

# Face Processing & Face Detection

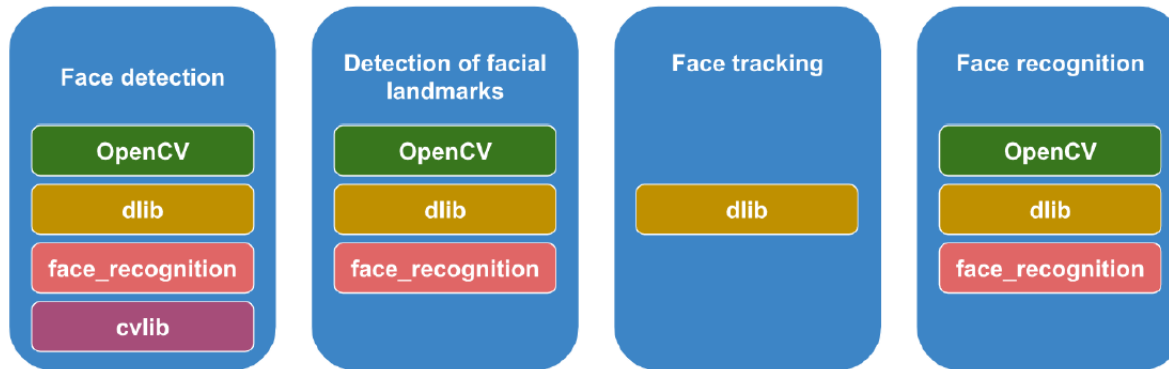
using OpenCV, dlib, face\_recognition,  
cvlib modules

2021년 2학기

서경대학교 김진헌

# 얼굴 영상 처리 기술의 장르

1

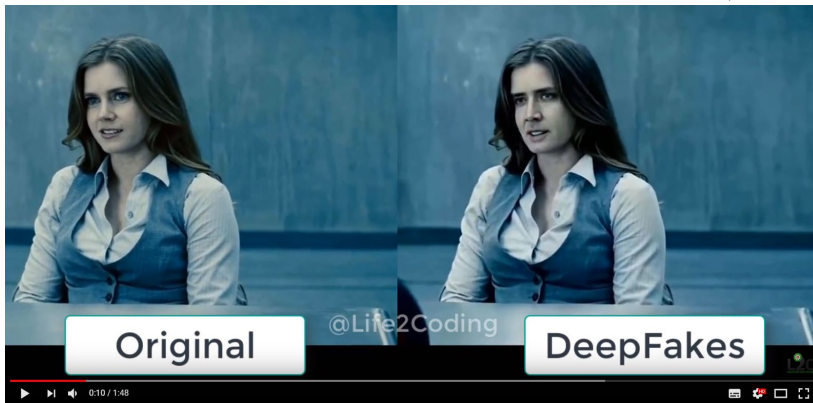


- Face detection
  - ▣ 얼굴의 위치와 크기 검출
- Detection of facial landmark
  - ▣ 얼굴의 주요 부위 위치/크기 검출: 눈, 입, 코, 뺨 등..
- Face tracking
  - ▣ 움직이는 얼굴의 위치와 크기 검출
- Face recognition
  - ▣ Face identification(1: N): 등록된 얼굴 중에 누구인지를 맞추는 처리=>응용사례: Access Control
  - ▣ Face verification(1:1): 자신이라고 주장하는 사람이 맞는지 맞추는 처리 => ATM

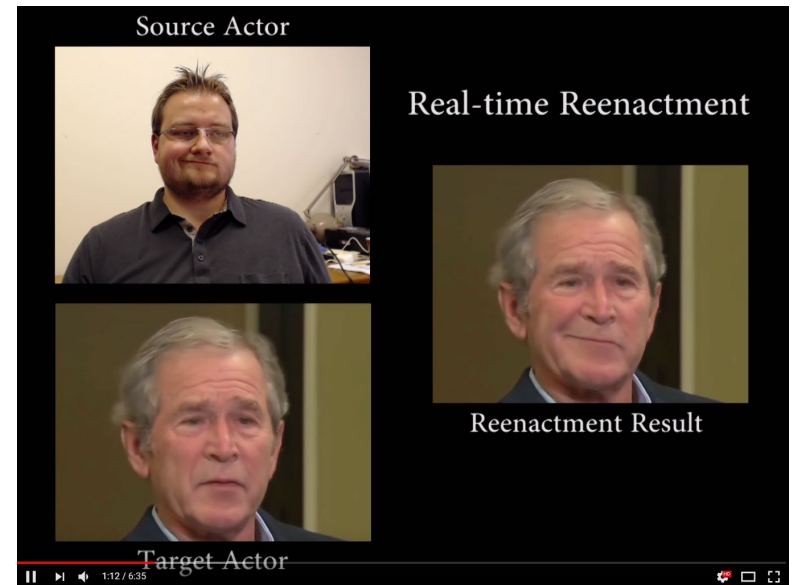
# 참고: 얼굴 영상 처리 기술의 최신동향

2

DeepFakes Video Collections



Face2Face



(a) Original (b) Blurring (c) Pixelation (d) Ours




AnonymousNet(2019)  
Face De-identification

# 잠깐,,, 얼굴인식 콘테스트 -- 미완성

3

- FRVT(Face Recognition Vendor Test), 미국 국가기술표준원(NIST)
  - ▣ 2018 중국이 1~5위 석권.
    - 상하이의 스타트업 이투커지(이투 테크놀로지)가 2016년 이루 3년 연속 1위,
  - ▣ 2021년
    - 국내 기업(알체라) 출국심사대 촬영부문- 세계 12위. 국내 1위, 330개 프로그램 참여
- 중국은 5개의 스타트업 주도 IBM, AWS, MS 등은 사업 포기 또는 중단

# □ Face Detection

 1\_face\_detection\_opencv\_dnn.py

 2\_face\_detection\_opencv\_haar.py

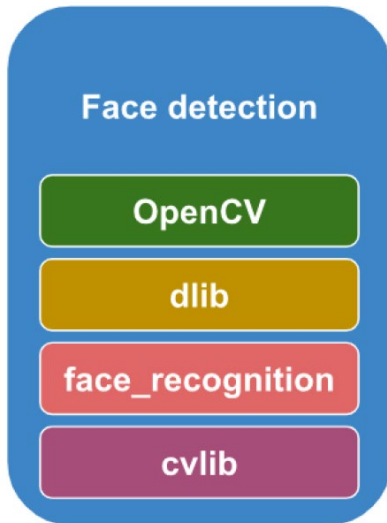
 3\_face\_detection\_dlib\_hog.py

 4\_face\_detection\_dlib\_cnn.py

# 얼굴 검출에 활용 모듈들..

학습 대상은  
푸른색으로 표기

5



## □ OpenCV

### ▣ Haar cascade classifier

- `object = cv2.CascadeClassifier("모델.xml")`
- `얼굴_사각형_리스트=Object.detectMultiScale(gray)`
- `얼굴_사각형_ndarray`  
`=cv2.face.getFacedHAAR(color, "모델.xml")`

