



5주차 chap10

chap10 동적비전

10.1 모션 분석

물체의 모션 정보를 분석하는 알고리즘

비디오 video : 시간 순서에 따라 정지 영상을 나열한 구조

- 동영상 dynamic image 라고도 부름
- 프레임 frame : 비디오를 구성하는 영상 한 장, 2차원 공간에 정의
- 프레임 + 시간 축을 3차원 시공간 형성
- 컬러 영상의 프레임 $m \times n \times 3$ 구조 텐서, T장의 프레임을 담은 비디오는 $m \times n \times 3 \times T$ 의 4차원 구조 텐

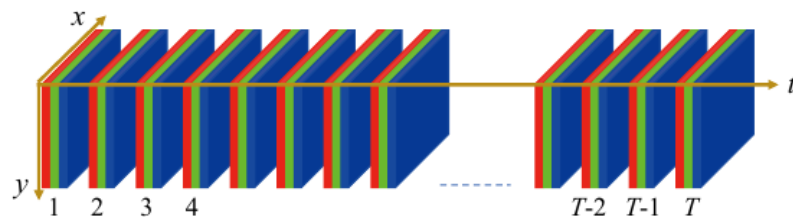


그림 10-2 3차원 공간에 표현되는 비디오

딥러닝 이전엔, 배경이 고정된 상황에서는 "차영상" 을 분석해 정보 획득

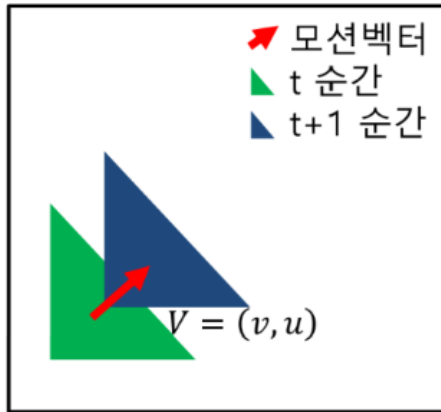
- t 순간 프레임의 (j,i) 화소값
- $d(j,i,t) = |f(j,i,0) - f(j,i,t)|$, $0 \leq j < m$, $0 \leq i < n$, $1 \leq t \leq T$

모션 벡터와 광류

<목표> 움직이는 물체는 연속 프레임에 명암 변화를 발생하니, 이를 분석해 역으로 물체의 모션 정보 확인

광류 optical flow : 화소별로 모션벡터 motion vector를 추정해 기록한 맵

== 이전 프레임과 현재 프레임의 차이를 이용하고 픽셀값과 픽셀과의 관계를 통해 각 픽셀의 이동motion을 계산해 추출



$f(y, x, t)$	$f(y, x, t + 1)$
11111111111111	11111111111111
11111111111111	11111111111111
11111111111111	11115111111111
11151111111111	11116511111111
11165111111111	11117651111111
11117651111111	11118765111111
11187651111111	11111111111111
11111111111111	11111111111111

출처 : https://gaussian37.github.io/vision-concept-optical_flow/

모션벡터를 추정하는 알고리즘은 물체가 이동, 회전, 크기변환을 하며, 환경이 조명변화와 잡음을 복합적으로 발생시켜 화소가 다음 순간에 어디로 이동했는지 확인하기는 어려움

=>

해결 방안으로 연속한 두 영상에서 같은 물체는 같은 명암으로 나타난다는 **밝기 항상성 brightness constancy** 조건 가정

시간 간격이 짧다면 테일러 근사로 전개 -> 광류조건식 optical flow constraint equation 생성

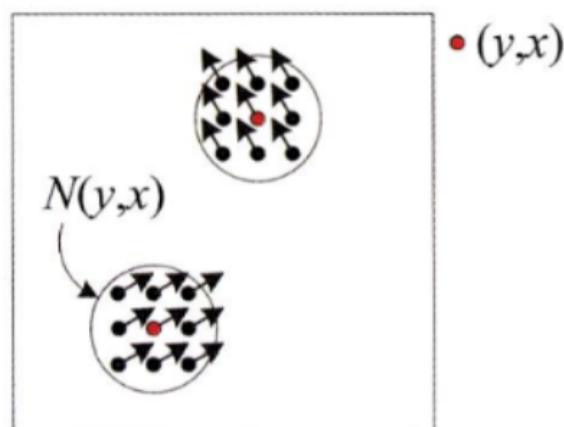
$$\frac{\partial f}{\partial y} v + \frac{\partial f}{\partial x} u + \frac{\partial f}{\partial t} = 0 \quad (10.4)$$

- 각각 v, u 는 dt 동안 y, x 방향으로 이동한 양으로 모션벡터에 해당
- $df/dy, df/dx$ 는 각각 y, x 방향의 명암 변화로서 에지검출에 사용한 에지연산자 사용
- (영상의 y 축 변화) $x \cdot v +$ (영상의 x 축 변화) $u +$ 영상의 t 축 변화 $= 0$
- 식은 하나지만 변수는 두개이므로 유일한 해를 확정할 수는 없지만 **이웃 화소 관계를 고려하면** 정확한 해 도출 가능

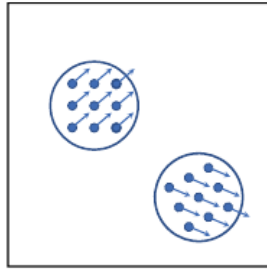
Lukas-Kanade 알고리즘

어떤 화소는 이웃 화소와 유일한 모션 벡터를 갖는다는 지역조건 사용

- 장점 | 연산량 적음
- 단점 | 정확도 낮음



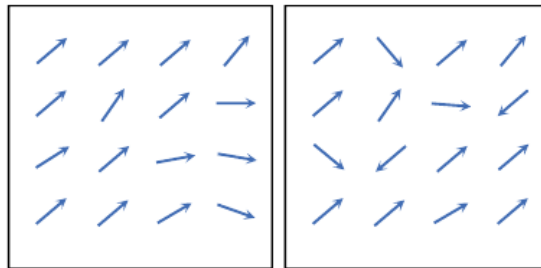
1. 밝기항상성
2. frame 간 움직임 적음
3. 픽셀 (y,x)를 중심으로 하는 윈도우 영역 N(y,x)의 optical flow 동일



