 2 (f_2)

1	-23	-21
-1	-26	-17
1	-8	-13

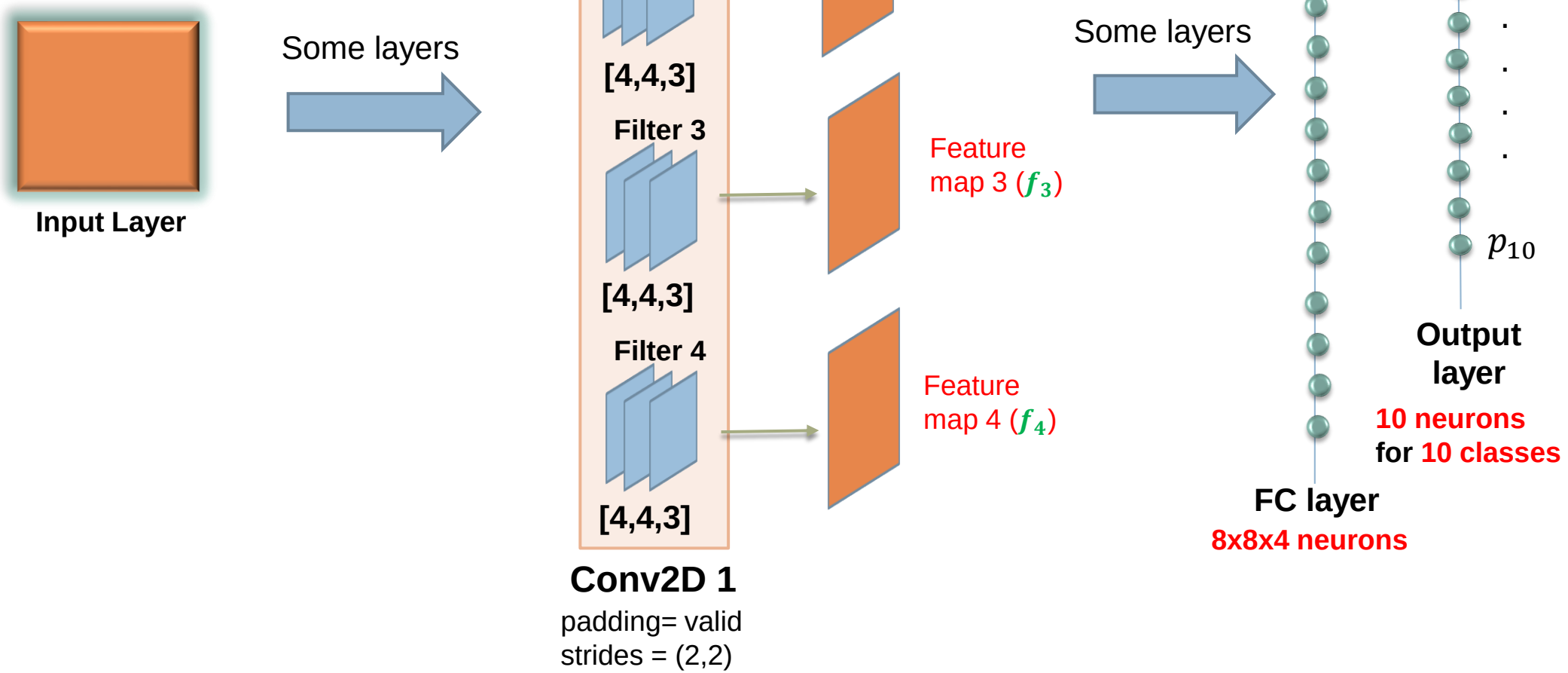
- Find **the input image I^*** that **maximally activate** a feature map **corresponding** to a filter
  - I^* shows the patterns in input images that can fire (maximally activate) a certain filter
- To visualize **filter 1,** we find the input I^* that **maximizes** average of feature map f_1
  - $I^* = argmax_I \mathcal{A}(I)$ where $\mathcal{A}(I) := mean(f_1) - \alpha\Omega(I)$
  - $\Omega(I)$ is a **norm** over I to encourage the sparsity
- Using **gradient ascent** for updating
  - $I_{t+1} = I_t + \eta \nabla_I \mathcal{A}(I_t)$
  - $\eta > 0$ is the learning rate

Activation map visualization of VGG19



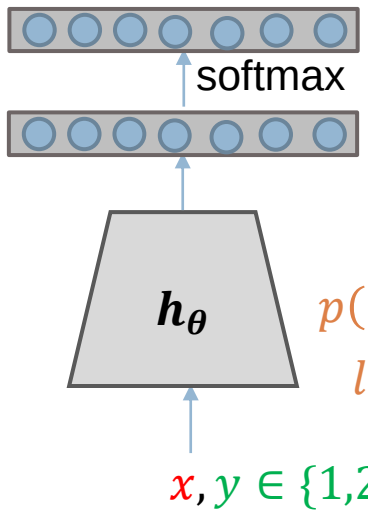
Visualize a Class

- How to visualize **class 1**?
- Find input I^* that **maximizes prediction probability p_1.**
- $I^* = argmax_I\ p_1$



Adversarial Machine Learning

Some notions



$f(x;\theta) = softmax(h(x;\theta))$ (prediction probabilities)