### ▣ Deep learning based classifier

- SSD(Single Shot Multi Box): VGG16기반
- OpenCV DNN(Deep Neural Network) face detector는 ResNet-10 Network를 사용하는 SSD(Single Shot Multibox Detector, [개념 소개 링크](#)) 구조를 기반으로 설계되었다.
- [논문: SSD: Single Shot MultiBox Detector](#) (by C. Szegedy et al. 2016)

VGG 16

## □ Dlib

- ▣ HOG(Histogram of Gradients) 기반: 0.25초
- ▣ CNN 기반: 5초. GPU 버전으로 컴파일 필요.

## □ face\_recognition

- ▣ 내부적으로 dlib의 HOG, CNN detector 사용
- ▣ 실험에 의하면 *pyCharm의 Setting에서는 설치할 수 없었다.*
  - *pip로 설치 가능: pip install face\_recognition*
    - *먼저 cmake가 설치되어 있어야 한다.*
    - *내 경우는 pip 명령으로 설치하였음, 어쩌면 visual studio가 필요할 지 모른다 (확실히 모름).*
  - 수동으로 설치하는 방법(C++ 컴파일러가 있어야 한다):
- ▣ Labeled Faces in the Wild benchmark 상으로 99.38% 정확도를 가진다.
- ▣ 함수 뿐만 아니라, 커맨드 도구로도 지원한다.

## □ cvlib

- ▣ 내부적으로는 OpenCV DNN을 사용

# SSD: 개요

OpenCV Deep learning based classifier가 사용

7

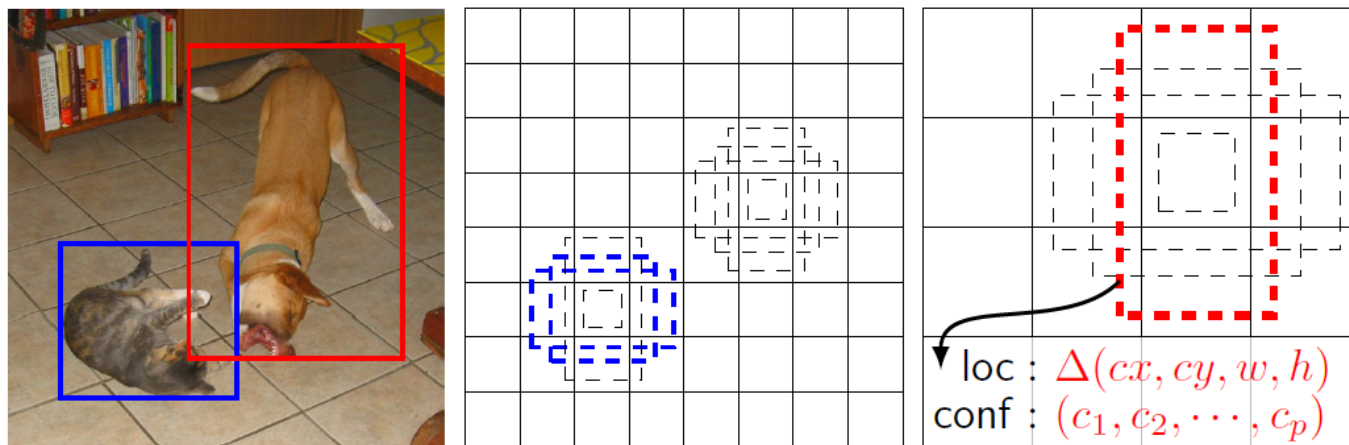
- Region-Convolutional Neural Network (R-CNN)의 문제점을 해결하고자 제안
  - R-CNN 문제점: 데이터가 너무 커서 다루기 힘들다. 학습이 다단계(예: training region proposal vs classifier)로 이루어 진다. Inference 단계에서 시간이 많이 걸린다.
  - 이를 개선하고자 나온 것이 YOLO(You Only Look Once)와 SSD multibox이다.
- SSD의 성능
  - 74% *mAP (mean Average Precision)* at 59 frames per second on standard datasets such as PascalVOC and COCO
    - PascalVOC: 20 classes. The train/val data has 11,530 images containing 27,450 ROI annotated objects and 6,929 segmentations.
    - COCO: 330K images (>200K labeled), 1.5 million object instances, 80 object categories, 91 stuff categories, 5 captions per image, 250,000 people with keypoints
- SSD는 전에 나온 YOLO 보다 빠르고 정확하다.
- Faster R-CNN을 포함해서 이들과 같은 explicit region proposals과 pooling을 수행하는 더 처리 시간이 오래 걸리는 다른 기술만큼 정확하다.



# SSD: 개요

8

- **Single Shot:** this means that the tasks of object localization and classification are done in a *single forward pass* of the network
- **MultiBox:** this is the name of a technique for bounding box regression developed by Szegedy et al.
- **Detector:** The network is an object detector that also classifies those detected objects



(a) Image with GT boxes    (b)  $8 \times 8$  feature map    (c)  $4 \times 4$  feature map

영상 입력은 300x300으로 정한다.

(a) 2개의 영역에 대해 고양이와 강아지라는 라벨이 붙어 있다.

(b), (c) 각 영역에 4개의 기본 박스를 8x8과 4x4 영역에 대해 적용하기를...

모든 카테고리에 대해 위치(offset)와 확실성(confidence)을 예측한다. 학습할 때는 ground truth 정보를 바탕으로 맞는 블록에 대해서는 positive, 나머지 영역에 대해서는 negative 값을 준다. 모델의 loss는 위치와 확실성의 가중 평균으로 구한다.

