Report

November 29, 2019

1 CMSC426, Project03, Shiyuan Duan

1.1 Introduction

In this project we are implementing an image classfier. We are using SIFT algrithm to generate a Kx128 matrix for each image where K is the number of descriptors we are using. Then we collect all descriptors generated by each image and form a NKx128 matrix where N is the number of image. We will use KMeans clustering algrithm to cluster thoes data in 128 dimensional space. The clusters are the visual bags. For each image we will convert the descriptors into a histogram and use the histogram in an SVM model to classify our images.

1.2 SIFT implementation

The most important part of this project is the implementation of SIFT algrithm. Idealy the SIFT algrithem should take an image and return a list of 128 dimension descriptors. SIFT can be break down to several steps and the following report will discuss each step in detail

1.2.1 Creating Scale Space

This is the very first part of SIFT. In this part we create a difference of gaussian pyrimid (In theory we should use laplacian guassians but it is computationally expensive, and DoG is a good approximation of LoG therefore we will be using DoG instead). The idea is that in the gaussian pyrimid thereare 6 octaves and in each octave there are 5 DoG. By constructing the pyrimid we can achieve scale invariant in this algrithm. This is achieved by executing the following code

[ ]:
```
    # creating feature space
    (oct1_h, oct1_w) = np.shape(test_img)
    octave1 = [cv2.GaussianBlur(test_img, (5,5), 2**(i/5)).astype(np.float64)␣
↪for i in range(6)]
    dof1 = [abs(octave1[i+1] - octave1[i])/255 for i in range(5)]

    test_img2 = cv2.resize(test_img,((int(np.shape(test_img)[1]/2), int(np.
↪shape(test_img)[0]/2))))
    octave2 = [cv2.GaussianBlur(test_img2, (5,5), 2**(i/5)).astype(np.float64)␣
↪for i in range(6)]
    dof2 = [abs(octave2[i+1] - octave2[i])/255 for i in range(5)]

    test_img3 = cv2.resize(test_img,((int(np.shape(test_img2)[1]/2), int(np.
↪shape(test_img2)[0]/2))))
```

```
    octave3 = [cv2.GaussianBlur(test_img3, (5,5), 2**(i/5)).astype(np.float64)␣
↪for i in range(6)]
    dof3 = [abs(octave3[i+1] - octave3[i])/255 for i in range(5)]

    test_img4 = cv2.resize(test_img,((int(np.shape(test_img3)[1]/2), int(np.
↪shape(test_img3)[0]/2))))
    octave4 = [cv2.GaussianBlur(test_img4, (5,5), 2**(i/5)).astype(np.float64)␣
↪for i in range(6)]
    dof4 = [abs(octave4[i+1] - octave4[i])/255 for i in range(5)]

    test_img5 = cv2.resize(test_img,((int(np.shape(test_img4)[1]/2), int(np.
↪shape(test_img4)[0]/2))))
    octave5 = [cv2.GaussianBlur(test_img5, (5,5), 2**(i/5)).astype(np.float64)␣
↪for i in range(6)]
    dof5 = [abs(octave5[i+1] - octave5[i])/255 for i in range(5)]
```

1.2.2 Local extrema detection

In each DoG image(except for the top and bottom), we look for the local min/max compared with it's neighboors around it and on s+1 and s-1 DoG. This step is essentially locating the candidate key points. This step can be simple with the help of helper function.

1.2.3 Keypoint localization and thresh holding

In the previous step we have located the candidate key points, but we actually find a lot of them and not all of them are useful to us and not all of them are the real extrema. In this step we will locate the real extrema and get rid of the keypoints that are not useful to us. A detailed code is shown below

[ ]:
```
def localize_kps(imgs, x, y, s):
    dx_kernal = np.array([[0,1/2,0],[0,0,0],[0,-1/2,0]])
    dy_kernal = np.array([[0,0,0],[1/2,0,-1/2],[0,0,0]])
    dxx_kernal = np.array([[0,0,0],[1,-2,1],[0,0,0]])
    dyy_kernal = np.array([[0,1,0],[0,-2,0],[0,1,0]])
    dxy_kernal = np.array([[-1,0,1],[0,0,0],[1,0,-1]])

    dx_img = cv2.filter2D(imgs[s], -1, dx_kernal)
    dy_img = cv2.filter2D(imgs[s], -1, dy_kernal)
    dxx_img = cv2.filter2D(imgs[s], -1, dxx_kernal)
    dxy_img = cv2.filter2D(imgs[s], -1, dxy_kernal)
    dyy_img = cv2.filter2D(imgs[s], -1, dyy_kernal)

    dx_img_prev = cv2.filter2D(imgs[s-1], -1, dx_kernal)
    dx_img_next = cv2.filter2D(imgs[s+1], -1, dx_kernal)

    dy_img_prev = cv2.filter2D(imgs[s-1], -1, dy_kernal)
    dy_img_next = cv2.filter2D(imgs[s+1], -1, dy_kernal)
```

```
    dx = dx_img[y, x]
    dy = dy_img[y, x]
    ds = imgs[s+1][y, x] - imgs[s-1][y,x]
    dxx = dxx_img[y, x]
    dxy = dxy_img[y, x]
    dyy = dyy_img[y, x]
    dxs = (dx_img_prev[y, x] - dx_img_next[y, x])/2
    dys = (dy_img_prev[y, x] - dy_img_next[y, x])/2
    dss = imgs[s+1][y,x] - 2*imgs[s][y,x] + imgs[s-1][y,x]

    J = np.array([dx, dy, ds])
    HD = np.array([ [dxx, dxy, dxs], [dxy, dyy, dys], [dxs, dys, dss]])


    offset = -np.linalg.pinv(HD).dot(J)
    return offset, J, HD[:2,:2], x, y, s
```