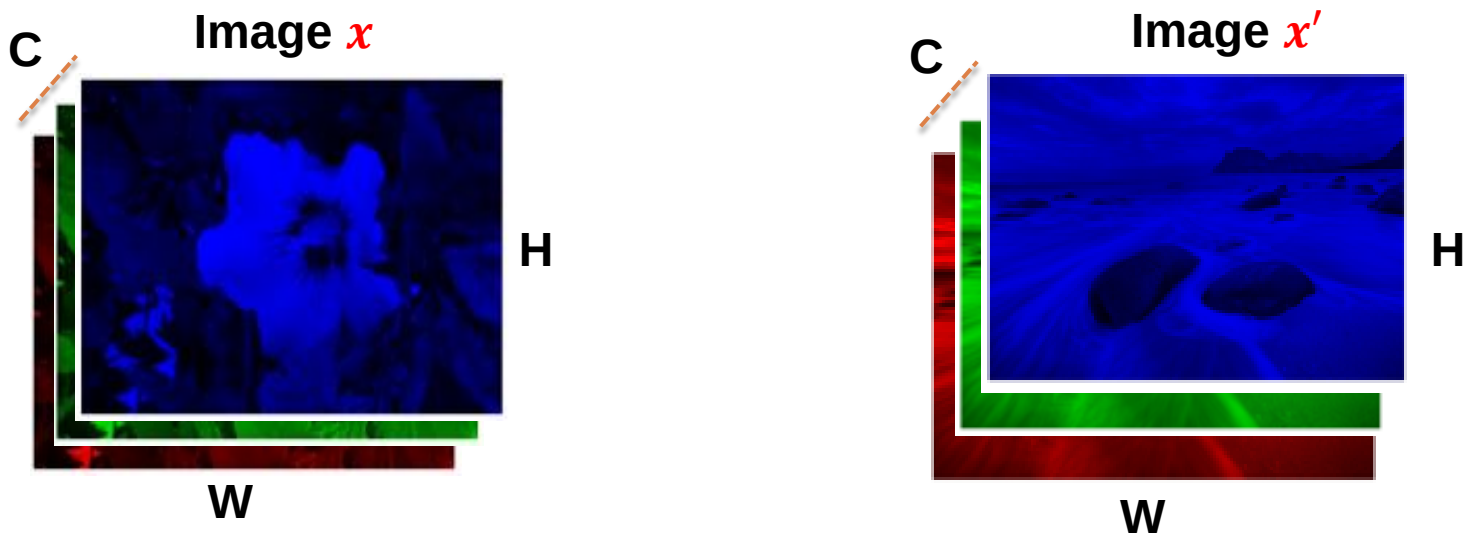
$h(x;\theta)$ (logits)

$$f(x;\theta) = [f_m(x;\theta)]_{m=1}^{M}$$

$$p(y = m \mid x) = f_m(x;\theta) \; for \; 1 \le m \le M$$

$$l(f(x;\theta), y) = CE\left(\mathbf{1}_y, f(x;\theta)\right) = -\log f_y(x;\theta)$$

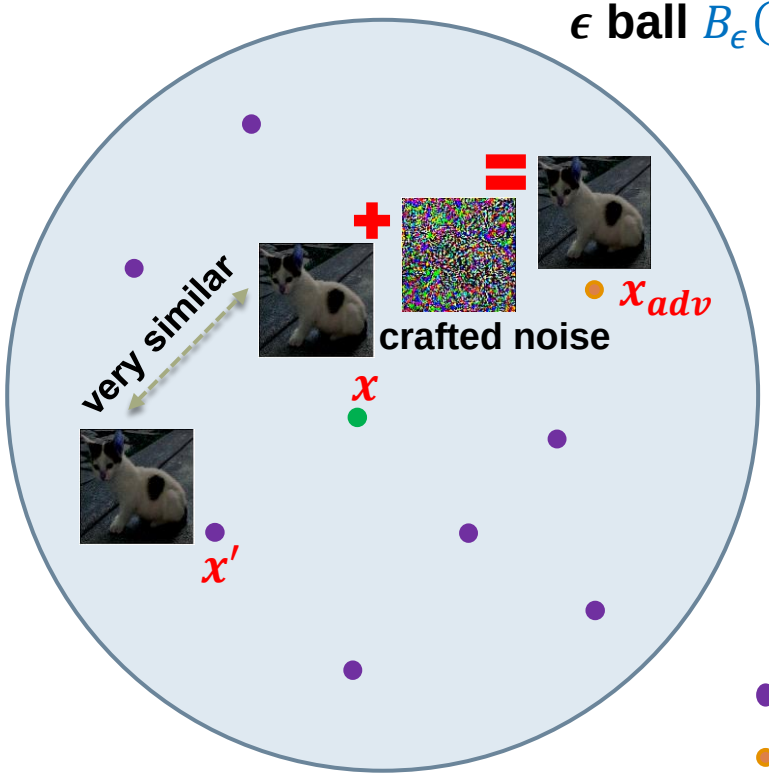
Distance between two images

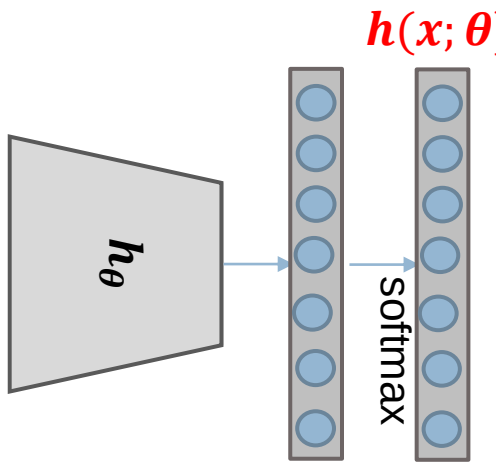


- x_{ijk} ($1 \leq i \leq H$, $1 \leq j \leq W$, and $1 \leq k \leq C$) is **the pixel** with coordinate $(height = i, width = j)$ on the channel (or slice) k of image x.
  - $0 \leq x_{ijk} \leq 255$ (original colour image) or $0 \leq x_{ijk} \leq 1$ (scale image)
- L_2 distance
  - $\|x - x'\|_2 = \sqrt{\sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{k=1}^{C} \left(x_{ijk} - x'_{ijk}\right)^2}$
- L_1 distance
  - $\|x - x'\|_1 = \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{k=1}^{C} |x_{ijk} - x'_{ijk}|$
- L_∞ distance
  - $\|x - x'\|_\infty = \max_{1 \leq i \leq H, 1 \leq j \leq W, 1 \leq k \leq C} |x_{ijk} - x'_{ijk}|$

