sx2261_hw6

November 18, 2019

1 Homework 6, Motion

In this homework, you will use the brightness constancy constraint to implement the Lucas-Kanade optical flow algorithm, and evaluate it on synthetic data. In addition, we ask some analytical questions, which you can answer in the cells provided.

Due: Nov. 22, 2019, 11:59 PM New York Time

Deliverables: Your submission should include this iPython notebook (titled <uni>.ipynb, run through once before submission, and do not clear the output) and generate a pdf. Please submit both the pdf and notebook (no zipped files).

1.1 Part 1) Filter the frames and calculate their derivatives

a) Before calculating the derivatives of an image, smoothing is often needed. Please explain in your own words why this should be the case. Your answer:

Derivatives are strongly affected by noise because image noise obviously results in pixels that look very different from their neighbors. And appropriate image smoothing will make meighboring pixels look alike, eliminating the effect of noises to some extent.

b) Video Gradients To implement the brightness constancy constraint, we need to calculate spatial and temporal gradients. Implement the functions below to calculate these gradients. The function calculate_derivatives() should return a tuple of 3 matrices: gradient in x, gradient in y, and gradient in time. Remember to use the gaussian filter to smooth the first image before calculating Ix and Iy, and use the box filter for the images to derive It.

[1]:
```
import numpy as np
import math
from scipy import signal

def gaussian2D(sigma=0.5):
    """
    2D gaussian filter
    """
    size = int(math.ceil(sigma * 6))
    if (size % 2 == 0):
        size += 1
    r, c = np.ogrid[-size / 2: size / 2 + 1, -size / 2: size / 2 + 1]
    g = np.exp(-(c * c + r * r) / (2. * sigma ** 2))
    g = g / (g.sum() + 0.000001)
```

```
    return g

def box2D(n):
    """
    2D box filter
    """
    box = np.full((n, n), 1. / (n * n))
    return box


def calculate_derivatives(i1, i2, sigma=0.5, n=3):
    """
    Derive Ix, Iy and It in this function

    To derive the spatial derivative in one image, you need to smooth the image␣
 ↪with gaussian filter,
    and calculate the derivative, signal.convolve2d and np.gradient might be␣
 ↪useful here

    To derive the temporal derivative in two images, you need to filter the␣
 ↪images with box filters,
    and then calculate the difference between the results
    """
    box_fil = box2D(n)
    gauss_fil = gaussian2D(sigma)

    # take the i1 as the changed image(later happened image)
    #step 1
    #smooth the i2 image with gaussian filter
    g_i1 = signal.convolve2d(i1,gauss_fil,mode='same')
    # caculate the gradient of x and y: Ix, Iy
    Ix, Iy = np.gradient(g_i1)


    #step 2: derive the temporal derivative
    #filter the i1 and i2 using box filters
    b_i1 = signal.convolve2d(i1,box_fil,mode='same')
    b_i2 = signal.convolve2d(i2,box_fil,mode='same')

    # the temporal derivatives It
    It = b_i2 - b_i1
    return Ix, Iy, It
```

Once you are done, please run the following code to test if it's correct. The error should be in the magnitude of e-08. We have created some synthetic data. Your code should correctly recover the gradients with a low margin of error.

[2]:
```
def rel_error(x, y):
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

# test the calculate_derivatives function
a = np.arange(50, step=2).reshape((5,5))
b = np.roll(a, 1, axis=1)
ix, iy, it = calculate_derivatives(a, b, 3, 3)
correct_ix = np.array([[ 1.19566094,  1.44638748,  1.60119287,  1.62253849,  1.
 ↪50447539],
                       [ 1.0953402,   1.32258973,  1.4614055,   1.4781469,   1.
 ↪36814814],
                       [ 0.7722809,   0.92753122,  1.01928721,  1.02535579,  0.
 ↪94404567],
                       [ 0.25022598,  0.29355951,  0.31471869,  0.30865011,  0.
 ↪27704506],
                       [-0.04909038, -0.06915144, -0.0875189,  -0.09965607, -0.
 ↪10218035]])
correct_iy = np.array([[ 0.81434768,  0.67021012,  0.31799052, -0.11348914, -0.
 ↪33688675],
                       [ 1.06507422,  0.87297609,  0.40606602, -0.16184788, -0.
 ↪45494985],
                       [ 1.26884674,  1.03627543,  0.47354769, -0.2067465,  -0.
 ↪55688427],
                       [ 1.37557486,  1.1199824,   0.5038906,  -0.23708941, -0.
 ↪61757009],
                       [ 1.3555138,   1.10076814,  0.48863828, -0.24442014, -0.
 ↪62009437]])
correct_it = np.array([[ 1.33333333,  0.88888889, -1.33333333, -1.33333333, -0.
 ↪88888889],
                       [ 2.,          1.33333333, -2.,         -2.,         -1.
 ↪33333333],
                       [ 2.,          1.33333333, -2.,         -2.,         -1.
 ↪33333333],
                       [ 2.,          1.33333333, -2.,         -2.,         -1.
 ↪33333333],
                       [ 1.33333333,  0.88888889, -1.33333333, -1.33333333, -0.
 ↪88888889]])

print('Testing derivatives:')
print('Ix difference: ', rel_error(ix, correct_ix))
print('Iy difference: ', rel_error(iy, correct_iy))
print('It difference: ', rel_error(it, correct_it))
```

```
Testing derivatives:
Ix difference:  3.7486934875982425e-08
Iy difference:  1.1881858448439447e-08
```

```
It difference:  1.2500021866561346e-09
```