## □ Accuracy      일반적인 정확도의 정의

$$\text{Precision} = \frac{TP}{TP + FP}$$

*TP = True Positives (Predicted as positive as was correct)*

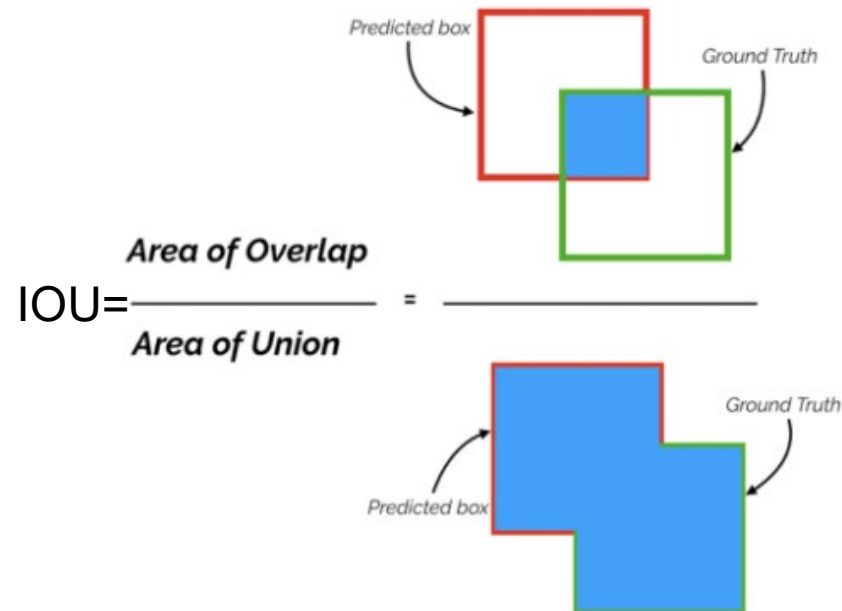
*FP = False Positives (Predicted as positive but was incorrect)*

단순 Accuracy는 영상에서 어느 곳이 무엇이 있는지를 판단해야 하는 객체검출(object detection) 작업에는 적절하지 않다.

□ 객체 검출에는 mAP(mean average precision)를 사용하여 검출 성능을 계량화 한다.

□ 이 지표는 IOU(intersection over union)와 올바른 객체의 분류 정확도를 함께 고려한다.

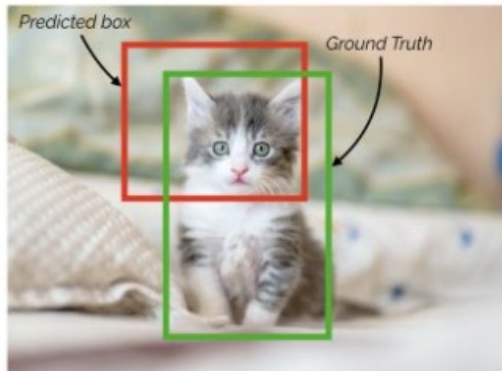
- IOU: Ground truth 영역과 추정한 영역 predicted box의 겹친 영역 비율



# 참고: mAP/Recall

[mAP 소개 링크](#)

10



IoU for the prediction = ~0.3

IoU threshold = 0.5 = False Positive (FP)

IoU threshold = 0.2 = True Positive (TP)

IOU 점수는 threshold를 어떻게 정하느냐에 따라 좌측 붉은 박스의 예측이 FP가 될 수도 있고, TP가 될 수도 있다.  
→ 임계값의 변화에 따른 평균화가 필요

$$\text{Recall\_Accuracy} = TP / (FP + FN)$$

TP = 실제 해당 객체가 있는데 IOU가 임계값을 넘게 나와 있다고 맞춘 사례의 수

FN = 실제 해당 객체가 있는데 IOU가 임계값을 넘지 못해 틀린 사례의 수

$$\text{mAP score} = (\sum_{\text{class}=1}^c \sum_{\text{threshold\_set}=1}^t \text{Recall\_Accuracy}) / (c \cdot t)$$

**mAP는 Recall\_Accuracy의 평균 정확도라 할 수 있다.**

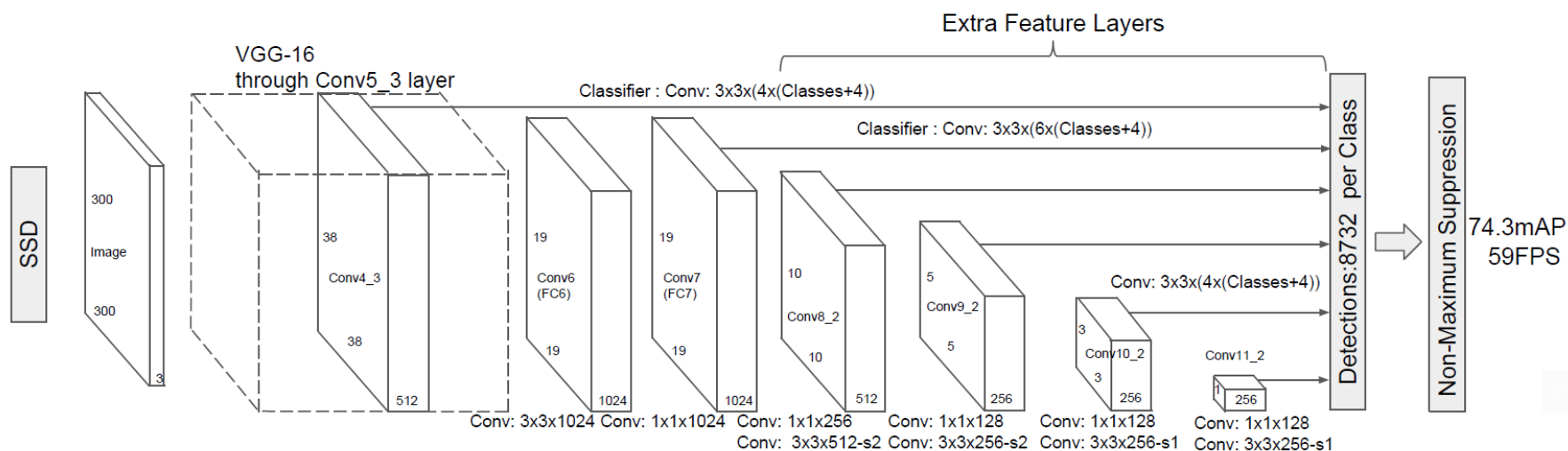
The mean Average Precision or mAP score is calculated by taking the mean AP over all classes and/or overall IoU thresholds, depending on different detection challenges that exist.