Horn-Schun 알고리즘

영상 전체에 걸쳐 서서히 변해야 한다는 광역조건 사용

- 장점 | 정확도 높음
 - 단점 | 모든 픽셀을 계산하기에 속도 느림
1. 밝기항상성
 2. frame 간 움직임 적음
 3. optical flow 균일



Farneback 알고리즘

인접한 두 프레임 간의 움직임을 확장 다항식 기반으로 계산하는 dense optical flow의 한 종류

- 장점 | 정확도 높음
- 단점 | 계산 과정이 복잡해 시간 오래 걸림

```
# 10-1 Farneback 알고리즘으로 광류 추정
import numpy as np
import cv2 as cv
import sys

# 광류 맵 flow를 원본 영상 img에 그리는 함수
def draw_OpticalFlow(img, flow, step=16):
    # flow : img 와 동일 크기, 화소마다 y,x방향의 이동량인 모션벡터 보유
    # step : step 만큼 건너 모션 벡터 표시
    for y in range(step//2, frame.shape[0], step):
        for x in range(step//2, frame.shape[1], step): # step 만큼 건너 뛰어 화소 접근
            dx, dy = flow[y, x].astype(np.int) # 해당 화소의 모션 벡터 저장
            if (dx*dx+dy*dy)>1:
                cv.line(img, (x, y), (x+dx, y+dy), (0, 0, 255), 2) # 큰 모션 있는 곳은 빨간색
            else:
                cv.line(img, (x, y), (x+dx, y+dy), (0, 255, 0), 2) # 작으면 파란색

# 메인 프로그램
```

```

cap=cv.VideoCapture(0,cv.CAP_DSHOW) # 카메라와 연결 시도
if not cap.isOpened(): sys.exit('카메라 연결 실패')

prev=None # 시작 순간에는 이전 프레임이 없어 none 설정

while(1):
    ret,frame=cap.read() # 비디오를 구성하는 프레임 획득
    if not ret: sys('프레임 획득에 실패하여 루프를 나갑니다.')

    # 첫 프레임
    if prev is None: # 첫 프레임이면 광류 계산 없이 prev만 설정
        prev=cv.cvtColor(frame,cv.COLOR_BGR2GRAY) # 명암 변환
        continue

    # 첫 프레임 제외 모든 경우
    curr=cv.cvtColor(frame,cv.COLOR_BGR2GRAY) # 명암 변환
    flow=cv.calcOpticalFlowFarneback(prev,curr,None,0.5,3,15,3,5,1.2,0)
    # 광류맵 추출해 저장

    draw_OpticalFlow(frame,flow) # 광류맵 원본영상에 그리는 함수 호출
    cv.imshow('Optical flow',frame) # 윈도우 디스플레이

    prev=curr

    key=cv.waitKey(1) # 1밀리초 동안 키보드 입력 기다림
    if key==ord('q'): # 'q' 키가 들어오면 루프를 빠져나감
        break

cap.release() # 카메라와 연결을 끊음
cv.destroyAllWindows()

## 움직임이 일어난 곳에 큰 모션 벡터 발생

```

```

cv.calcOpticalFlowFarneback( prev, next, flow, pyr_scale, levels, winsize, iterations, poly_n, poly_sigma, flags )

```

- input 배열 이전 영상
- input 배열 현재 영상
- 출력 계산된 옵티컬 플로우
- 축소 비율
- 영상 개수
- 윈도우 크기
- 각 영상개수(levels)에서 알고리즘 반복 횟수
- 다항식 확장을 위한 이웃 픽셀 크기
- 가우시안 표준편차
- flags
 - 0 : ?
 - cv2.OPTFLOW_USE_INITIAL_FLOW : 인풋 플로우를 초기 플로우의 근사치 사용
 - cv2.OPTFLOW_FARNEBACK_GAUSSIAN : 가우스 원사이즈x원사이즈 필터 사용

희소 광류 추정을 이용한 **KT 추적 알고리즘**

지역 특징을 추적해 유리하도록 개조한 특징을 추출한 다음, 이들 특징을 광류 정보를 이용해 추적하는 방식으로 동작
 희소 광류 sparse optical flow : 지역 특징으로 추출된 점에서만 모션 벡터 추정

```

# 10-2 KLT 추적 알고리즘을 이용한 물체 추적 프로그램
import numpy as np
import cv2 as cv

cap=cv.VideoCapture('slow_traffic_small.mp4')

```

```

## vtest.avi 파일 이용

feature_params=dict(maxCorners=100,qualityLevel=0.3,minDistance=7,blockSize=7)
lk_params=dict(winSize=(15,15),maxLevel=2,
               criteria=(cv.TERM_CRITERIA_EPS|cv.TERM_CRITERIA_COUNT,10,0.03))

color=np.random.randint(0,255,(100,3)) # 추적 경로를 색으로 구별하기 위해 난수로 색 생성

ret,old_frame=cap.read() # 첫 프레임 읽기
old_gray=cv.cvtColor(old_frame,cv.COLOR_BGR2GRAY) # 명암 영상 변환
p0=cv.goodFeaturesToTrack(old_gray,mask=None,**feature_params) # 추적에 유리한 특징점 추출

mask=np.zeros_like(old_frame) # 물체의 이동 궤적을 그릴 영상 (추적 경로 표시할 mask 생성)

while(1):
    ret,frame=cap.read()
    if not ret: break

    new_gray=cv.cvtColor(frame,cv.COLOR_BGR2GRAY) # 명암영상 저장
    p1,match,err=cv.calcOpticalFlowPyrLK(old_gray,new_gray,p0,None,**lk_params)
    # 특정 위치에서 광류 & 추적 정보 계산

    if p1 is not None: # 양호한 쌍 선택
        good_new=p1[match==1]
        good_old=p0[match==1]

    for i in range(len(good_new)): # 이동 궤적 그리기
        a,b=int(good_new[i][0]),int(good_new[i][1])
        c,d=int(good_old[i][0]),int(good_old[i][1])
        mask=cv.line(mask,(a,b),(c,d),color[i].tolist(),2) # 특징점 선으로 넣기
        frame=cv.circle(frame,(a,b),5,color[i].tolist(),-1) # 특징점 원으로 넣기

    img=cv.add(frame,mask)
    cv.imshow('LTK tracker',img)
    cv.waitKey(30)

    old_gray=new_gray.copy() # 이번 것이 이전 것이 됨
    p0=good_new.reshape(-1,1,2)

cv.destroyAllWindows()