1.2 Part 2) Lucas-Kanade Optical Flow

a) Before the implementation, please explain why pseudo inverse is used in calculating the flow matrices Your answer:

Brightness constancy constraint equation with Spatial coherence constraint:

$$(A^T A)d = A^T b$$

the equation is solvable only when $A^T A$ is invertible and well-conditioned. Also,the (Moore-Penrose) pseudo-inverse of the matrix($A^T A$) could satisfy this condition because it calculates the generalized inverse of a matrix using its singular value decomposition (SVD) and including **large singular values**.

b) Implement Lucas-Kanade optical flow. In the function below, implement Lucas-Kanade optical flow using the brightness constancy constraint.

[3]:
```
def optical_flow(I1g, I2g, x, y, window_size, sigma, n):
    """
    use calculate_derivatives to obtain Ix, Iy and It, then use the window size␣
 ↪to crop the derivatives around the image
    location x, y. To calculate the pseudo inverse, you can use the pinv␣
 ↪function included in numpy
    :param i1: the first frame
    :param i2: the second frame
    :param x: location to calculate optical flow
    :param y: location to calculate optical flow
    :param window: size of the window
    :param sigma: smoothing coefficient
    :param n: box filter size
    :return: u, v
    """
    w = int(window_size/2)

    # Calculate f_x, f_y, f_t
    fx, fy, ft = calculate_derivatives(I1g, I2g, sigma, n)
    # the shape of fx
    # the shape of fy
    # the shape of ft
    # step 1 crop the window
    crop_ix = np.ones([window_size,window_size])
    crop_ix[:,:] = fx[x-w:x+w+1,y-w:y+w+1]

    crop_iy = np.ones([window_size,window_size])
    crop_iy[:,:] = fy[x-w:x+w+1,y-w:y+w+1]

    crop_it = np.ones([window_size,window_size])
    crop_it[:,:] = ft[x-w:x+w+1,y-w:y+w+1]
```

```
    # step2 reshape or flatten
    temp_ix = crop_ix.reshape(window_size*window_size,1)
    temp_iy = crop_iy.reshape(window_size*window_size,1)
    A = np.ones([window_size*window_size,2])
    A[:,0] = temp_ix[:,0]
    A[:,1] = temp_iy[:,0]

    b = - crop_it.reshape(window_size*window_size,1)
    # step3 computer u,v
    # A_t_A np.dot(A.transpose,A)
    A_t_A = np.dot(A.transpose(),A)
    A_t_b = np.dot(A.transpose(),b)

    # the pseudo inverse of np.dot(A.transpose,A)
    inv_A_t_A = np.linalg.pinv(A_t_A)
    # the shape: 2*1
    res = np.dot(inv_A_t_A, A_t_b)
    u = res[0][0]
    v = res[1][0]

    return u,v
```

Test Your Implementation Run the code below to test your implementation. You should get a very low margin of error.

[4]:
```
u, v = optical_flow(a, b, 2, 2, 3, 3, 3)

correct_u = 1.4660487320722453
correct_v = -1.84228885348387
print('Testing derivatives:')
print('u difference: ', rel_error(u, correct_u))
print('v difference: ', rel_error(v, correct_v))
```

```
Testing derivatives:
u difference:  7.572893044666172e-17
v difference:  6.026324387327555e-17
```

c) Is the Lucas-Kanade method for optical flow suitable for all motions? What type(s) of motion will not be properly estimated by the Lucas-Kanade method? Your answer:

Lucas_Kanade method is only effective for small motions, considering the derivation of the algorithm including the approximation of Taylor series expansion.

The motions that are much larger than one pixel will be not properly estimated by the method.

[ ]:

[ ]: