B-1. 특징기술자 검출

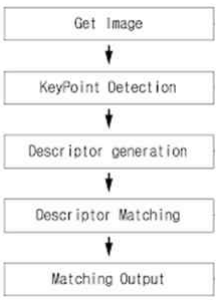
1

객체 인식

- 영상으로부터 검출된 특징정보(feature)를 이용
  - 기계학습을 통해 생성된 분류기로 인식하는 방법
  - **특징기술자(feature descriptor)를 매칭하여 인식하는 방법**

- **특징정보의 조건**
  - 영상의 여러 변화에 대해서 불변성을 가져야 한다
    - 노이즈, 스케일 변화, 회전, 시점 변화, 조명 변화

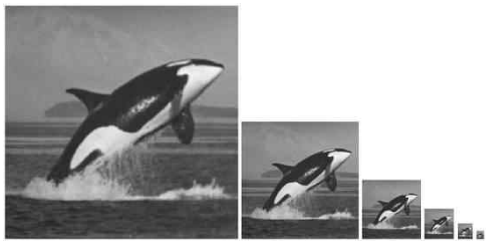
- **특징기술자 기반 객체 인식**
  1) 입력영상에서 **특징점(keypoints)**을 검출
  2) 특징점 주변 모양을 특징정보(local feature)로 나타내는 **특징기술자** 추출
  3) 특징기술자를 데이터베이스 내의 특징기술자와 매칭하여 객체를 식별



2

이미지 피라미드(Pyramid)

- 이미지 처리를 하는데 있어 동일한 이미지의 여러 해상도의 다양한 크기를 필요로 하는 경우가 있음
  - 예를 들어, 동일한 사람의 얼굴이 단체 사진 속의 작은 얼굴이어도, 증명 사진 속의 큰 얼굴이어도 인식해야하는 경우
- 이미지 피라미드(Pyramid)
  - 이미지를 여러 스케일(scale)에 걸쳐서 분석하는 가장 기본적인 방법으로 입력 이미지의 크기를 단계적으로 변화(축소)시켜 가면서 필요한 분석 작업을 하는 것이며, 이렇게 생성된 일련의 이미지 집합을 이미지 피라미드라고 함

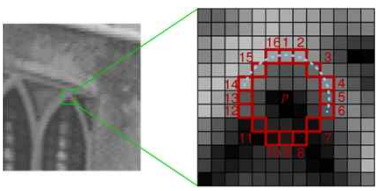


3

FAST

특징만 검출

- Features from Accelerated Segment Test
- 1개의 파라미터를 이용한 **코너** 검출
  1. 한 픽셀 *p* 를 중심으로 하는 3픽셀의 반지름을 가지는 원을 만들고,
  2. 그 원 위의 16개의 픽셀 값을 보고 코너를 찾는다.
  3. *p* 보다 임계값(threshold) 이상 밝거나, 어두운 픽셀들이 *n* 개 이상 연속적으로 존재하면 *p*를 Corner로 판단한다.



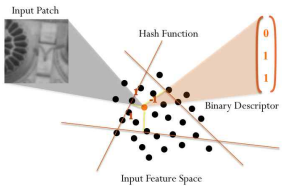
4

BRIEF

- Binary Robust Independent Elementary Features
- **일반적으로 Feature descriptor 저장을 위한 메모리 필요**
  - Descriptor 1개당 SIFT의 경우 512B, SURF의 경우 256B 필요
  - 리소스가 제한된 **모바일 디바이스**에서 구동 불가능
- BREIF descriptor는 이미지 패치 내에서의 픽셀 값 비교를 통해 생성된 바이너리의 나열로 구성
  - 비교 테스트

$$f_n(p) := \sum_{1 \le i \le n} 2^{i-1} \tau(p; x_i, y_i) \qquad \tau(p; x, y) := \begin{cases} 1 & p(x) < p(y) \\ 0 & p(x) \ge p(y) \end{cases}$$