Adversarial examples

ϵ ball $B_\epsilon(x) = \{x' : \|x' - x\| \leq \epsilon\}$ with a **small** $\epsilon > 0$.





$$f(x;\theta) = softmax(h(x;\theta))$$

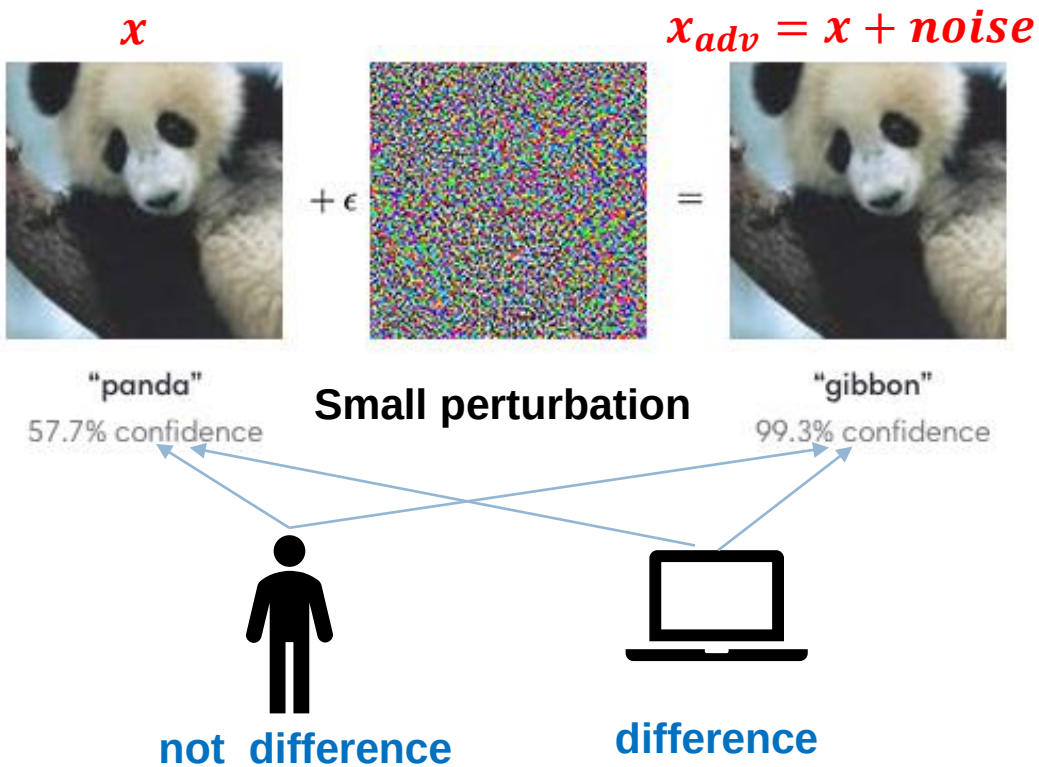
$$argmax_{1\leq i\leq M}\, f_i(x;\theta) = \hat{y} \neq \hat{y}_{adv} = argmax_{1\leq i\leq M}\, f_i(x_{adv};\theta)$$

- **Good** example which has **same predicted label** as x.
- **Adversarial** example which has **different predicted label** as x.

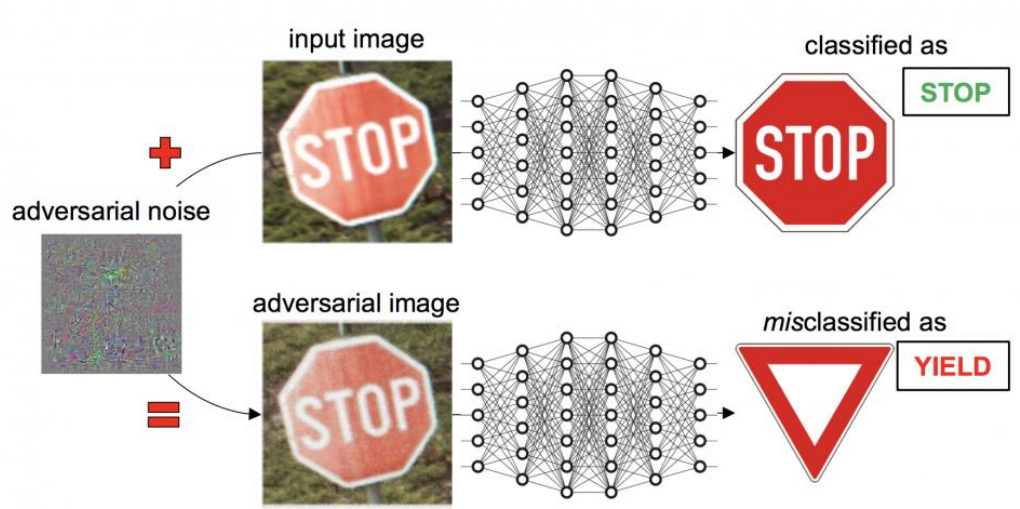
- The **true sense** of a **robust model**
  - If x' and x are **close**, the **predictions** of x' and x must be the **same**.
  - If $x' \in B_\epsilon(x)$ (i.e., $\|x' - x\| \leq \epsilon$), $argmax_{1\leq i\leq M}\, f_i(x;\theta) = \hat{y} = \hat{y}' = argmax_{1\leq i\leq M}\, f_i(x';\theta)$.
- Unfortunately, we can **easily** find **adversarial examples** $x_{adv} \in B_\epsilon(x)$ that can **fool** the classifier
  - $argmax_{1\leq i\leq M}\, f_i(x;\theta) = \hat{y} \neq \hat{y}_{adv} = argmax_{1\leq i\leq M}\, f_i(x_{adv};\theta)$

Adversarial examples

(Source: OpenAI)