```

```
p1,match,err=cv.calcOpticalFlowPyrLK(old_gray,new_gray,p0,None,**lk_params)
```

- old_gray : 이전 프레임 영상
- new_gray : 다음 프레임 영상
- p0 : 이전 프레임의 코너 특징점, cv2.goodFeaturesToTrack()으로 검출
- p1 : 다음 프레임에서 이동한 코너 특징점 = 새로 찾은 특징점
- match : 결과 상태 벡터, nextPts와 같은 길이, 대응점이 있으면 1, 없으면 0 = 매칭 성공 여부
- err : 결과 에러 벡터, 대응점 간의 오차 = 매칭 오류

딥러닝 기반 광류 추정

광류는 매 화소마다 모션 벡터를 지정해야 하기에 참값 만들기 어려움



(a) KITTI 데이터셋

(b) Sintel 데이터셋

(c) Flying chairs 데이터셋

그림 10-5 광류 데이터셋[Dosovitskiy2015]

- 자율주행차를 위해 제작한 KITTI 데이터셋이 제공하는 광류데이터
차량에 설치된 여러 장치를 통해 광류의 참값을 자동으로 레이블링
- 컴퓨터그래픽을 제작한 애니메이션 영화 장면에서 광류의 참값을 레이블링한 Sintel 데이터 셋
컴퓨터그래픽 프로그램이 물체의 움직임에 대한 정보를 알고 있어 자동으로 레이블링 가능
- FlowNet이 자체제작한 Flying chairs 데이터셋
실제 영상에 의자를 인위적으로 추가한 다음 자동으로 광류 계산해 레이블링

분할결과를 활용한 광류 추정의 성능 개선하려는 시도 〇

자율주행 영상을 대상으로 알고리즘을 구상, 의미분할 대신 서로 다른 자동차 구분해주는 사례분할 적용



그림 10-6 사례 분할을 활용한 광류 추정[Bai2016]

10.2 추적

KLT 지역 특징을 추적하기에 뚜렷한 특징점이 나타나지 않는 물체 추적 불가

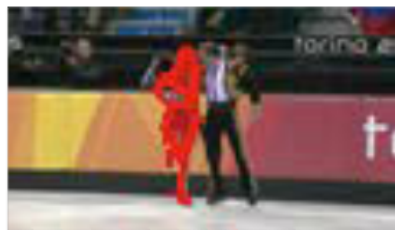
검출이나 분석할 결과를 잘 활용하면 박스나 영역 단위로 물체 추적 가능하며 추적 대상이 어떤 물체인지 인식 가능

<목적> 물체를 실시간으로 추적하는 알고리즘

재식별 re-identification : 끊긴 추적을 매칭해 같은 물체로 이어주는 과정

추적할 물체의 개수에 따라

- VOT Visual Object Tracking 시각 물체 추정
- MOT Multiple Object Tracking 다중 물체 추정 : 영상에 있는 여러 물체를 찾아야 하는데 첫 프레임에서 물체 위치를 지정 해주지 않고 추적할 물체의 부류 지정



(a) VOT(iceskater2 비디오)



(b) MOT(MOT20-03 비디오의 457번째 프레임)

그림 10-7 VOT 대회와 MOT 대회가 제공하는 데이터셋

TIP 각 데이터셋의 공식 사이트는 <https://votchallenge.net>와 <https://motchallenge.net>이다.

성능 척도

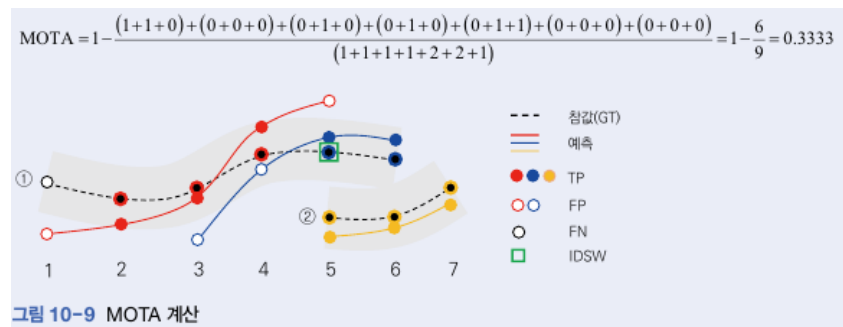
프레임 간의 연관성까지 고려해야 하기에 분할보다 복잡

MOT (MOT Accuracy)

$$MOTA = 1 - \frac{\sum_{t=1,T} (FN_t + FP_t + IDSW_t)}{\sum_{t=1,T} GT_t} \quad (10.5)$$

모든 프레임에서 아래의 값들 count

- GT_t : t 순간에서 참값
 - FN_t : t 순간에서 거짓 부정
 - FP_t : t 순간에서 거짓 긍정
 - $IDSW_t$: 번호 교환 오류 개수
- 이들 값은 매칭 알고리즘에 따라 정해지며 IoU가 임계값을 넘으면 매칭