    - $p(x)$는 위치 x에 대한 이미지 p의 픽셀 값
  - ***n*개 비교 테스트**에 대해서 반복하면 특징을 구함



- 비교 테스트를 하기 전에 smoothing을 하는 것 또한 중요
- 장점 : 간결하고 빠름

단점 : 스케일 불변 x, 회전 불변 x

5

ORB

- Oriented FAST and Rotated BRIEF

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y)$$

  1. FAST를 사용하여 keypoint를 검출
  2. Harris 코너 반응값을 사용하여 keypoint들 중 최상위 N개를 추출
  3. 다양한 스케일의 피라미드를 만들어 keypoints 검출(크기 불변 특징)
  4. 특징으로 추출된(코너) 픽셀을 중심으로 윈도우를 형성하고, 중심에 있는 코너로부터 밝기의 중심($C = (m_{01}/m_{00}, m_{10}/m_{00})$)의 방향을 계산하여 코너의 방향성을 구함.
  5. 특징점의 방향을 이미지 패치 내에서의 밝기 모멘트로 간단하게 계산함.

$$\theta = atan2\ (m_{01}, m_{10})$$

특징점의 방향 θ 에 따라 조정된 BRIEF를 적용하여 n차원의 이진 비트스트링을 특징기술자로 계산함.(회전 불변 특징)

$$g_n(p, \theta) = f_n(p) | (x_i, y_i) \in S_\theta$$

단점 : 스케일 불변 x <-??

6

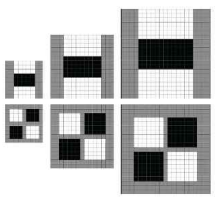
SIFT

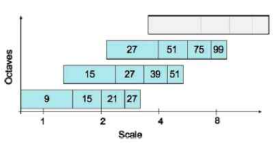
- Scale-Invariant Feature Transform
- **영상의 크기와 회전에 불변하는 특징점을 추출하는 알고리즘**
  1. **Scale-space extreme detection** : 특징의 크기와 위치를 결정
     - 입력 영상에 σ를 증가시켜가며 Gaussian 필터를 적용한 영상을 만든다. σ가 두 배가 될 때마다 영상을 ½로 다운샘플링하고 위의 과정을 반복한다.
     - DoG(Difference of Gaussian) 영상에서 x, y, s축으로 인접한 지점보다 DoG값의 절대값이 큰(극값을 갖는) 지점을 찾아 **특징점 후보로 선택**한다
  2. **Keypoint localization and filtering** : 특징에 좋지 않은 점들 제거
     - 낮은 극값을 갖는 특징점과 낮은 에지 응답을 갖는 특징점 후보 제거
  3. **Orientation assignment** : 방향성분을 결정
  4. **Descriptor construction** :특징을 재표현함
     - 특징점 주변 영역을 4×4 블록으로 나누고, 각 블록 내의 그래디언트 방향의 분포를 8개의 bin을 갖는 히스토그램으로 만든다.
     - SIFT 특징기술자는 **16블록×8bin =128 차원을 갖는 벡터 형태**가 된다.
- **장점 :** 영상 크기, 조명 변화, 평행이동, 회전, 겹침에 강함
- **단점 :** 계산량이 많음

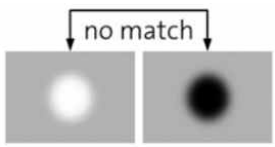
7

SURF

- Speeded Up Robust Features
- **여러 개의 영상으로부터 크기, 조명, 시점 등의 환경변화에 불변하는 특징점을 찾는 알고리즘**
- **SURF의 특징**
  - **Integral image를 이용**하여 특징점의 고속 검출
  - Haar-wavelet 응답을 이용한 특징기술자
  - **Scale-space를 이용**하여 영상의 크기를 줄이는 대신 필터의 크기를 키움으로써 고속화
  - **빠른 매칭** : Laplacian 부호를 비교해서 부호가 같은 경우에만 유클리드 거리를 계산







Octave가 증가하면 필터 크기도 2배 증가

8

특징 기술자 in OpenCV

- `fd_creator = cv2.ORB_create([nfeatures[, WTA_K[, scoreType[, patchSize[, fastThreshold]]]])`