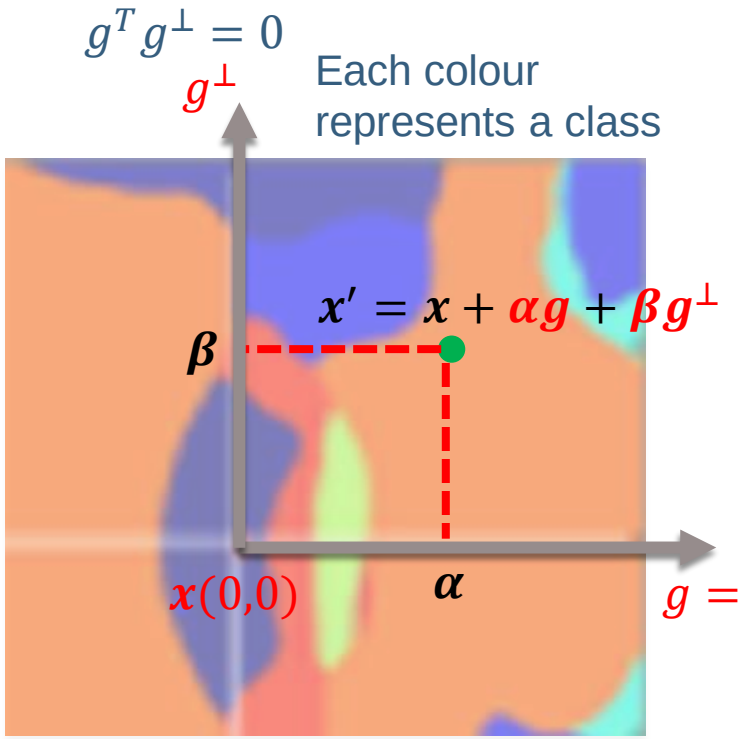
(Source: Internet)

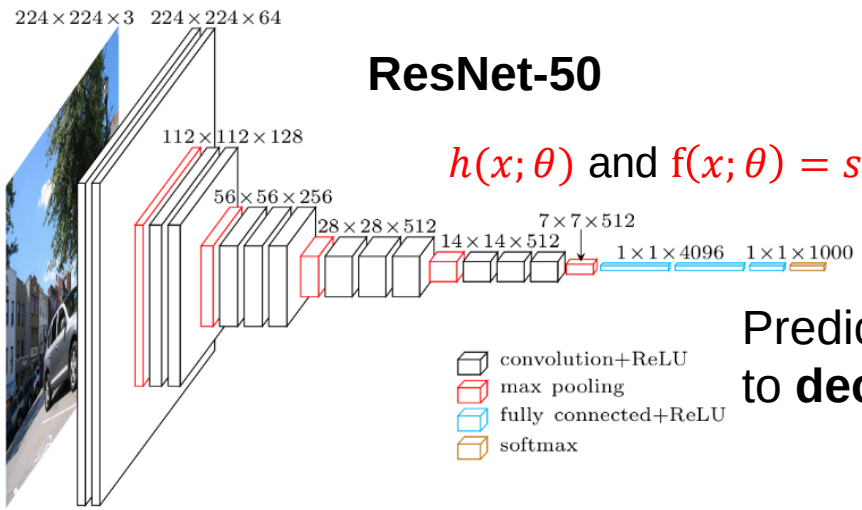


Misleading autonomous car driving system

- Adversarial example $\boldsymbol{x_{adv}}$ of a **clean data example** $\boldsymbol{x}$ w.r.t a model $f(.;\theta)$
  - $d(x_{adv}, x) = \|x_{adv} - x\| \leq \epsilon$ where $\|.\|$ is a norm (e.g., $\|.\|_1, \|.\|_2, \|.\|_\infty$)
  - f predicts x and x_{adv} with **different labels**, i.e., $argmax_i f_i(x;\theta) \neq argmax_i f_i(x_{adv};\theta)$

Visualization of decision regions





ResNet-50

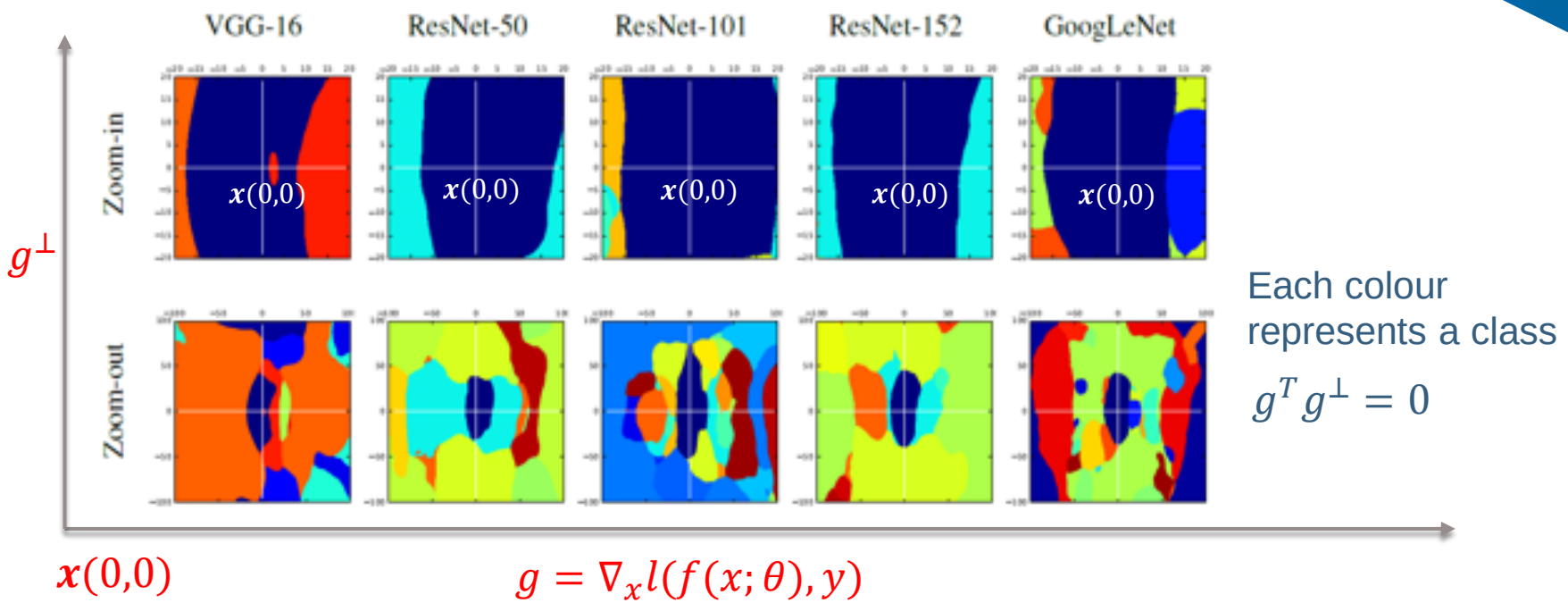
$h(x;\theta)$ and $f(x;\theta) = softmax(h(x;\theta))$