- 비디오 길이 $T=7$
- 점선 궤적은 참값 (1-6 순간 지속하는 궤적(1), 5-7 순간 지속하는 궤적(2))
- IoU 임계값 = 0.5
- 순간1 : 빨간색 임계값 넘지 못해 매칭 불발 => FN, FP
- 순간2 : 매칭 발생 => TP
- 순간3 : 빨간색 매칭, 파란색 불발 => TP, FP
- 순간4 : 빨간색 매칭, 파란색 불발 => TP, FP
GT가 파란색과 더 가까우나 이전 순간에서 빨간색과 매칭되었기 때문에 성능평가프로그램이 이력을 고려해 결정
- 순간5 : 빨간색 임계값 넘지 못해 매칭 불발, 파란색 매칭, 노란색 매칭 => FP, TP, TP
궤적(1)입장에서 다른 물체와 쌍이 맺어져 빨간색과 파란색의 번호교환 발생
- 순간6 : 빨간색 매칭, 노란색 불발 => TP, TP
- 순간7 : 노란색 매칭 => TP

$$MOTA = 1 - \frac{(1+1+0)+(0+0+0)+(0+1+0)+(0+1+0)+(0+1+1)+(0+0+0)+(0+0+0)}{(1+1+1+1+2+2+1)} = 1 - \frac{6}{9} = 0.3333$$

단점 : 쌍 맺기 성공률이 낮더라도 검출 성공률 높으면 좋은 점수 부여

딥러닝 기반 추적

딥러닝 모델로 모델 검출 → 이웃프레임에서 검출된 물체 집합 매칭해 쌍 맺기

↓ details : 검출 → 특징 추출 → 거리계산 → 쌍맺기

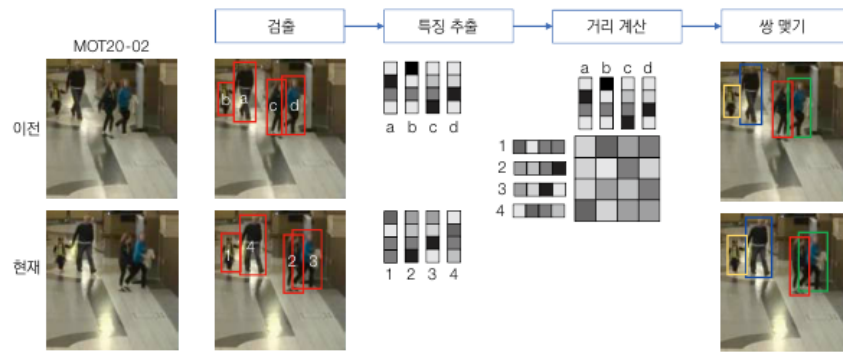


그림 10-10 물체 추적의 네 단계 처리 과정

step1 검출

현재 프레임에 물체 검출 알고리즘을 적용해 박스 찾기

→ 검출 성능이 전체 추적 성능을 좌우하며, 성능이 높은 모델 활용 ex. RCNN, YOLO

step2 특징 추출

단순히 박스의 위치 정보를 특징 사용 or CNN으로 특징 추출

step3 거리계산

이전 프레임 박스와 현재 프레임 박스의 거리 행렬 구하기

▼ 박스의 위치 정보

$$IoU = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

박스의 IoU 계산하고 1-IoU를 거리로 사용

▼ CNN

특징 벡터 사이의 거리 사용

step4 쌍맺기

거리 행렬을 분석해 이전 프레임 박스와 현재 프레임 박스를 쌍으로 맺어 물체의 이동 궤적 구성

- 쌍 맺는 일은 헝가리안 알고리즘 적용
- [알고리즘] Hungarian Maximum Matching Algorithm (Kuhn-Munkres algorithm)_(tistory.com)

SORT

SORT *Simple Online and Real-time Tracking*

검출 faster RCNN

현재 순간 t의 프레임에 faster RCNN을 적용해 물체 검출 = B_detection

추적 대상 물체를 사람으로 국한하기에 사람에 해당하는 박스만 남기고 모두 drop

특징 추출

이전 순간 $t-1$ 에 결정해 놓은 목표물의 위치 정보와 이동 이력 정보 사용

$$\mathbf{b} = (x, y, s, r, \dot{x}, \dot{y}, \dot{s}) \quad (10.6)$$

목표물의 정보 표현

- x, y (목표물의 중심 위치), s (크기) : 목표물의 위치 정보
- r 높이와 너비 비율 (고정된 값 사용)
- $\dot{x}, \dot{y}, \dot{s}$: 목표물이 이전에 이동했던 정보 누적한 이동 이력 정보

$B_{predict}$: t 순간을 예측한 박스

이동량을 나타내는 $\dot{x}, \dot{y}, \dot{s}$ dot를 각각 x, y, s 에 더하는 단순한 방법 이용

거리계산

1. $B_{detection}$ 과 $B_{predict}$ 에 있는 IoU 계산 → $1 - \text{IoU}$ 를 도출해 거리 변환
 2. 얻은 거리를 거리 행렬 채우기
- 두 박스 개수가 다른 경우 부족한 쪽에 가상 박스를 채워 정방행렬 도출

쌍맺기

거리 행렬을 이용, 헝가리안 알고리즘 적용해 최적의 매칭 쌍 찾기

▼ 헝가리안 알고리즘

최소비용이 되도록 작업자에게 과업을 할당하는 최적화 알고리즘

모든 노동자를 서로 다른 작업에 할당한다면, 이는 가장 적은 비용을 발생

“행 : 작업자 / 열 : 작업” 을 할당하는 개념

SORT에서는 행에 현재 프레임에 해당하는 $B_{detection}$, 열에 이전 프레임에서 예측된 $B_{predict}$ 을 배치한다. $B_{detection} = \{1, 2, 3, 4\}$ 가 있고, $B_{predict} = \{a, b, c\}$ 가 있다고 가정한다. 박스 쌍의 IoU를 계산하고 $1 - \text{IoU}$ 를 해당 요소에 기록하여 [그림 10-11(b)]의 거리 행렬을 얻었다고 가정한다. 4열에는 가상의 박스 d를 배치하였다. 이 행렬에 헝가리안 알고리즘을 적용하면 1-b, 2-d, 3-a, 4-c 쌍을 얻는데 2-d 쌍은 버린다.