  `features = cv2.ORB.create()`
  - nfeatures는 최대 특징점의 개수로 기본값은 500,
  - WTA_K는 descriptor에서의 임의의 좌표쌍의 개수로 기본값은 2,
  - scoreType은 0이면 HARRIS_SCORE, 1이면 FAST_SCORE로 기본값은0,
  - patchSize는 descriptor의 패치 크기로 기본값은 20임

- `fd_creator = cv2.xfeatures2d.SIFT_create()`
- `fd_creator = cv2.xfeatures2d.SURF_create()`

- `keypoints, descripters = fd_creator.detectAndCompute(image, None)`

9

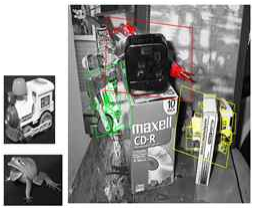
B-2. 특징 매칭

10

특징 매칭(feature matching)

- **특징 매칭**
  - 두 영상 img1과 img2의 특징 벡터를 $kp1_i (i=1,2,...,m)$와 $kp2_j (j=1,2,...,n)$라 표기할 때, 다음을 만족하면 매칭 성공

$$d(kp1_i, kp2_j) < T$$

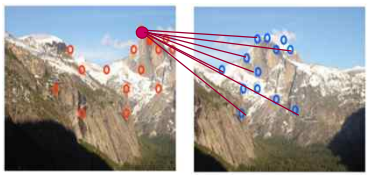


  - 여러 가지 문제를 해결하는 열쇠
    - **물체 인식 또는 증강 현실**
      - 모델 영상은 깨끗한 배경 위에 물체가 놓임
      - 장면 영상은 심한 혼재와 가림이 발생
    - **파노라마 영상 제작**
      - 두 영상이 동등한 입장에서 참여

11

BF 매칭

- 가장 간단한 매칭은 **전수조사 방법(Brute-Force 매칭, BF 매칭)**임
  - 영상 A의 특징기술자(descriptor)를 영상 B의 **모든** 특징기술자(descriptor)와 거리를 계산
  - 장점 : 정확하게 찾을 수는 있으나,
  - 단점 : 영상의 크기가 크게 되면 성능의 문제가 발생 가능
  - **`cv2.BFMatcher(거리계산 방법, crossCheck 유무)`**
    - 보통 `SIFT, SURF`는 `cv2.NORM_L2`, `ORB, BRIEF`는 `cv2.NORM_HAMMING`
    - `crossChecker`가 `true`이면 `BF` 매칭을 하여 가장 일치하는 부분만 `return`



NORM_L2

$$dist(\vec{a}, \vec{b}) = \left[\sum_i (a_i - b_i)^2\right]^{1/2}$$

NORM_HAMMING

$$dist(\vec{a}, \vec{b}) = \sum_i (a_i = = b_i) ? 1 : 0$$

12

FLANN 매칭

- **FLANN(Fast Library for Approximate Nearest Neighbors)**
  - 큰 이미지에서 BF 매칭보다 빠르게 동작
  - 모든 특징기술자(descriptor)와 거리를 계산하는 대신, 현재 **주위의 이웃**에 있는 특징기술자(descriptor)에 대해서만 거리를 계산
  - **cv2.FlannBasedMatcher**(index_params, search_params)
    - **In SIFT & SURF**
      - index_params = **dict**(algorithm = FLANN_INDEX_KDTREE, trees = 5)
      - search_params = **dict**(checks = 50)
    - **In ORB**
      - index_params = **dict**(algorithm = FLANN_INDEX_LSH, number=6, key_size=12, multi_probe_level=1)
      - search_params = **dict**(checks = 100)

dict 함수는
새 딕셔너리를 만드는 Python 함수
{ algorithm : FLANN_INDEX_KDTREE,
trees : 5 }

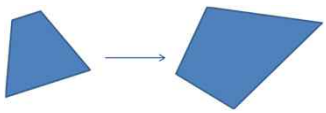
check는 index tree를
탐색해야 하는 횟수.
값이 클수록 탐색시간이
오래걸리지만, 보다 정확하게
탐색함.