Predicts $x' = x + \alpha g + \beta g^\perp$ to **decide** the **colour**

Following the **gradient direction** is easy to **get out the blue region**.

- Start from $x = x + 0g + 0g^\perp$ (coordinate $(0,0)$), if we **go along with** the **gradient direction** $g = \nabla_x l(f(x;\theta), y)$, we can reach $x_{adv} = x + \alpha g + 0g^\perp$ such that
  - x_{adv} is very close to x → human cannot tell how they **are different** → the **same class** from human **perceptron**
  - x_{adv} is predicted to **different class** from x by the classifier
  - x_{adv} is called **adversarial example** of x

Decision regions of deep learning model



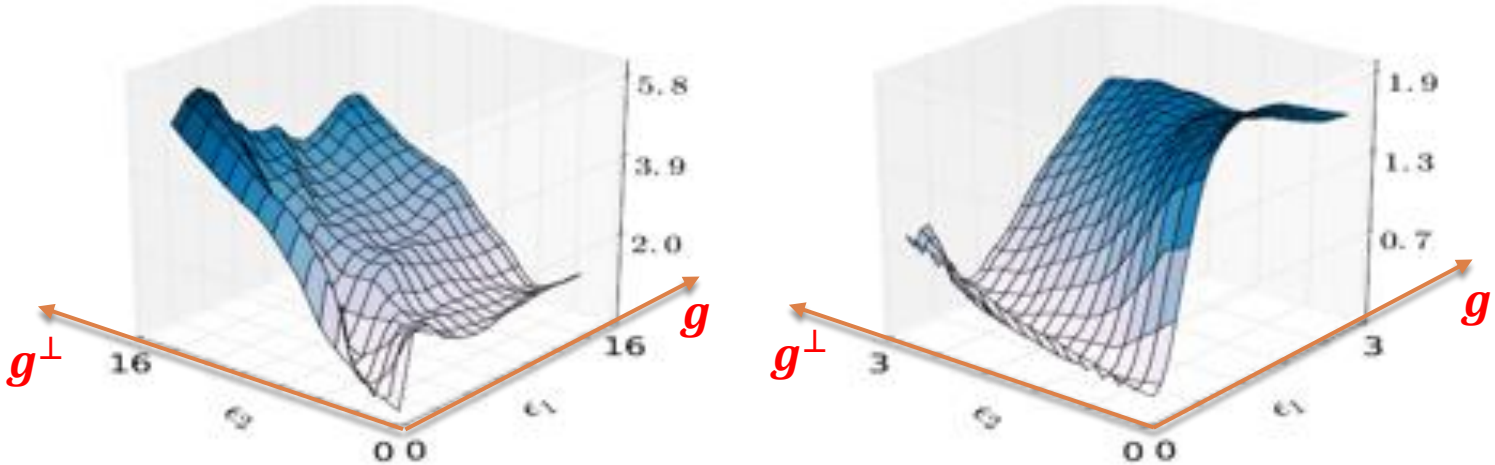
Each colour represents a class

$g^T g^{\perp} = 0$

- Deep learning models are **fragile** and **easy to be attacked**
  - Start from $x = x + 0g + 0g^{\perp}$ (coordinate $(\mathbf{0}, \mathbf{0})$), if we go along with the gradient direction $g = \nabla_x l(f(x;\theta), y)$, we can reach $x_{adv} = x + \alpha g + 0g^{\perp}$ such that
    - x_{adv} is very close to x → human cannot tell how they **are different** → the **same class** from human **perceptron**
    - x_{adv} is predicted to **different class** from x by the classifier
    - x_{adv} is called **adversarial example** of x

Delving into Transferable Adversarial Examples and Black-box Attacks
Yanpei L., et al., ICLR 2016

Loss surface of deep learning model



Zoom out **Zoom in**

The **loss surface** of DL model **around** $\boldsymbol{x}$. This is plotted on $\boldsymbol{x' = x + \epsilon_1 g + \epsilon_2 g^{\perp}}$.

- At the **data point** $\boldsymbol{x}$, if we move along with the **gradient direction** g, the **loss function** is **maximally locally increased**.
- **Taylor expansion** around x
  - $l(f(x+h;\theta),y) \approx l(f(x;\theta),y) + g^T h$ where $g = \nabla_x l(f(x;\theta),y)$.
  - The gradient direction is the **steepest direction** to increase the loss function locally.