In PASCAL VOC2007 challenge, AP for one object class is calculated for an IoU threshold of 0.5. So the mAP is averaged over all object classes.

For the COCO 2017 challenge, the mAP is averaged over all object categories and 10 IoU thresholds.

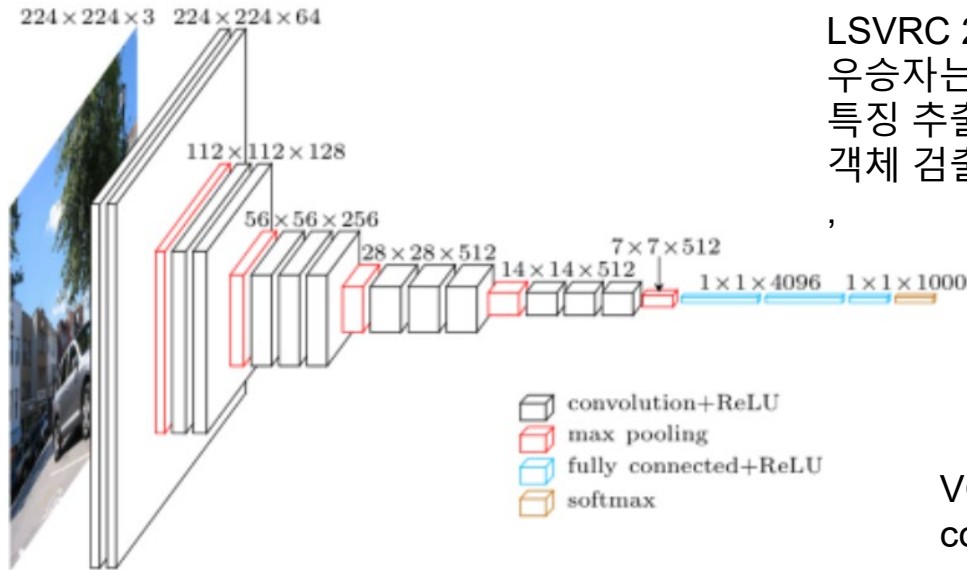
# SSD: 구조

11



# VGG net

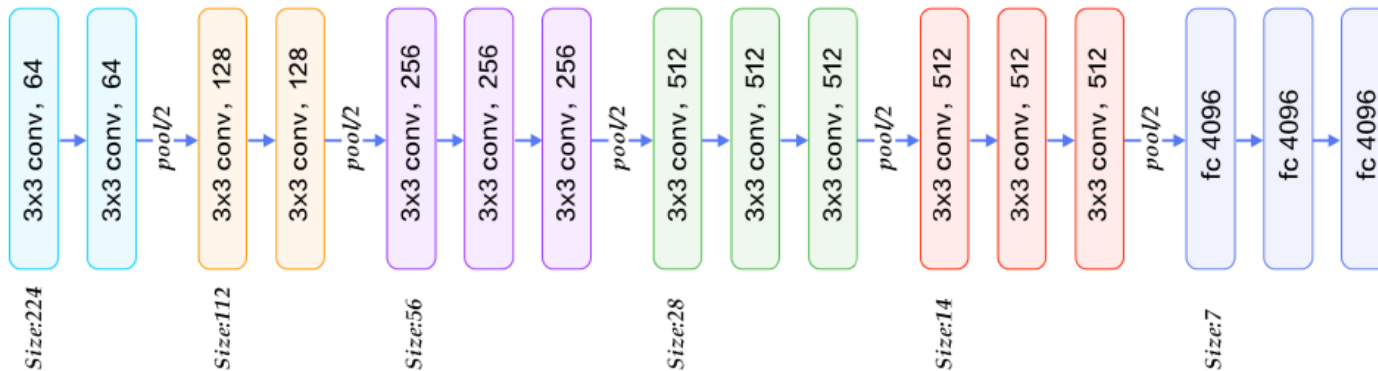
12



LSVRC 2014에서 옥스포드대학팀에서 소개.  
우승자는 아니지만 쉽고 쉽게 구현할 수 있고,  
특징 추출 능력이 우수  
객체 검출, 세그멘테이션에 많이 활용.

VGG16 architecture with the 3 x 3  
convolution layers, maxpooling, and fully  
connected layers

VGG architecture (input is 224x224x3)



# OpenCV Programming using DNN Classifier

13

1\_face\_detection\_opencv\_dnn.py

디텍터를 다운받는 소스(.py)를 다운받을 수 있는 곳: [model downloading script](#)

- **Face detector (FP16)**: Floating-point 16 version of the original Caffe implementation (5.1 MB)
- **Face detector (UINT8)**: 8-bit quantized version using TensorFlow (2.6 MB)

어떤 경우가 되었던 다음 2종류의 파일로 구성되어 있다.

## □ Caffe model의 경우

- ▣ res10\_300x300\_ssd\_iter\_140000\_fp16.caffemodel: This file contains the weights for the actual layers
- ▣ deploy.prototxt: This file defines the model architecture.

## □ TensorFlow Model의 경우

- ▣ opencv\_face\_detector\_uint8.pb: This file contains the weights for the actual layers.
- ▣ opencv\_face\_detector.pbtxt: This file defines the model architecture.

# 프로그램 수행절차

14

1\_face\_detection\_opencv\_dnn.py

## 1. 모델을 로드한다.

```
# 1) Caffe Model의 경우
net = cv2.dnn.readNetFromCaffe("deploy.prototxt", "res10_300x300_ssd_iter_140000_fp16.caffemodel")
# Tensorflow Model의 경우
#net = cv2.dnn.readNetFromTensorflow("opencv_face_detector_uint8.pb", "opencv_face_detector.pbtxt")
```

## 2. 영상 데이터를 300x300의 크기로 맞추고, BGR 화소에 대해 특정값을 빼준다.

```
# Create 4-dimensional blob(NCHW) from image:
blob = cv2.dnn.blobFromImage(image, 1.0, (300, 300), [104., 117., 123.], False, False)
print(blob.shape) ➡ (1, 3, 300, 300)
```

## 3. 입력을 인가하여 순방향 네트워크 흐름을 진행시킨다. ➡검출을 행한다.

```
# Set the blob as input and obtain the detections:
```

```
net.setInput(blob) 입력 연결
```

```
detections = net.forward() 네트워크 가동
```

```
print(type(detections), detections.shape)
```

➡ <class 'numpy.ndarray'> (1, 1, 53, 7)

검출된 얼굴의 수

[2]: Confidence(0~1)를 담고 있다.  
0.7이상 얼굴로 인정.