5 kd-tree와 50 check가
비교적 정확한 결과 도출

13

Homography(Projective transform)

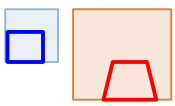
- Homography
  - 2D 평면에서 임의의 사각형을 임의의 사각형으로 매핑시킬 수 있는 변환



Homograpy (Projective Transformation)
어떤 평면이 서로 다른 카메라 위치에 대해 이미지 A와 이미지 B로 투영되었다면, 이미지 A와 이미지 B의 관계를 homography로 표현할 수 있다.

  - Input : *n* unordered images

① **Feature extraction** with SIFT features from all images
② **Matching correspondences between images** with Flann based matcher
③ **Compute Homography** with these matching points to estimate a relative orientation transform within the two images
   - **cv2.findHomography(src_pts, dst_pts, 계산 방법, 임계치)**
     - Homography 행렬 계산
     - 계산 방법
       - 0 - a regular method using all the points
       - **CV_RANSAC** - RANSAC-based robust method
       - **CV_LMEDS** - Least-Median robust method



④ **Homography 행렬을 이용한 좌표 변환 :**
   - **cv2.perspectiveTransform(temp_points, H)**

14

특징 매칭 in OpenCV

- 매칭 그리기
  - **cv2.drawMatches(img1, keypoints1, img2, keypoints2, matches1to2, outImg) ➔ outimg**
- DescriptorMatcher
  - Descriptor 매칭을 취한 추상 기반 클래스
    - **cv2.DescriptorMatcher.match(descriptor1, descriptor2)) ➔ matches**
      - matches.distance : 두 descriptor 사이의 거리
      - matches.trainIdx : descriptor1의 인덱스
      - matches.queryIdx : descriptor2의 인덱스
      - matches.imgIdx : descriptor1의 영상에서의 인덱스
    - **cv2.DescriptorMatcher.knnMatch(descriptor1, descriptor2, k)) ➔ matches**
      - k개의 매치
    - **cv2.DescriptorMatcher.radiusMatch(descriptor1, descriptor2, maxDistance) ➔ matches**
      - 거리가 maxDistance 이내인 거리의 매치

15

예제 9-13 : ORB + 특징 매칭

```
# Python으로 배우는 OpenCV 프로그래밍

src1 = cv2.imread('./data/book1.jpg') # 'cup1.jpg'
src2 = cv2.imread('./data/book2.jpg') # 'cup2.jpg'
img1= cv2.cvtColor(src1,cv2.COLOR_BGR2GRAY)
img2= cv2.cvtColor(src2,cv2.COLOR_BGR2GRAY)

orbF   = cv2.ORB_create(nfeatures=1000) # ①
kp1, des1 = orbF.detectAndCompute(img1, None)
kp2, des2 = orbF.detectAndCompute(img2, None)

bf = cv2.BFMatcher_create(cv2.NORM_HAMMING, crossCheck=True) # ②
matches = bf.match(des1,des2)

matches = sorted(matches, key = lambda m: m.distance) # 정렬
print('len(matches)=', len(matches))
for i, m in enumerate(matches[:3]):
        print('matches[{}]=(queryIdx:{}, trainIdx:{}, distance:{})'.format(
            i, m. queryIdx, m.trainIdx, m.distance))

dst = cv2.drawMatches(img1,kp1,img2,kp2,matches,None,flags=0)
cv2.imshow('dst',  dst)
```

16

예제 9-14 : SIFT, SURF + 특징 매칭

```
# Python으로 배우는 OpenCV 프로그래밍

src1 = cv2.imread('./data/book1.jpg')
src2 = cv2.imread('./data/book2.jpg')
img1= cv2.cvtColor(src1,cv2.COLOR_BGR2GRAY)
img2= cv2.cvtColor(src2,cv2.COLOR_BGR2GRAY)

siftF = cv2.xfeatures2d.SIFT_create()
kp1, des1 = siftF.detectAndCompute(img1, None)
kp2, des2 = siftF.detectAndCompute(img2, None)

# SURF
##surF = cv2.xfeatures2d.SURF_create()
##kp1, des1 = surF.detectAndCompute(img1, None)
##kp2, des2 = surF.detectAndCompute(img2, None)

flan = cv2.FlannBasedMatcher_create()
matches = flan.knnMatch(des1,des2, k=2)

print('len(matches)=', len(matches))
for i, m in enumerate(matches[:3]):
    for j, n in enumerate(m):
        print('matches[{}][{}]=(queryIdx:{}, trainIdx:{},
         distance:{})'.format(i, j, n. queryIdx, n.trainIdx, n.distance))
dst = cv2.drawMatchesKnn(img1,kp1,img2,kp2,matches,None,flags=0)
cv2.imshow('dst',  dst)
```