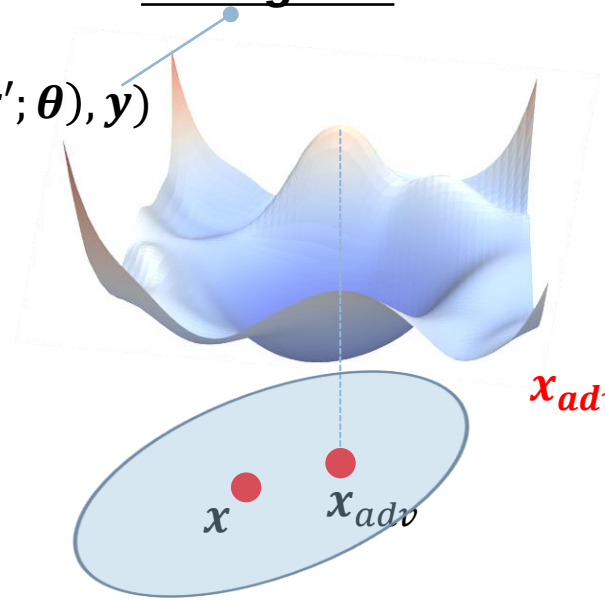
Ensemble Adversarial Training Attacks and Defences
Tramer et al., ICLR 2017

Adversarial attack

How to run untargeted attack?

***Untargeted* attacks**

Loss surface $\boldsymbol{l(f(x';\theta),y)}$



$B_\epsilon(x) = \{x' : \|x' - x\| \leq \epsilon\}$

$\boldsymbol{x}$ has true label $\boldsymbol{y} \in \{\boldsymbol{1}, \boldsymbol{2}, \ldots, \boldsymbol{M}\}$

$$\boldsymbol{x_{adv} = argmax_{x' \in B_\epsilon(x)} l(f(x';\theta),y)}$$

- **Fast Gradient Sign Method** (FGSM)
  - $x_{adv} = x + \epsilon sign(\nabla_x l(f(x;\theta),y))$
  - $sign(t) = \begin{cases} 1 & if\ t > 0 \\ -1 & if\ t < 0 \\ 0 & otherwise \end{cases}$
  - One-step update

Explaining and Harnessing Adversarial Examples
Goodfellow et al., ICLR 2015

- **Projected Gradient Descent (Ascent)** (PGD)
  - $x_0 = x + Uniform([-\epsilon, \epsilon])$
  - $\tilde{x}_{t+1} = x_t + \eta \nabla_x l(f(x_t;\theta),y)$
  - $x_{t+1} = Proj_{B_\epsilon(x)}(\tilde{x}_{t+1})$
  - Run in k steps ($k = 20$), $\eta > 0$ is the learning rate
  - $x_{adv} = x_k$

Towards Deep Learning Models Resistant to Adversarial Attacks
Madry et al., ICLR 2017

- **Momentum Iterative Method** (MIM)
  - $x_0 = x + Uniform([-\epsilon, \epsilon])$
  - $v_{t+1} = \gamma v_t + \frac{\nabla_x l(f(x_t;\theta),y)}{\|\nabla_x l(f(x_t;\theta),y)\|_1}$
  - $x_{t+1} = Proj_{B_\epsilon(x)}(x_t + \eta v_{t+1})$
  - Run in k steps ($k = 20$), $\gamma > 0$ is momentum decay parameter
  - $x_{adv} = x_k$

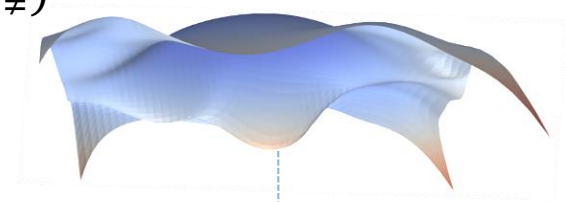
Boosting Adversarial Attacks with Momentum
Dong et al., CVPR 2018

Adversarial attack

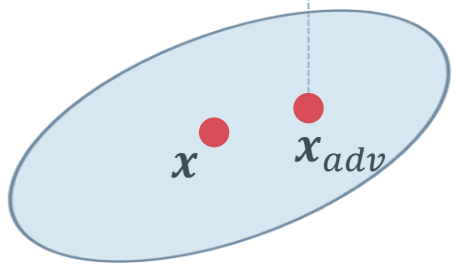
How to run targeted attack?

***Targeted* attacks**

Loss surface $\boldsymbol{l(f(x';\theta), y_{\neq})}$
$\boldsymbol{y_{\neq}}$ is different from $\mathbf{y}$



$B_\epsilon(x) = \{x' : \|x' - x\| \leq \epsilon\}$
$\boldsymbol{x}$ has true label $\boldsymbol{y} \in \{\mathbf{1}, \mathbf{2}, \ldots, \boldsymbol{M}\}$



$$\boldsymbol{x_{adv} = argmin_{x' \in B_\epsilon(x)} l(f(x';\theta), y_{\neq})}$$

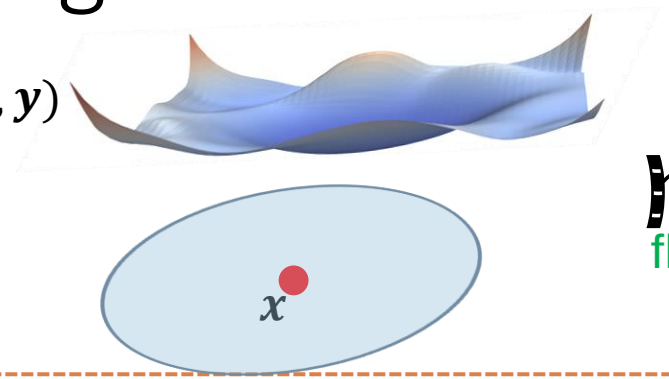
- **Fast Gradient Sign Method** (FGSM)
  - $x_{adv} = x - \epsilon sign(\nabla_x l(f(x;\theta), y))$
  - $sign(t) = \begin{cases} 1 & if\ t > 0 \\ -1 & if\ t < 0 \\ 0 & otherwise \end{cases}$
  - One-step update