1.2.4 Orientation assignment

So far we have successfully located all the keypoints but we have not yet achieved orientation assignment. The idea in this step is that there is a dominant orientation in each keypoints. We determine the dominant key point by selecting a patch around the key point(the patch size is determined by the scale). Then we cast it with gaussian kernel of 1.5σ. Finally we calculate each pixel's orientation and magnitude and put them in a histogram of 36 bins each representing 10 degrees. The bin with highest value is the dominant orientation. In the paper, it is mentioned that any bins exceeding 80% of the highest bin will also be counted as a keypoint with different orientation

1.2.5 Descriptor creation

Finally we can create the descriptor. To make it rotation invariant, I first take a patch of size roughly $16 * sqrt(2)$ and rotate the patch. This is to garentee that after rotating I can see select 16x16 patch and do not loss any information. Then the 16x16 patch is selected and split into 16 sub_regions each with 4x4 dimension. Then it is similar to the orientation assignment step but this time we are using an 8-bin histogram on each subregion. Finally we will get a $4\not{8} = 128$ dimensional matrix. We do this for each key point and we will obtain our Kx128 matrix. A detailed implementation is shown below

[ ]:
```
def get_descriptor(new_kp, D_img):
    [y, x, theta] = new_kp
    x = int(x)
    y = int(y)
    try:
        kp_sub_area = D_img[y-13:y+13,x-13:x+13]
        plt.imshow(kp_sub_area,cmap='gray')
        M = cv2.getRotationMatrix2D((13,13), theta, 1.0)
```

```
        rotated_sub_area = cv2.warpAffine(kp_sub_area, M, np.shape(kp_sub_area))

        dx_kernal = np.array([[0,1/2,0],[0,0,0],[0,-1/2,0]])
        dy_kernal = np.array([[0,0,0],[1/2,0,-1/2],[0,0,0]])
        dx_img = cv2.filter2D(rotated_sub_area, -1, dx_kernal)+1e-15
        dy_img = cv2.filter2D(rotated_sub_area, -1, dy_kernal)+1e-15

        angle_img = np.degrees(np.arctan(dy_img/dx_img))%360

        angle_subarea = angle_img[13-8:13+8, 13-8:13+8]
        descriptor = np.array(subarea_to_descriptor(angle_subarea))
        descriptor = np.ndarray.flatten(descriptor)

        return descriptor
    except Exception as e:
        return np.reshape(np.array([]),(0,128))


def subarea_to_descriptor(sub_area):
    sub_regions = [sub_area[y:y+4, x:x+4] for x in range(0,16,4) for y in␣
 ↪range(0,16,4)]

    descriptor = []
    for sub_region in sub_regions:
        descriptor.append(sub_region_to_hist(sub_region))
    return descriptor


def sub_region_to_hist(sub_region):
    kernel = np.array([[1,3,3,1],[3,9,9,3],[3,9,9,3],[1,3,3,1]])/64
    hist = np.zeros(8, dtype = np.float32)
    for y in range(4):
        for x in range(4):
            hist[int(sub_region[y,x]//45)] += sub_region[y,x]*kernel[y,x]

    hist /= np.linalg.norm(hist)
    hist[hist>0.2] = 0.2
    hist /= np.linalg.norm(hist)

    return hist
```

Finally we can put everything in one function img_to_descriptors(img). This can be found in code.

1.3 Creating bags of visual words

In this step we are creating bags of visual words. This is done by stacking all the descriptors in training image and use kmeans to cluster them. Each cluster should represent a bag of visual word.

This step is trivial with the help of sklearn.cluster.KMean

1.4 Converting image into a histogram

After we have created the bags of visual words, we can convert our image to a histogram. The idea is that the descriptors are clustered and classified into different groups of visual word. For any given image we can calculate the descriptors and classify all the descriptors based on our kmeans algrithm. We construct the histogram by the frequency in each class.

1.5 Training

The training is the final step our this project and it is the most trivial part. We generate a histogram for each training image and label them for a SVM model.

1.6 Result

The results are shown in the last blocks in the code file. It shows the confusion matrix, a sample histogram of an image and a sample bag of visual word.

1.7 Conclusion

As shown in the confusion matrix, we achieved accuarcy of 84% for airplanes and Leopard. However, our model performed poorly on dolphin with accuarcy of 67%. This may because we only used 50 images for training set and 15 images for testing. There's may not be enough training images. Also dolphin maybe hard to classify because dolphin do not have distinctive pattern liek leopard and airplanes. The images are too smooth resulting less number of keypoints.