[3:7]=정규화된 얼굴 사각형  
영상의 폭, 높이를 곱하여 사용

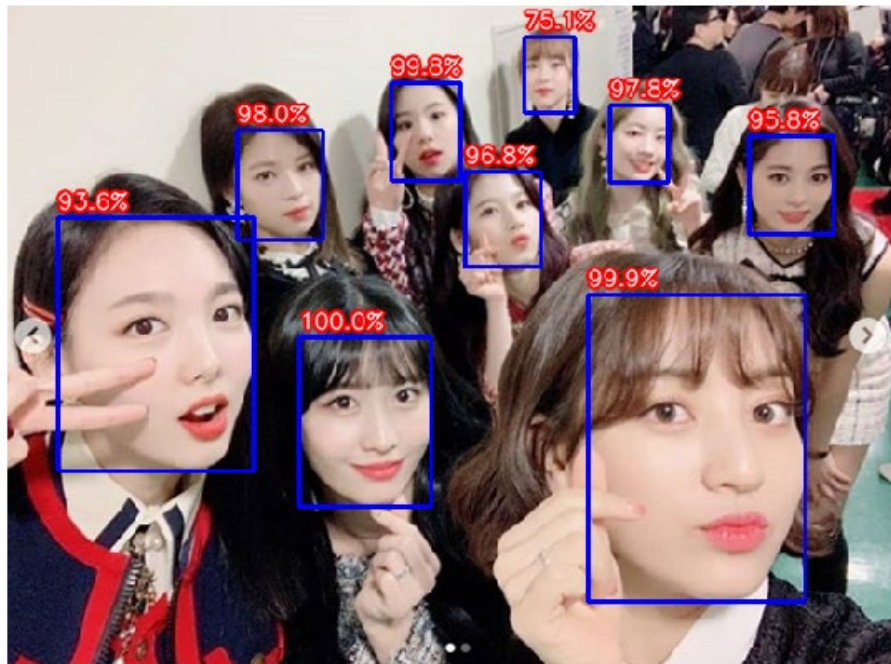


# 실험 결과

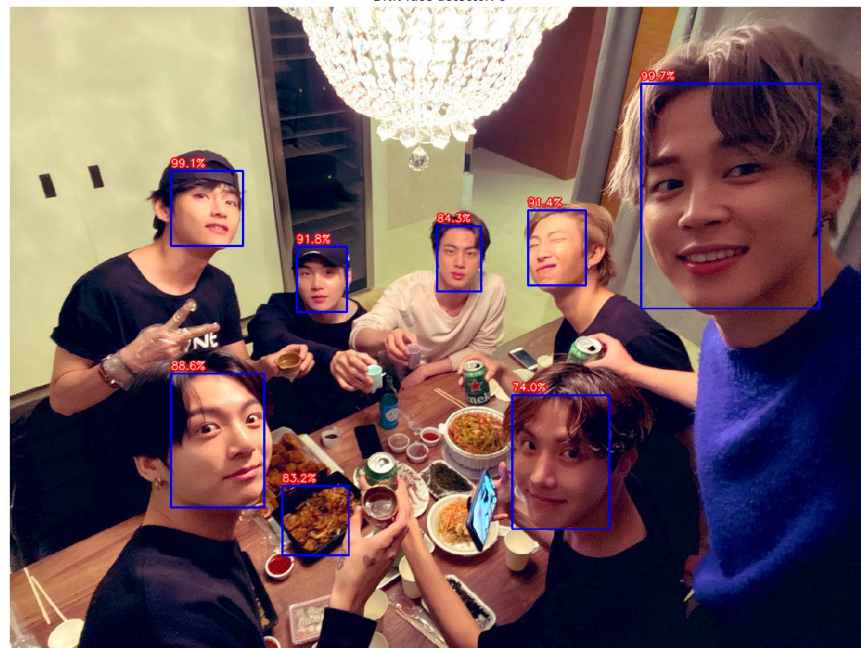
15

1\_face\_detection\_opencv\_dnn.py

DNN face detector: 9



DNN face detector: 8



Caffe Model 사용. Confidence 0.7이상만 표현



# DLIB face detection

HOG 기반

16

- frontal face detector 생성
  - ▣ `dlib.get_frontal_face_detector()`
  - ▣ **Histogram of Oriented Gradients (HOG)** features + SVM 기반 학습 알고리즘
  - ▣ a sliding window detection approach.
- 3,000개의 “*Labeled Faces in the Wild*” DB를 사용하여 학습)
- 불과 몇 개의 영상으로 학습이 가능

```
# Load frontal face detector from dlib:
```

```
detector = dlib.get_frontal_face_detector()
```

```
# Detect faces:
```

```
rects_1 = detector(gray, 0)      # 정상크기
```

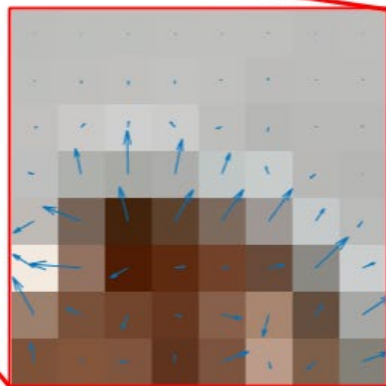
```
rects_2 = detector(gray, 1)      # 입력 영상을 2배로 사용
```

# HOG(1)

17

3\_face\_detection\_dlib\_hog.py

- 8x8 화소에 대해 Gradient magnitude와 Unsigned Gradients를 구한다.
- 보행자 검출의 경우에는 8x16 (64x128 pixels) sub window를 사용하였다.



