객체 검출을 위한 방법 :Features2D 와 Homography

2317038

허지원

1. 서론

CNN 기반의 딥러닝이 대중화되기 이전의 컴퓨터 비전 분야에서는 객체 검출을 위해 두 이미지의 특징점을 추출하고, Homography를 계산하는 방식을 활용하였다. 대표적인 알고리즘으로는 SURF와 SIFT가 있다.

A. SURF(Speeded-Up Robust Features)

: 객체 인식 및 특징점 매칭에 사용되는 알고리즘으로, 핵심 아이디어는 SIFT의 정확도를 유지하면서 계산의 효율성을 개선하여 속도를 높이는 데에 있다.

- **SURF 알고리즘의 특징**

1) Hessian 행렬 기반의 특징점 검출
:Blob 구조의 특징점 탐지에 최적화된 기술

$$H(p,\sigma) = \begin{bmatrix} L_{xx}(p,\sigma) & L_{xy}(p,\sigma) \\ L_{xy}(p,\sigma) & L_{yy}(p,\sigma) \end{bmatrix}$$

- L_{xx}, L_{yy}, L_{xy}: 가우시안 2차 미분의 컨볼루션 결과

$$|H(p,\sigma)| = L_{xx}(p,\sigma)L_{yy}(p,\sigma) - L_{xy}(p,\sigma)^2$$

- $|H(p,\sigma)|$이 임계 값을 넘는 위치를 특징점으로 검출

2) 적분 이미지(Integral Image)를 통한 연산 가속화
- 적분 이미지(Integral Image): 이미지의 grayscale 픽셀 값의 누적 값
- 필터링의 연산 시간을 단축시킴

$$S = (A - B - C + D)$$

- A, B, C, D: 영역 모서리 포인트의 적분 이미지 값

- **SURF vs SIFT 비교**

특성	SURF	SIFT
속도	3배	1배 (기준)
조명 변화(luminance)	상대적 취약	우수
시점 변화 (viewpoint changes)	제한적	우수
노이즈 민감도(noise)	낮음	높음

B. 호모그래피(Homography)

: 서로 다른 시점에 있는 두 평면 이미지 사이의 변환 관계를 나타내는 3X3 행렬이다.

- 동차 좌표계를 사용하여 9개 요소 중 8개 요소가 독립적임 (스케일 불변성)
- 이동, 회전, 스케일, 전단, 원근의 기하학적 효과를 가짐

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = H \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \qquad H = \begin{bmatrix} h_{11} h_{12} h_{13} \\ h_{21} h_{22} h_{23} \\ h_{31} h_{32} h_{33} \end{bmatrix}$$

$$u' = \frac{h_{11}u + h_{12}v + h_{13}}{h_{31}u + h_{32}v + h_{33}}, \qquad v' = \frac{h_{21}u + h_{22}v + h_{23}}{h_{31}u + h_{32}v + h_{33}}$$

- (u, v): 원본 평면의 점
- (u', v'): 변환된 평면의 점
- 호모그래피 행렬 H: 3X3 행렬 (8개의 독립적 파라미터)

▫ Homography 계산 방식

A. 대응점 기반 계산: 최소 4쌍의 대응점이 필요 (8개의 방정식 생성)

$$\begin{cases} u_i'(h_{31}u_i + h_{32}v_i + h_{33}) = h_{11}u_i + h_{12}v_i + h_{13} \\ v_i'(h_{31}u_i + h_{32}v_i + h_{33}) = h_{21}u_i + h_{22}v_i + h_{23} \end{cases}$$

B. RANSAC을 이용한 노이즈 처리

2. 코드 흐름

1) 헤더와 네임스페이스 정의

```
#include <stdio.h>
#include <iostream>
#include "opencv2/core.hpp"
#include "opencv2/imgproc.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/calib3d.hpp"
#include "opencv2/xfeatures2d.hpp"

using namespace cv;
using namespace cv::xfeatures2d;
```

A. 헤더 정의: OpenCV에서 이미지 처리, 특징점 추출, 매칭, 시각화 등을 사용하기 위한 헤더를 불러옴

B. 네임스페이스 정의: OpenCV의 네임스페이스를 사용

2) main 함수

```
void readme();

/* @function main */
int main( int argc, char** argv )
{
  if( argc != 3 )
  { readme(); return -1; }
```

A. 프로그램 인자(argc, argv)
   i. argc: command 라인에서 전달된 인자의 총 개수
      ex) ./program image1.jpg image2.jpg → argc = 3 (프로그램 이름 포함)
   ii. argv: 인자 값이 저장된 문자열 배열
      argv: 실행 파일 경로 (ex: "./program")
      argv[1], argv[2]: 사용자가 입력한 이미지 경로
B. 인자 검증: 프로그램 이름(argv), object 이미지(argv[1]), scene 이미지(argv[2])의 3개 인자가 필요

3) 이미지 로드

```
Mat img_object = imread( argv[1], IMREAD_GRAYSCALE );
Mat img_scene = imread( argv[2], IMREAD_GRAYSCALE );

if( !img_object.data || !img_scene.data )
{ std::cout<< " --(!) Error reading images " << std::endl; return -1; }
```

A. 이미지 로드: argv[1], argv[2]를 imread 함수에 전달하여 실제 이미지 파일을 grayscale로 로드
B. 에러 핸들링

4) SURF를 이용한 특징점 검출 및 descriptor 계산

```
//-- Step 1: Detect the keypoints and extract descriptors using SURF
int minHessian = 400;

Ptr<SURF> detector = SURF::create( minHessian );

std::vector<KeyPoint> keypoints_object, keypoints_scene;
Mat descriptors_object, descriptors_scene;

detector->detectAndCompute( img_object, Mat(), keypoints_object, descriptors_object );
detector->detectAndCompute( img_scene, Mat(), keypoints_scene, descriptors_scene );
```

A. minHessian 초기화: 특징점 필터링 임계값인 minHessian을 초기화하여 특징점 검출 민감도를 제어
   i. 값이 높을수록 강한 특징점만 선택 → 검출 수는 적어지고 정확도는 향상됨
B. SURF 객체 생성: SURF::create를 통해 검출기 생성
C. detectAndCompute 함수 호출: 이미지에서 특징점을 검출하고 descriptor 계산
   i. 파라미터
      - img_object: 입력 이미지
      - Mat(): 이미지의 전체 영역을 처리
      - keypoints: 검출된 특징점을 저장하는 벡터
      - descriptors: 계산된 descriptor를 저장하는 배열

5) FLANN 기반 매칭

```
//-- Step 2: Matching descriptor vectors using FLANN matcher
FlannBasedMatcher matcher;
std::vector< DMatch > matches;
matcher.match( descriptors_object, descriptors_scene, matches );
```

A. FLANN(Fast Library for Approximate Nearest Neighbors): 대용량 데이터에서 빠른 근사 최근접 이웃 검색을 위한 라이브러리
B. matcher.match: 모든 object 이미지에 대해 scene 이미지와 유사한 것을 탐색

6) 좋은 매칭 선별

```
double max_dist = 0; double min_dist = 100;

//-- Quick calculation of max and min distances between keypoints
for( int i = 0; i < descriptors_object.rows; i++ )
{ double dist = matches[i].distance;
  if( dist < min_dist ) min_dist = dist;
  if( dist > max_dist ) max_dist = dist;
}

printf("-- Max dist : %f \n", max_dist );
printf("-- Min dist : %f \n", min_dist );

//-- Draw only "good" matches (i.e. whose distance is less than 3*min_dist )
std::vector< DMatch > good_matches;

for( int i = 0; i < descriptors_object.rows; i++ )
{ if( matches[i].distance <= 3*min_dist )
   { good_matches.push_back( matches[i]); }
}
```

A. max_dist, min_dist 초기화: 최대/최소 거리를 초기화
B. 최대/최소값 탐색: for문을 활용하여 모든 쌍의 거리에서의 최대/최소값 탐색
C. 좋은 매칭 선별: for문을 활용하여 모든 쌍에서의 값이 최소값의 3배 이하인 쌍만 좋은 매칭으로 선별

7) 매칭 결과 시각화

```
Mat img_matches;
drawMatches( img_object, keypoints_object, img_scene, keypoints_scene,
             good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
             std::vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );
```

A. drawMatches 함수 호출: 두 이미지의 특징점과 매칭 결과를 한 이미지에 출력
B. 결과 저장: img_matches에 결과를 저장

8) Homograpy 계산 및 객체의 위치 추정

```
//-- Localize the object
std::vector<Point2f> obj;
std::vector<Point2f> scene;

for( size_t i = 0; i < good_matches.size(); i++ )
{
  //-- Get the keypoints from the good matches
  obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
  scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
}

Mat H = findHomography( obj, scene, RANSAC );

//-- Get the corners from the image_1 ( the object to be "detected" )
std::vector<Point2f> obj_corners(4);
obj_corners[0] = cvPoint(0,0); obj_corners[1] = cvPoint( img_object.cols, 0 );
obj_corners[2] = cvPoint( img_object.cols, img_object.rows ); obj_corners[3] = cvPoint( 0, img_object.rows );
std::vector<Point2f> scene_corners(4);

perspectiveTransform( obj_corners, scene_corners, H);
```