TIP 헝가리안 알고리즘의 구체적인 동작 원리는 다음 문서를 참고한다.

https://web.archive.org/web/20120105112913/http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf

	a	b	c
1	2	5	1
2	1	3	4
3	2	3	6

(a) 작업자에게 과업을 할당하는 문제

	a	b	c	d
1	0.9	0.2	0.7	1.0
2	0.7	0.4	0.8	1.0
3	0.1	0.3	1.0	1.0
4	1.0	0.7	0.3	1.0


(b) $B_{predict}$ 박스를 $B_{detection}$ 박스에 할당하는 문제

그림 10-11 헝가리안 알고리즘

후처리

B_predict 에 있는 목표물 식 정보를 갱신

- 매칭 성공, 매칭 실패 목표물을 구별해 처리
- 매칭 성공
 - 쌓이 된 박스 정보로 기존 x,y,s,r 값 대치
 - 이동 이력정보는 칼만 필터(잡음과 변형이 심한 시계열 데이터에서 이전 샘플의 분포를 감안해 현재 측정치 보완하는 기법)를 사용해 갱신
- 매칭 실패
 - x,y,s dot에 x,y,s를 각각 더해 x,y,s 갱신

 거리 행렬 구할 때 박스의 IoU만 사용한 것이 오류가 발생하게 함

⇒ DeepSORT : 박스의 IoU와 CNN으로 구한 특징을 같이 사용해 성능 향상

```
# 10-3 SORT 로 사람 추적
import numpy as np
import cv2 as cv
import sys

##### 프로그램 9-1과 동일 #####
# YOLO v3 읽어 모델을 구성하는 함수와 모델로 물체를 검출하는 함수 포함
def construct_yolo_v3(): # 모델 구성
    f=open('coco_names.txt', 'r')
    class_names=[line.strip() for line in f.readlines()]

    model=cv.dnn.readNet('yolov3.weights','yolov3.cfg')
    layer_names=model.getLayerNames()
    out_layers=[layer_names[i-1] for i in model.getUnconnectedOutLayers()]

    return model,out_layers,class_names

def yolo_detect(img,yolo_model,out_layers): # 물체 검출
    height,width=img.shape[0],img.shape[1]
    test_img=cv.dnn.blobFromImage(img,1.0/255,(448,448),(0,0,0),swapRB=True)

    yolo_model.setInput(test_img)
    output3=yolo_model.forward(out_layers)

    box,conf,id=[],[],[] # 박스, 신뢰도, 부류 번호
    for output in output3:
        for vec85 in output:
            scores=vec85[5:]
            class_id=np.argmax(scores)
            confidence=scores[class_id]
            if confidence>0.5: # 신뢰도가 50% 이상인 경우만 취함
                centerx,centery=int(vec85[0]*width),int(vec85[1]*height)
                w,h=int(vec85[2]*width),int(vec85[3]*height)
                x,y=int(centerx-w/2),int(centery-h/2)
                box.append([x,y,x+w,y+h])
                conf.append(float(confidence))
                id.append(class_id)

    ind=cv.dnn.NMSBoxes(box,conf,0.5,0.4)
    objects=[box[i]+conf[i]+id[i] for i in range(len(box)) if i in ind]
    return objects

#####

model,out_layers,class_names=construct_yolo_v3() # YOLO 모델 생성
colors=np.random.uniform(0,255,size=(100,3))
# 추적하는 물체를 서로 다른 색으로 표현하기 위해 100개 색으로 트랙 구분
```

```

from sort import Sort

sort=Sort()

cap=cv.VideoCapture(0,cv.CAP_DSHOW)
if not cap.isOpened(): sys.exit('카메라 연결 실패')

while True:
    ret,frame=cap.read()
    if not ret: sys.exit('프레임 획득에 실패하여 루프를 나갑니다.')

    # 검출 후, 사람에 해당하는 것만 저장
    res=yolo_detect(frame,model,out_layers) # 물체 검출
    persons=[res[i] for i in range(len(res)) if res[i][5]==0] # 부류 0은 사람

    if len(persons)==0:
        tracks=sort.update() # 검출된 사람이 없을 때
    else:
        tracks=sort.update(np.array(persons)) # 검출된 사람이 있을 때 함수 호출

    # 추적 물체의 번호에 따라 색을 달리 해 직사각형과 물체 번호 표시
    for i in range(len(tracks)):
        x1,y1,x2,y2,track_id=tracks[i].astype(int)
        cv.rectangle(frame,(x1,y1),(x2,y2),colors[track_id],2)
        cv.putText(frame,str(track_id),(x1+10,y1+40),cv.FONT_HERSHEY_PLAIN,3,colors[track_id],2)

    cv.imshow('Person tracking by SORT',frame) # 추적 물체 윈도우 표시

    key=cv.waitKey(1)
    if key==ord('q'): break

cap.release() # 카메라와 연결을 끊음
cv.destroyAllWindows()