2	3	4	4	3	4	2	2
5	11	17	13	7	9	3	4
11	21	23	27	22	17	4	6
23	99	165	135	85	32	26	2
91	155	133	136	144	152	57	28
98	196	76	38	26	60	170	51
165	60	60	27	77	85	43	136
71	13	34	23	108	27	48	110

Gradient Magnitude

80	36	5	10	0	64	90	73
37	9	9	179	78	27	169	166
87	136	173	39	102	163	152	176
76	13	1	168	159	22	125	143
120	70	14	150	145	144	145	143
58	86	119	98	100	101	133	113
30	65	157	75	78	165	145	124
11	170	91	4	110	17	133	110

Gradient Direction

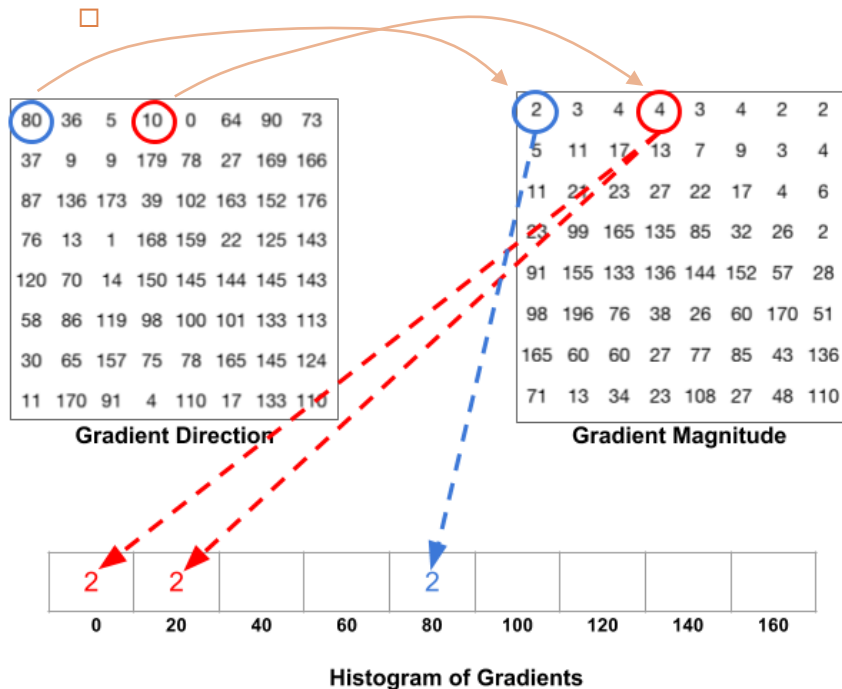
1) 8x8 블록에 대해  
Gradient magnitude와  
Direction(0~180)을 구한다.

# HOG(2)

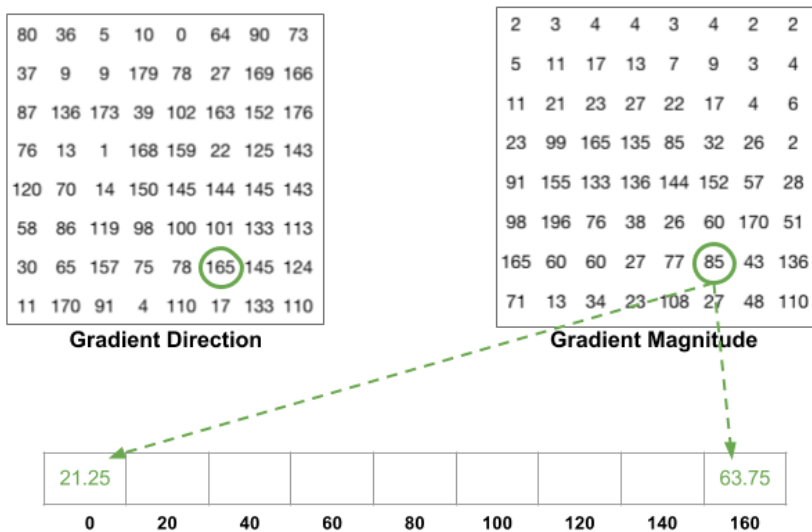
18

3\_face\_detection\_dlib\_hog.py

- Direction에 따라 9개의 bin(0, 20, 40, ... 160)에 magnitude의 값을 voting한다.
- Gradient는 0~180로 정한다.



방향이 80도이면서 그래디언트가 2인 화소는 bin 80에 2를 증가시킨다.  
 방향이 10이면 bin 0과 20에 그래디언트를 나누어 증가시킨다.



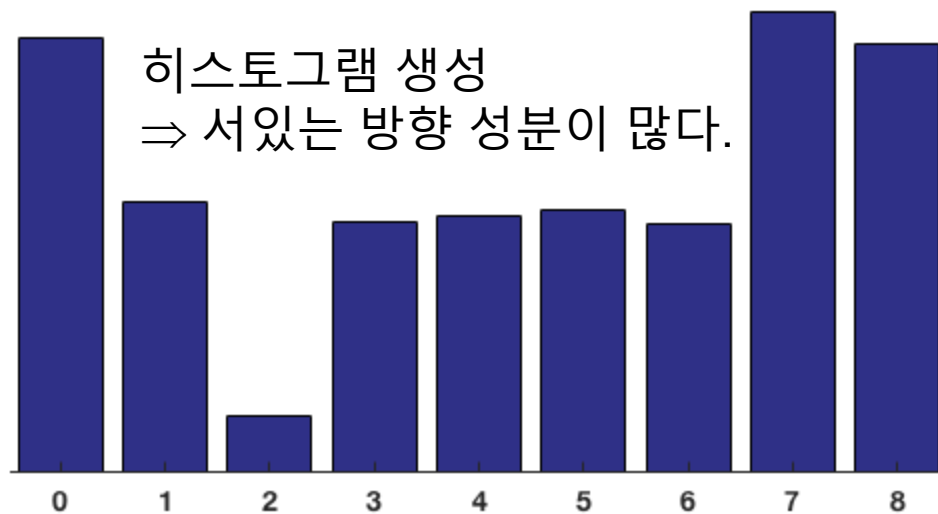
방향이 165도이면 bin 160과 bin 0에 그 가까운 비율만큼 나누어 증가시킨다.

2) 9개의 방향 bin에 magnitude만큼의 보팅을 행한다.

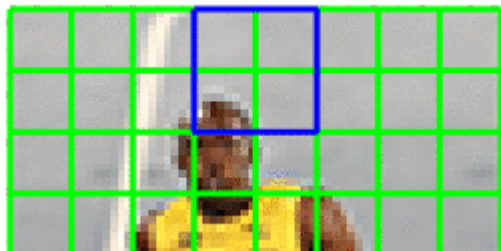
# HOG(3)

19

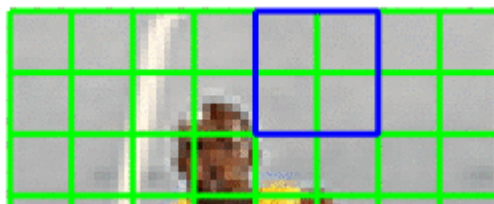
3\_face\_detection\_dlib\_hog.py



→ 16x16마다 이동하므로 7회 x 36 벡터 ..



3) 16x16 화소에 대해 정규화를 행한다.  
8x8 블록에 대해 9x1의 히스토그램이  
있으므로, 이를 모두 직렬로 연결하면  
36x1의 histogram vector가 만들어진다.



4) 16x16 블록을 옮겨가면서 매년  
36x1 벡터를 생성한다.



64x128 서브 윈도우에 대하여 가로로  
7번, 세로로 15번 = 105번의 연산  
각 연산마다 36개의 히스토그램  
벡터가 만들어지므로 보행자 검출  
서브 윈도우마다 36x105=3780  
차원의 벡터가 생성된다.

# DLIB face detection 실험

HOG 기반

20

3\_face\_detection\_dlib\_hog.py

**dlib HoG face detector: file=bts1.jpg, shape=(340, 640, 3)**

detector(gray, 0): num=0, time=0.0189



detector(gray, 0)

입력 영상을 원래 크기로 사용하면 검출 못함

detector(gray, 1): num=7, time=0.0947

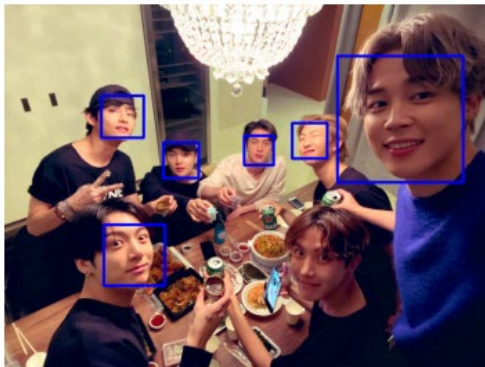


detector(gray, 1)

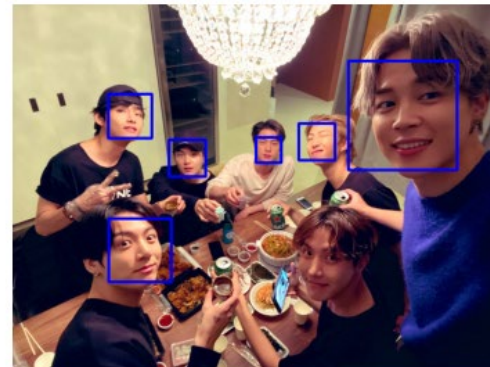
입력 영상을 2배로 하여야 검출되었음

**dlib HoG face detector: file=bts2.jpg, shape=(900, 1200, 3)**

detector(gray, 0): num=6, time=0.0888



detector(gray, 1): num=6, time=0.4458



입력 영상이 충분히 커서  
2배 스케일링을 할 필요  
없었음.  
사전에 적절한 입력  
영상의 크기를 설정하는  
것이 좋겠다.



# DLIB face detection

HOG 기반

21

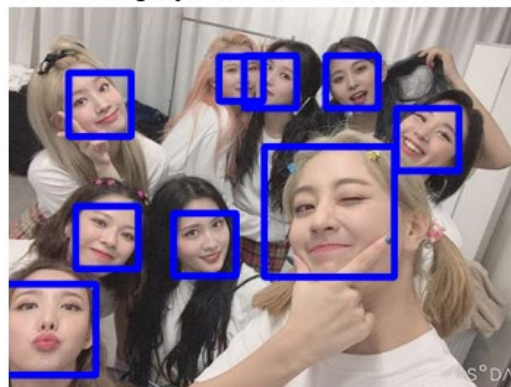
3\_face\_detection\_dlib\_hog.py

**dlib HoG face detector: file=twice1.jpg, shape=(381, 512, 3)**

detector(gray, 0): num=7, time=0.0170



detector(gray, 1): num=9, time=0.0868



detector(gray, 0)

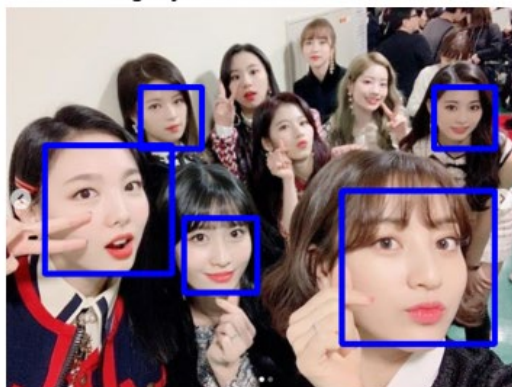
입력 영상이 작아 놓친 인물이 있음

detector(gray, 1)