A. 매칭 특징점 좌표 추출: good_matches에서 object 이미지와 scene 이미지의 매칭된 특징점 좌표를 각각 추출
B. findHomography 함수 호출: 두 점 집합(obj, scene) 사이의 homography를 계산
   i. 함수 파라미터
      - RANSAC: 이상치를 자동으로 제거하며 계산
C. 객체 코너 좌표 정의: 객체 이미지의 네 꼭짓점 좌표를 scene 이미지 상의 좌표로 변환

9) 결과 표시

```
//-- Draw lines between the corners (the mapped object in the scene - image_2 )
line( img_matches, scene_corners[0] + Point2f( img_object.cols, 0), scene_corners[1] + Point2f( img_object.cols, 0),
      Scalar(0, 255, 0), 4 );
line( img_matches, scene_corners[1] + Point2f( img_object.cols, 0), scene_corners[2] + Point2f( img_object.cols, 0),
      Scalar( 0, 255, 0), 4 );
line( img_matches, scene_corners[2] + Point2f( img_object.cols, 0), scene_corners[3] + Point2f( img_object.cols, 0),
      Scalar( 0, 255, 0), 4 );
line( img_matches, scene_corners[3] + Point2f( img_object.cols, 0), scene_corners[0] + Point2f( img_object.cols, 0),
      Scalar( 0, 255, 0), 4 );

//-- Show detected matches
imshow( "Good Matches & Object detection", img_matches );

waitKey(0);
return 0;
}
```

A. 변환된 코너에 사각형 표시: 호모그래피로 변환된 객체의 네 꼭짓점을 연결해 사각형 표시 및 선 연결
B. 이미지 표시

10) 실행 결과

A. minHessian 값에 따른 결과
   i. 값이 커질 때(ex: 1000)
      - 특징점 개수 감소: 강한 특징점만 검출하여 정확도가 올라가지만, 검출률은 줄어든다.
      - 변화에 따른 실패율 증가: 크기/회전의 변화가 큰 경우에는 검출이 실패할 가능성이 높아진다.
      - 속도 향상: 처리할 특징점의 수가 감소하여 매칭 속도가 빨라진다.
   ii. 값이 작아질 때(ex: 100)
      - 특징점 개수 증가: 약한 특징점도 검출되어 잘못된 검출을 할

가능성이 높아지고, 정확도가 낮아진다.
- 변화에 따른 실패율 감소: 다양한 특징점을 확보하기 때문에 일부 환경에서 검출률이 높아진다.
- 속도 저하

B. 좋은 매칭 기준에 따른 결과 (3*min_dist)

i. 기준 완화(ex: 5*min_dist)
- 매칭 수 증가: 더 많은 매칭을 포함시키지만 호모그래피의 계산에 대한 정확도는 낮아진다.
- 노이즈 증가: RANSAC이 처리해야 할 노이즈가 많아진다.

ii. 기준 강화(ex: 2*min_dist)
- 매칭 수 감소: 정확한 매칭만 선택하므로 호모그래피 정확도가 높아진다.
- 검출 실패의 위험: 유효한 매칭이 4개 미만일 경우 호모그래피 계산이 불가능하다.

3. 코드 분석

➢ **구현 방식**

1) 특징점 검출 (SURF): 객체/장면 이미지에서 회전 및 스케일 변화에 강인한 특징점을 추출한다.
2) 디스크립터 매칭 (FLANN): 두 이미지의 특징점을 매칭하여 유사도가 높은 쌍을 선택한다.
3) 기하학적 검증 (Homography + RANSAC): 매칭된 점들을 기반으로 객체의 위치를 추정한다.
4) 시각화: 매칭 결과와 객체 위치를 이미지에 표시

➢ **장점**

- SURF: SIFT와 비교하여 더 빠르고 조명 변화에 강인하다
- RANSAC: 노이즈를 효과적으로 제거하여 정확한 호모그래피를 계산한다
- FLANN: 대용량의 고차원 데이터에서 효율적으로 근사 검색을 한다

➢ **한계**

- 호모그래피는 3D의 객체에서는 검출이 불가능하다

4. 참고문헌

OpenCV 코드
https://docs.opencv.org/3.4.1/d7/dff/tutorial_feature_homography.html