- **Projected Gradient Descent (Ascent)** (PGD)
  - $x_0 = x + Uniform([-\epsilon, \epsilon])$
  - $\tilde{x}_{t+1} = x_t - \eta \nabla_x l(f(x_t;\theta), y)$
  - $x_{t+1} = Proj_{B_\epsilon(x)}(\tilde{x}_{t+1})$
  - Run in k steps ($k = 20$), $\eta > 0$ is the learning rate
  - $x_{adv} = x_k$

- **Momentum Iterative Method** (MIM)
  - $x_0 = x + Uniform([-\epsilon, \epsilon])$
  - $v_{t+1} = \gamma v_t + \frac{\nabla_x l(f(x_t;\theta), y)}{\|\nabla_x l(f(x_t;\theta), y)\|_1}$
  - $x_{t+1} = Proj_{B_\epsilon(x)}(x_t - \eta v_{t+1})$
  - Run in k steps ($k = 20$), $\gamma > 0$ is momentum decay parameter
  - $x_{adv} = x_k$

Adversarial training

Loss surface $l(f(x';\theta),y)$



We wish a **smooth** (or flat) **loss surface** around x

- The **ideal optimization problem** for adversarial training
  - $\min_{\theta} \mathbb{E}_{(x,y)\sim p(x,y)} \left[\max_{x' \in B_\epsilon(x)} l(f(x';\theta),y) \right]$
  - $p(x,y)$ is the **data distribution** that generates the pair of data point x and label y
  - Minimize the loss of over **most violated** adversarial examples

- **PGD adversarial training**
  - for epoch in n_epochs
    - for iter in range(n_iter_per_epoch)
      - Sample mini-batch $(x_1,y_1),\dots,(x_b,y_b)$ from the training set
      - Find PGD untargeted adversarial examples $x_1^{adv},\dots,x_b^{adv}$ for $x_1,\dots,x_b$ w.r.t labels $y_1,\dots,y_b$
      - $batch\_loss = \frac{1}{b}\sum_{i=1}^{b} l(f(x_i;\theta),y_i) + \frac{1}{b}\sum_{i=1}^{b} l(f(x_i^{adv};\theta),y_i)$
      - $\theta = \theta - \eta \frac{\partial batch\_loss}{\partial \theta}$

Adversarial training in practice

Principle to develop machine learning models

- Training condition **must match** testing condition

In reality, given $x_1, \dots, x_b$ and a model f_θ, how to **attack**?

↓

Use f_θ to predict $x_1, \dots, x_b$ → **predicted labels** $y_1^{pred}, \dots, y_b^{pred}$

↓

Attack f_θ using $y_1^{pred}, \dots, y_b^{pred}$ → **untargeted adversarial examples** $x_1^{adv}, \dots, x_b^{adv}$

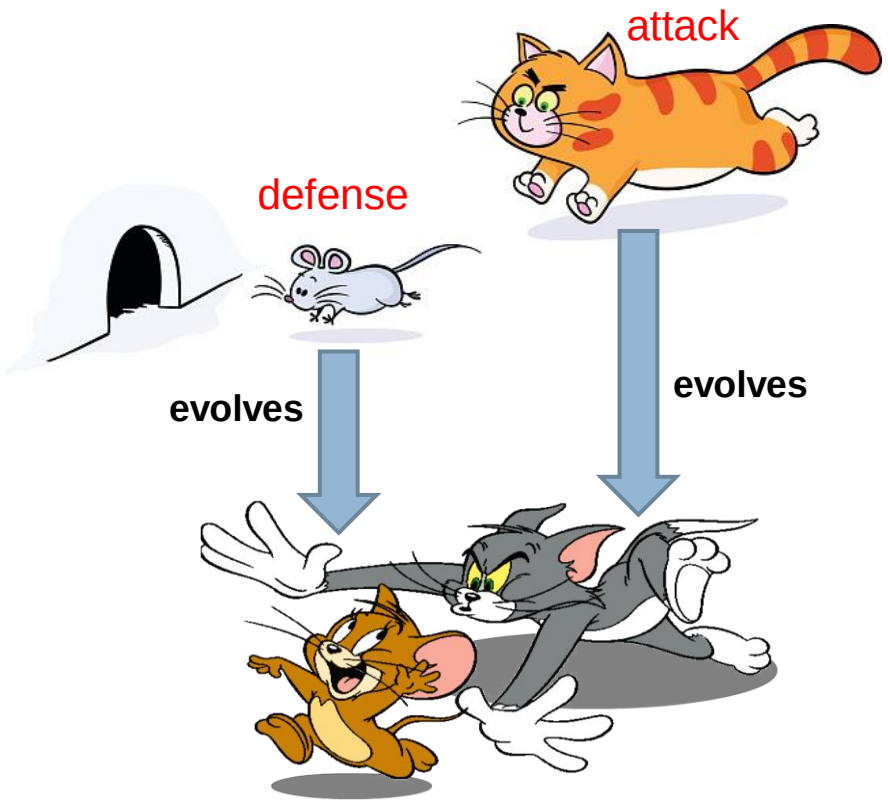
↓

Need to ensure f_θ to **secure** to these $x_1^{adv}, \dots, x_b^{adv}$?

PGD adversarial training in practice

- for epoch in n_epochs
  - for iter in range(n_iter_per_epoch)
    - Sample mini-batch $(x_1, y_1), \dots, (x_b, y_b)$ from the training set
    - Evaluate the **predicted** labels $y_1^{pred}, \dots, y_b^{pred}$ by the **current model**, i.e., $y_1^{pred} = argmax_m f_m(x_1; \theta)$, and etc.
    - Find PGD **untargeted** adversarial examples $x_1^{adv}, \dots, x_b^{adv}$ for $x_1, \dots, x_b$ w.r.t labels $y_1^{pred}, \dots, y_b^{pred}$
    - $batch\_loss = \frac{1}{b}\sum_{i=1}^{b} l(f(x_i; \theta), y_i) + \frac{1}{b}\sum_{i=1}^{b} l(f(x_i^{adv}; \theta), y_i)$
    - $\theta = \theta - \eta \frac{\partial batch\_loss}{\partial \theta}$

Attack vs Defense: Cat-and-mouse game



(Source: Internet)

- **Attack** is **stronger** than **defense**
- No defense method can **really defend** on ImageNet dataset.