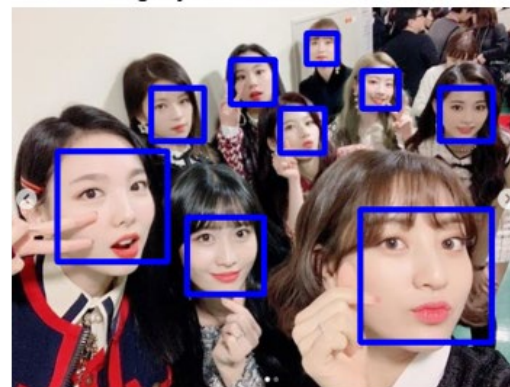
입력 영상을 2배로 하여 모두 검출됨

**dlib HoG face detector: file=twice2.jpg, shape=(448, 600, 3)**

detector(gray, 0): num=5, time=0.0239



detector(gray, 1): num=9, time=0.1167



# DLIB face detection

HOG 기반

22

3\_face\_detection\_dlib\_hog.py

**dlib HoG face detector: file=faces\_cartoon.jpg, shape=(419, 609, 3)**

detector(gray, 0): num=0, time=0.0230

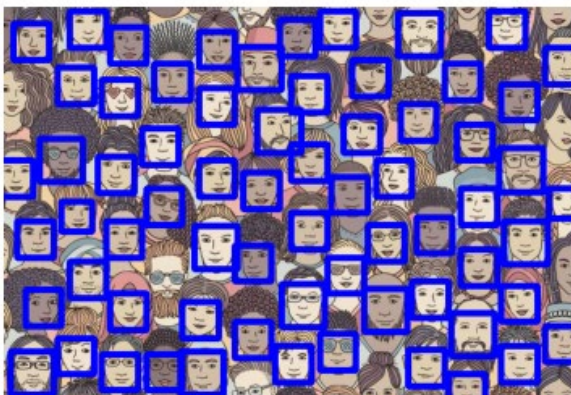


detector(gray, 1): num=66, time=0.1126

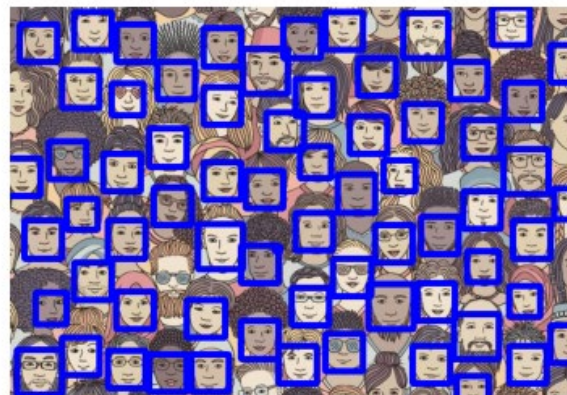


**dlib HoG face detector: file=faces\_cartoon.jpg, shape=(419, 609, 3)**

detector(gray, 3): num=67, time=1.3563



detector(gray, 4): num=68, time=6.9819





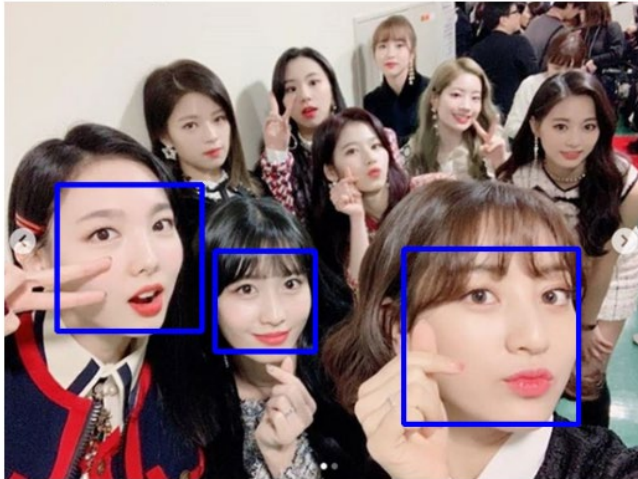
# DLIB face detection

CNN 기반

23

4\_face\_detection\_dlib\_CNN.py

dlib CNN face detector: file=twice2.jpg, shape=(448, 600, 3)  
cnn\_face\_detector(img, 0): num=3, time=0.6733



upsample\_time = 0      일부 인물 미검출

upsample\_time = 0    # 입력영상 *upsampling* 안한다.

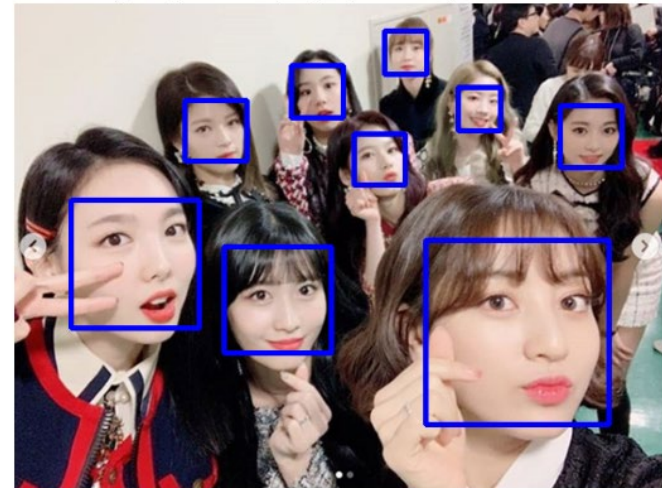
upsample\_time = 1    # *up sampling*을 1회 더 한 영상에 대해서 검출을 시도한다. 작은 얼굴을 찾아낼 수 있다.

upsample\_time = 3    # *up sampling*을 실시할 수록 시간은 더 늘어간다. 메모리 용량 초과를 불러올 수 있다.

rects = cnn\_face\_detector(img, upsample\_time)

CPU 사용시: time[sec.] for face detect= 6.69

dlib CNN face detector: file=twice2.jpg, shape=(448, 600, 3)  
cnn\_face\_detector(img, 1): num=9, time=0.7205



upsample\_time = 1      정면 얼굴 모두 검출

CPU 사용시: time[sec.] for face detect= 26.69

HOG보다 더욱 정밀하다. CPU로 처리할 경우 시간이 매우 오래 걸림



# DLIB face detection

CNN 기반

24

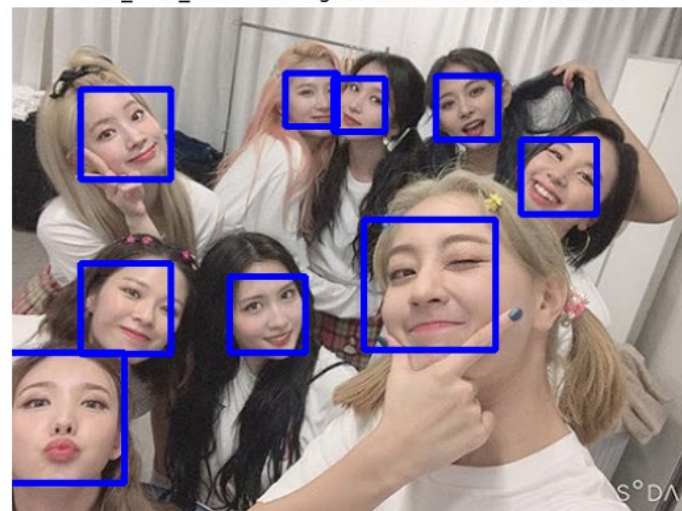
4\_face\_detection\_dlib\_CNN.py

**dlib CNN face detector: file=twice1.jpg, shape=(381, 512, 3)** **dlib CNN face detector: file=twice1.jpg, shape=(381, 512, 3)**

cnn\_face\_detector(img, 0): num=4, time=0.6652



cnn\_face\_detector(img, 1): num=9, time=0.7154



HoG의 검출 특성과 유사함...

# DLIB face detection

CNN 기반

25

4\_face\_detection\_dlib\_CNN.py

```
dlib CNN face detector  
file=faces_cartoon.jpg, shape=(419, 609, 3)
```

```
cnn_face_detector(img, 1): num=65, time=0.7083
```



HoG 검출: 0.1126초, 66명

CPU HoG로 검출한 것이 GPU로 수행한 것보다 빠름