```

- 카메라 시야에서 벗어났다 다시 들어오면 다른 번호 부여
- SORT는 재식별하지 않아, 다른 물체로 간주

10.3 MediaPipe를 이용해 비디오에서 사람 인식

비디오에 나타난 사람을 인식하는 여러 방법 설명

MediaPipe : 구글이 제공하는 기계학습 개발 프레임워크, 여러가지 유용한 비디오 처리 솔루션 제공

- 그래프를 이용해 비디오의 입력과 출력을 일관성있게 표현과 제어

▼ MediaPipe 파이션 인터페이스

face detection 얼굴 검출

face mesh 얼굴 그물망

hands 손

selfie segmentation 셀피 분할

pose 자세

holistic 몸 전체

objectron 오브젝트론

얼굴 검출

MediaPipe가 제공하는 BlazeFace라는 얼굴 검출 방법 적용

- SSD : faster RCNN을 개조한 모델

- BlazeFace는 SSD를 얼굴 검출에 맞게 처리 속도를 중요히 여기며 개조
 - 여러 박스의 정보를 가중 평균함으로써 정확률 향상
 - 랜드마크를 이용해 얼굴의 특정 부위에 장식을 다는 증강현실 구현 가능

```
# 10-4 BlazeFace로 얼굴 검출
import cv2 as cv
import mediapipe as mp

img=cv.imread('BSDS_376001.jpg')

mp_face_detection=mp.solutions.face_detection # 얼굴 검출 담당
mp_drawing=mp.solutions.drawing_utils # 검출 결과 그리는 용도

# 얼굴 검출에 쓸 객체 생성
face_detection=mp_face_detection.FaceDetection(model_selection=1,min_detection_confidence=0.5)
# 검출 수행하고 결과 저장
res=face_detection.process(cv.cvtColor(img,cv.COLOR_BGR2RGB)) # 채널 순서 BGR -> RGB 변환

if not res.detections:
    print('얼굴 검출에 실패했습니다. 다시 시도하세요.')
else:
    for detection in res.detections:
        mp_drawing.draw_detection(img,detection)
    cv.imshow('Face detection by MediaPipe',img)

cv.waitKey()
cv.destroyAllWindows()
```

```
mp_face_detection.FaceDetection(model_selection=1,min_detection_confidence=0.5)
```

- model_selection
 - 0 : 카메라로부터 2미터 이내 가깝게 있을 때 적합
 - 1 : 5미터 이내로 조금 멀리 있을 때 적합
- min_detection_confidence :
 - 0~1 사이의 실수 설정
 - 검출 신뢰도가 설정한 값보다 큰 경우만 검출 성공으로 간주

```
# 10-5 비디오에서 얼굴 검출
import cv2 as cv
import mediapipe as mp

mp_face_detection=mp.solutions.face_detection
mp_drawing=mp.solutions.drawing_utils

face_detection=mp_face_detection.FaceDetection(model_selection=1,min_detection_confidence=0.5)
#####

cap=cv.VideoCapture(0,cv.CAP_DSHOW) # OpenCV를 이용해 웹캠 연결 시도

while True:
    ret,frame=cap.read()
    if not ret:
        print('프레임 획득에 실패하여 루프를 나갑니다.')
        break

    res=face_detection.process(cv.cvtColor(frame,cv.COLOR_BGR2RGB)) # 얼굴 검출

    # 검출된 얼굴을 프레임에 표시
    if res.detections:
        for detection in res.detections:
```

```

        mp_drawing.draw_detection(frame, detection)

# 윈도우에 얼굴 영상 display
cv.imshow('MediaPipe Face Detection from video', cv.flip(frame, 1)) # 좌우 반전해 거울에 비친 모양 보여줌
if cv.waitKey(5) == ord('q'):
    break

cap.release()
cv.destroyAllWindows()

```

결과 : 변화있는 부분을 음영으로 처리해 쉽게 구별 가능

실제 영상에 가상의 물체를 추가할 때,

물체의 배경을 투명하게 처리하지 않으면 조각 영상을 붙인 듯한 느낌 O → 현실감 떨어짐

```

# 10-6 얼굴을 장식하는 증강 현실 구현
import cv2 as cv
import mediapipe as mp

# 장신구로 사용할 영상 준비
dice = cv.imread('dice.png', cv.IMREAD_UNCHANGED) # 증강 현실에 쓸 장신구(4개 채널) 읽어오기
dice = cv.resize(dice, dsize=(0, 0), fx=0.1, fy=0.1) # 영상 10% 축소
w, h = dice.shape[1], dice.shape[0] # 너비와 높이 저장

mp_face_detection = mp.solutions.face_detection
mp_drawing = mp.solutions.drawing_utils

face_detection = mp_face_detection.FaceDetection(model_selection=1, min_detection_confidence=0.5)

cap = cv.VideoCapture(0, cv.CAP_DSHOW)

while True:
    ret, frame = cap.read()
    if not ret:
        print('프레임 획득에 실패하여 루프를 나갑니다.')
        break

    res = face_detection.process(cv.cvtColor(frame, cv.COLOR_BGR2RGB))

    if res.detections:
        for det in res.detections:
            # 검출된 얼굴에 장신구 달고 윈도우에 디스플레이
            p = mp_face_detection.get_key_point(det, mp_face_detection.FaceKeyPoint.RIGHT_EYE)
            # 검출된 얼굴 정보를 담고 있는 det에서 오른쪽 눈 위치를 꺼내 저장

            # 필요한 장신구 영상의 왼쪽 위, 오른쪽 아래 좌표 계산 (오른쪽 눈 중심으로 장신구 배치를 위해)
            x1, x2 = int(p.x * frame.shape[1] - w // 2), int(p.x * frame.shape[1] + w // 2)
            y1, y2 = int(p.y * frame.shape[0] - h // 2), int(p.y * frame.shape[0] + h // 2)

            # 장신구 영상이 원본 영상 안에 머무는지 확인
            if x1 > 0 and y1 > 0 and x2 < frame.shape[1] and y2 < frame.shape[0]:
                # 원본 영상의 해당 위치에 장신구 영상 혼합
                alpha = dice[:, :, 3] / 255 # 투명도를 나타내는 알파값
                frame[y1:y2, x1:x2] = frame[y1:y2, x1:x2] * (1 - alpha) + dice[:, :, 3] * alpha # 혼합

    cv.imshow('MediaPipe Face AR', cv.flip(frame, 1)) # 거울 영상
    if cv.waitKey(5) == ord('q'):
        break

cap.release()
cv.destroyAllWindows()

# 얼굴이 빠르게 움직여도 지연없이 비디오 자연스럽게 디스플레이

```

얼굴 그물망 검출

얼굴에서 랜드마크를 검출하는 얼굴정렬 *face alignment*

→ 얼굴 영상을 인식하거나 정교하게 증강현실을 적용하는 등의 응용에서!!

FaceMesh 솔루션을 활용해 468개의 랜드마크 검출

- 얼굴에 조밀하게 랜드마크 배치해 매끄러운 얼굴 그물망 얻을 수 있음
- 랜드마크는 3차원 좌표로 표현
- 매 프레임에서 예측한 곳에 실제 얼굴이 있는지 신뢰도 계산, 신뢰도가 임계값보다 낮으면 다시 적용

```
# 10-7 FaceMesh로 얼굴 그물망 검출
import cv2 as cv
import mediapipe as mp

mp_mesh=mp.solutions.face_mesh # 얼굴 그물망 검출 모듈 저장
mp_drawing=mp.solutions.drawing_utils # 검출 결과를 그리는데 쓰는 모듈 저장
mp_styles=mp.solutions.drawing_styles # 그리는 유형 지정하는 모듈 저장

# 얼굴 그물망 검출에 쓸 객체 생성
mesh=mp_mesh.FaceMesh(max_num_faces=2,refine_landmarks=True,
                      min_detection_confidence=0.5,min_tracking_confidence=0.5)
cap=cv.VideoCapture(0,cv.CAP_DSHOW)

while True:
    ret,frame=cap.read()
    if not ret:
        print('프레임 획득에 실패하여 루프를 나갑니다.')
        break

    res=mesh.process(cv.cvtColor(frame,cv.COLOR_BGR2RGB)) # 실제 그물망 검출 수행한 뒤 결과 저장

    if res.multi_face_landmarks:
        # 검출된 얼굴 각각에 대해 그물망을 그리는 일 반복
        for landmarks in res.multi_face_landmarks:
            # 그물망 그리기 -> 변형하면 다양한 형태의 그물망 그리기 가능
            mp_drawing.draw_landmarks(image=frame, landmark_list=landmarks,
                                      connections=mp_mesh.FACEMESH_TESSELATION,
                                      landmark_drawing_spec=None,
                                      connection_drawing_spec=mp_styles.get_default_face_mesh_tesselation_style())

            # 얼굴 경계, 눈, 눈썹 그리기
            mp_drawing.draw_landmarks(image=frame, landmark_list=landmarks,
                                      connections=mp_mesh.FACEMESH_CONTOURS,
                                      landmark_drawing_spec=None,
                                      connection_drawing_spec=mp_styles.get_default_face_mesh_contours_style())

            # 눈동자 그리기
            mp_drawing.draw_landmarks(image=frame, landmark_list=landmarks,
                                      connections=mp_mesh.FACEMESH_IRISES,
                                      landmark_drawing_spec=None,
                                      connection_drawing_spec=mp_styles.get_default_face_mesh_iris_connections_style())

    cv.imshow('MediaPipe Face Mesh',cv.flip(frame,1)) # 좌우반전
    if cv.waitKey(5)==ord('q'):
        break

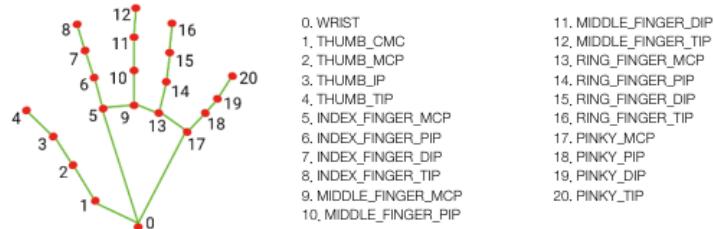
cap.release()
cv.destroyAllWindows()
```

```
mp_mesh.FaceMesh(
max_num_faces=2,refine_landmarks=True,min_detection_confidence=0.5,min_tracking_confidence=0.5)
```

- 얼굴 처리 개수
- 눈과 입에 있는 랜드마크 더 정교하게 검출
- 얼굴 검출 신뢰도가 지정값 초과일 때 성공으로 간주

- 랜드마크 추적 신뢰도가 지정값 미만이면 실패로 간주하고 새로운 얼굴 검출 수행

손 랜드마크 검출



얼굴 그물망과 마찬가지로 손 랜드마크도 3차원 좌표로 표현

```
mp_hand.Hands(max_num_hands=2,static_image_mode=False,min_detection_confidence=0.5,min_tracking_confidence=0.5)
```

- 얼굴 처리 개수
- static_image_mode
 - False : 입력을 비디오로 간주하고 첫 프레임에 손 검출을 담당하는 BlazeHand 적용, 이후 추적 사용
 - True : 매 프레임에 적용하기에 정확도가 높지만 시간 많이 걸림
- 얼굴 검출 신뢰도가 지정값 초과일 때 성공으로 간주
- 랜드마크 추적 신뢰도가 지정값 미만이면 실패로 간주하고 새로운 얼굴 검출 수행

```
# 10-8 손 랜드마크 검출
import cv2 as cv
import mediapipe as mp

mp_hand=mp.solutions.hands # 손검출 담당 모듈 저장
mp_drawing=mp.solutions.drawing_utils
mp_styles=mp.solutions.drawing_styles

# 손 랜드마크 검출에 사용할 객체 생성
hand=mp_hand.Hands(max_num_hands=2,static_image_mode=False,min_detection_confidence=0.5,min_tracking_confidence=0.5)

cap=cv.VideoCapture(0,cv.CAP_DSHOW)

while True:
    ret,frame=cap.read()
    if not ret:
        print('프레임 획득에 실패하여 루프를 나갑니다.')
        break

    res=hand.process(cv.cvtColor(frame,cv.COLOR_BGR2RGB)) # 실제 손 랜드마크 검출 수행

    if res.multi_hand_landmarks: # 검출된 손 있는지 파악
        for landmarks in res.multi_hand_landmarks: # 검출된 손 각각에 그물망 그리는 일 반복
            mp_drawing.draw_landmarks(frame,landmarks,mp_hand.HAND_CONNECTIONS,
                                      mp_styles.get_default_hand_landmarks_style(),
                                      mp_styles.get_default_hand_connections_style()) # 손 랜드마크 그리기

    cv.imshow('MediaPipe Hands',cv.flip(frame,1)) # 좌우반전
    if cv.waitKey(5)==ord('q'):
        break

cap.release()
cv.destroyAllWindows()
```