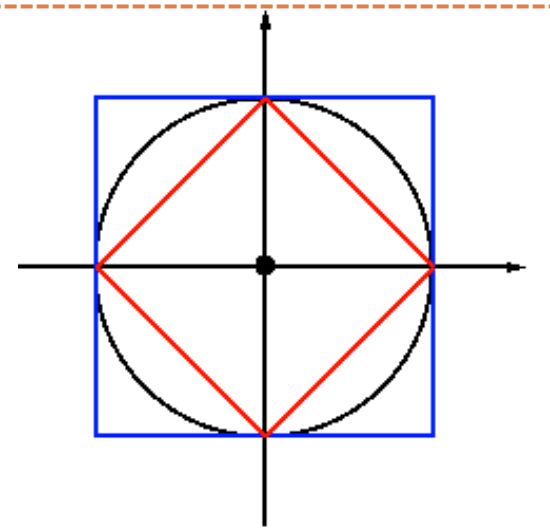
Summary

- Deep learning before and after 2012
- Revision of building-blocks of CNN
  - Conv layer, pooling layer, global pooling layer
  - Drawback of CNNs
- SOTA Convolutional Neural Networks
  - GoogLeNet and ResNet
- Visualization in CNNs
  - Activation map, visualizing filters, last layer nearest neighbour
- Adversarial Machine Learning
  - Adversarial examples, attack and defense

Thanks for your attention!

Revision of norm

- Given **two vectors $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^d$**, the **distance between two vectors** is defined as **a norm (length)** of $x - x'$
  - $\|x - x'\|_1 = \sum_{i=1}^{d} |x_i - x_i'|$
  - $\|x - x'\|_2 = \sqrt{\sum_{i=1}^{d} (x_i - x_i')^2}$
  - $\|x - x'\|_\infty = \max_{1 \le i \le d} |x_i - x_i'|$



Unit sphere with different norms

$\|x\|_\infty = 1$

$\|x\|_1 = 1$

$\|x\|_2 = 1$

Implementation of Attacks

```python
def fgsm_attack(model, input_image, input_label,
                epsilon=0.3,
                clip_value_min=0.,
                clip_value_max=1.0,
                soft_label=False):
    """
    Args:
        model: pretrained model
        input_image: original (clean) input image (tensor)
        input_label: original label (tensor, categorical representation)
        epsilon: perturbation boundary
        clip_value_min, clip_value_max: range of valid input
    Note:
        we expect the output of model should be logits vector
    """

    loss_fn = tf.nn.sparse_softmax_cross_entropy_with_logits

    if type(input_image) is np.ndarray:
        input_image = tf.convert_to_tensor(input_image)

    if type(input_label) is np.ndarray:
        input_label = tf.convert_to_tensor(input_label)

    with tf.GradientTape() as tape:
        tape.watch(input_image)
        prediction = model(input_image)
        if not soft_label:
            loss = loss_fn(input_label, prediction)
        else:
            pred_label = tf.math.argmax(prediction, axis=1)
            loss = loss_fn(pred_label, prediction)

    # Get the gradients of the loss w.r.t the input image
    gradient = tape.gradient(loss, input_image)

    # Get the final adversarial examples
    adv_image = input_image + epsilon * tf.sign(gradient)

    # Clip to a valid range
    adv_image = tf.clip_by_value(adv_image, clip_value_min, clip_value_max)

    # Stop the gradient to make the adversarial image as a constant input
    adv_image = tf.stop_gradient(adv_image)

    return adv_image
```

FGSM attack

```python
def pgd_attack(model, input_image, input_label,
               epsilon=0.3,
               num_steps=20,
               step_size=0.01,
               clip_value_min=0.,
               clip_value_max=1.0,
               soft_label=False):
    """
    Args:
        model: pretrained model
        input_image: original (clean) input image (tensor)
        input_label: original label (tensor, categorical representation)
        epsilon: perturbation boundary
        num_steps: number of attack steps
        step_size: size of each move in each attack step
        clip_value_min, clip_value_max: range of valid input
    Note:
        we expect the output of model should be logits vector
    """

    loss_fn = tf.nn.sparse_softmax_cross_entropy_with_logits

    if type(input_image) is np.ndarray:
        input_image = tf.convert_to_tensor(input_image)

    if type(input_label) is np.ndarray:
        input_label = tf.convert_to_tensor(input_label)

    # random initialization around input_image
    random_noise = tf.random.uniform(shape=input_image.shape, minval=-epsilon, maxval=epsilon)
    adv_image = input_image + random_noise

    for _ in range(num_steps):

        with tf.GradientTape(watch_accessed_variables=False) as tape:
            tape.watch(adv_image)
            prediction = model(adv_image)
            if not soft_label:
                loss = loss_fn(input_label, prediction)
            else:
                pred_label = tf.math.argmax(prediction, axis=1)
                loss = loss_fn(pred_label, prediction)

        # Get the gradient of the loss w.r.t the current point
        gradient = tape.gradient(loss, adv_image)

        # Move current adverarial example along the gradient direction with step size is eta
        adv_image = adv_image + step_size * tf.sign(gradient)

        # Clip to a valid boundary
        adv_image = tf.clip_by_value(adv_image, input_image-epsilon, input_image+epsilon)

        # Clip to a valid range
        adv_image = tf.clip_by_value(adv_image, clip_value_min, clip_value_max)

        # Stop the gradient to make the adversarial image as a constant input
        adv_image = tf.stop_gradient(adv_image)

    return adv_image
```

PGD attack