17

예제 9-13 : 특징 매칭 + Perspective

```
# Python으로 배우는 OpenCV 프로그래밍

src1_pts = np.float32([ kp1[m.queryIdx].pt for m in good_matches])
src2_pts = np.float32([ kp2[m.trainIdx].pt for m in good_matches])

H, mask = cv2.findHomography(src1_pts, src2_pts, cv2.RANSAC, 3.0)#cv2.LMEDS
mask_matches = mask.ravel().tolist() # list(mask.flatten())


h,w = img1.shape
pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
pts2 = cv2.perspectiveTransform(pts, H)
src2 = cv2.polylines(src2,[np.int32(pts2)],True,(255,0, 0),2)

draw_params=dict(matchColor = (0,255,0), singlePointColor = None,
                 matchesMask = mask_matches,  flags = 2)
dst2 = cv2.drawMatches(src1,kp1,src2,kp2, good_matches, None,**draw_params)
cv2.imshow('dst2',  dst2)
```

18

영상 스티칭(Stitching)

- **스티칭(stitching)**
  - 여러 영상에서 추출한 특징점의 기술자(descriptor)를 생성하고, 특징점들간의 정합 과정을 통해 하나의 영상으로 만드는 것
  - **스티치 객체 생성**
    - **cv2.createStitcher() ➔ retval**
  - **영상 스티칭**
    - **cv2.Stitcher.stitch(images[, pano]) ➔ retval, pano**

19

예제 9-19 / 20 : 영상 스티칭

```
path = '.\\images\\stitch_images\\'
images = []

for root, directories, files in os.walk(path):

    for file in files:
        if '.jpg' in file:
            img_input = cv2.imread(os.path.join(root, file))
            images.append(img_input)

stitcher = cv2.Stitcher.create() #cv2.Stitcher_PANORAMA


# Python으로 배우는 OpenCV 프로그래밍

status, dst = stitcher.stitch(images)

if status != cv2.Stitcher_OK:
    print("Can't stitch images, error code = %d" % status)
    exit(-1)

cv2.imshow('dst', dst)
cv2.imwrite('stitch_out.jpg', dst)
```

20

예제 9-19 / 20 : 영상 스티칭

```
# Python으로 배우는 OpenCV 프로그래밍

src1 = cv2.imread('images/stitch_image1.jpg')
src2 = cv2.imread('images/stitch_image2.jpg')
src3 = cv2.imread('images/stitch_image3.jpg')
src4 = cv2.imread('images/stitch_image4.jpg')

stitcher = cv2.Stitcher.create()

# 9-20
status, dst2 = stitcher.stitch((src1, src2))
status, dst3 = stitcher.stitch((dst2, src3))
status, dst4 = stitcher.stitch((dst3, src4))

cv2.imshow('dst2',  dst2)
cv2.imshow('dst3',  dst3)
cv2.imshow('dst4',  dst4)
```

21

예제 9-21 : 비디오 스티칭

```
cap = cv2.VideoCapture('./data/stitch_videoInput.mp4')
t = 0
images = []
STEP = 20
while True:
    t += 1
    retval, frame = cap.read()
    if not retval:
        break
    img = cv2.resize(frame, dsize=(640, 480))
    img = cv2.rotate(img, cv2.ROTATE_90_CLOCKWISE)
    if t%STEP == 0:    # STEP 번째 프레임
        images.append(img)

    cv2.imshow('img',img)
    key = cv2.waitKey(25)

stitcher = cv2.Stitcher.create()
status, dst = stitcher.stitch(images)
if status == cv2.STITCHER_OK:
    cv2.imwrite('./data/video_stitch_out.jpg', dst)
    cv2.imshow('dst',dst)
    cv2.waitKey()
```

22