10.4 자세 추정과 행동 분류

라이브러리로 얼굴과 손을 검출하고 사람의 자세를 추정하는 프로그래밍 실습

자세 추정 결과를 기반으로 행동 분류 수행

자세추정

pose estimation : 정지영상 또는 비디오 분석해 전신에 있는 관절 위치 파악

- 높은 성능의 자세 추정은 많은 응용에 활용 가능 ex. 선수의 자세 교정, 환자의 재활 등
- 관절 : 랜드마크 또는 키포인트라 부름

추정 척도

AP 와 mAP 을 사용

- 예측 랜드마크와 참값 랜드마크 사이의 OKS *object keypoint similarity* 를 계산
- 임계값을 변화해 정밀도와 재현률을 구해 그래프를 구성.
- 그래프로 부터 AP 측정
- 관절별로 AP계산하고 모든 관절의 AP를 평균하면 mAP

인체모델



(a) BlazePose의 골격 표현[Bazarevsky2020]

(b) SMPL 부피 표현[Loper2015]

그림 10-15 인체 모델

(a) 골격 표현법

- 랜드마크의 연결관계를 그래프로 표현
- 랜드마크를 2차원 좌표와 3차원 좌표로 표현하는 경우로 구분

(b) 부피 표현법 : 기본적으로 3차원 좌표 사용

- 3차원 랜드마크와 3차원 부피로 표현

1) 정지영상에서 자세 추정

랜드마크 검출하는 방법 두개

1-1. DeepPose

딥러닝을 자세 추정에 처음 적용한 모델

- 방법

input : 220*220*3영상

convolution layer : 5개, 13*13*192 특징맵 변환

fully connected layer : 2개, $2k$ 의 실수 출력 (k 는 랜드마크 개수)

- 전체 영상을 보고 좌표를 예측하며 가림이 발생한 관절의 위치까지 예측

BUT. 정확률이 떨어져

추정한 관절위치에서 패치를 잘라내 세밀히 위치 조정하는 후처리 단계 O

1-2 열지도 회귀 *heatmap regression*

랜드마크 좌표에 가우시안을 씌운 맵 예측

k 개의 2차원 맵 출력

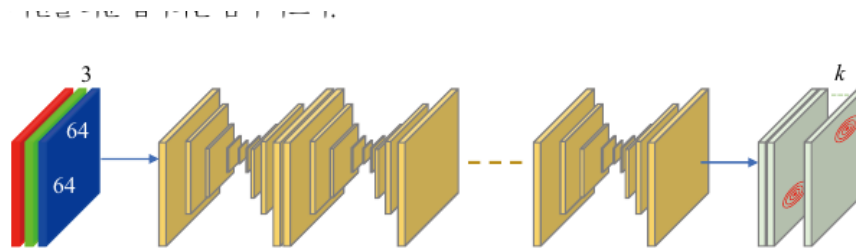


그림 10-16 열지도 회귀 기법을 사용하는 SHG 모델

2) 비디오에서 자세 추정

프레임이 시간 순서로 흐르는 구조인 비디오는 프레임 각각을 자세 추정 모델에 저장해 비디오의 자세 추정 결과 get

- 사람의 동작은 매끄럽게 변한다는 사실을 이용해 일관성 있는 자세 추정
- 다른 프레임의 랜드마크를 참조해 안보이는 랜드마크 위치 추론

이웃프레임을 고려하는 접근 방법

- 광류사용방법
- 순환신경망방법

3) BlazePose에서 자세 추정

랜드마크 검출하는 접근방법 중, 좌표회귀와 열지도 회귀가 있는데 이 둘을 모두 사용해 성능 향상

```
# 10-9 BlazePose를 이용해 자세 추정
import cv2 as cv
import mediapipe as mp

mp_pose=mp.solutions.pose # 자세 추정 담당하는 모듈 읽어 객체 저장
mp_drawing=mp.solutions.drawing_utils
mp_styles=mp.solutions.drawing_styles

# 랜드마크 검출에 쓸 객체 생성
pose=mp_pose.Pose(static_image_mode=False,enable_segmentation=True,min_detection_confidence=0.5,min_tracking_confidence=0.5)
## enable_segmentation=True : 전경과 배경 분할 지시, 배경을 흐릿하게 만드는 등 응용 가능

cap=cv.VideoCapture(0,cv.CAP_DSHOW)

while True:
    ret, frame=cap.read()
```

```

if not ret:
    print('프레임 획득에 실패하여 루프를 나갑니다.')
    break

res=pose.process(cv.cvtColor(frame,cv.COLOR_BGR2RGB)) # 실제 자세 추정

# 자세를 원본영상에 겹쳐 보이기
mp_drawing.draw_landmarks(frame,res.pose_landmarks,
                           mp_pose.POSE_CONNECTIONS,
                           landmark_drawing_spec=mp_styles.get_default_pose_landmarks_style())

cv.imshow('MediaPipe pose',cv.flip(frame,1)) # 좌우반전
if cv.waitKey(5)==ord('q'):
    mp_drawing.plot_landmarks(res.pose_world_landmarks,mp_pose.POSE_CONNECTIONS)
    break

cap.release()
cv.destroyAllWindows()

```

행동분류

아직 사람 성능에 못 미침

비디오의 일부를 보여주면 이후 행동을 예측하는 행동예측 문제는 행동 분류보다 어려움