16-720 Computer Vision A: Homework 1 (Fall 2022)

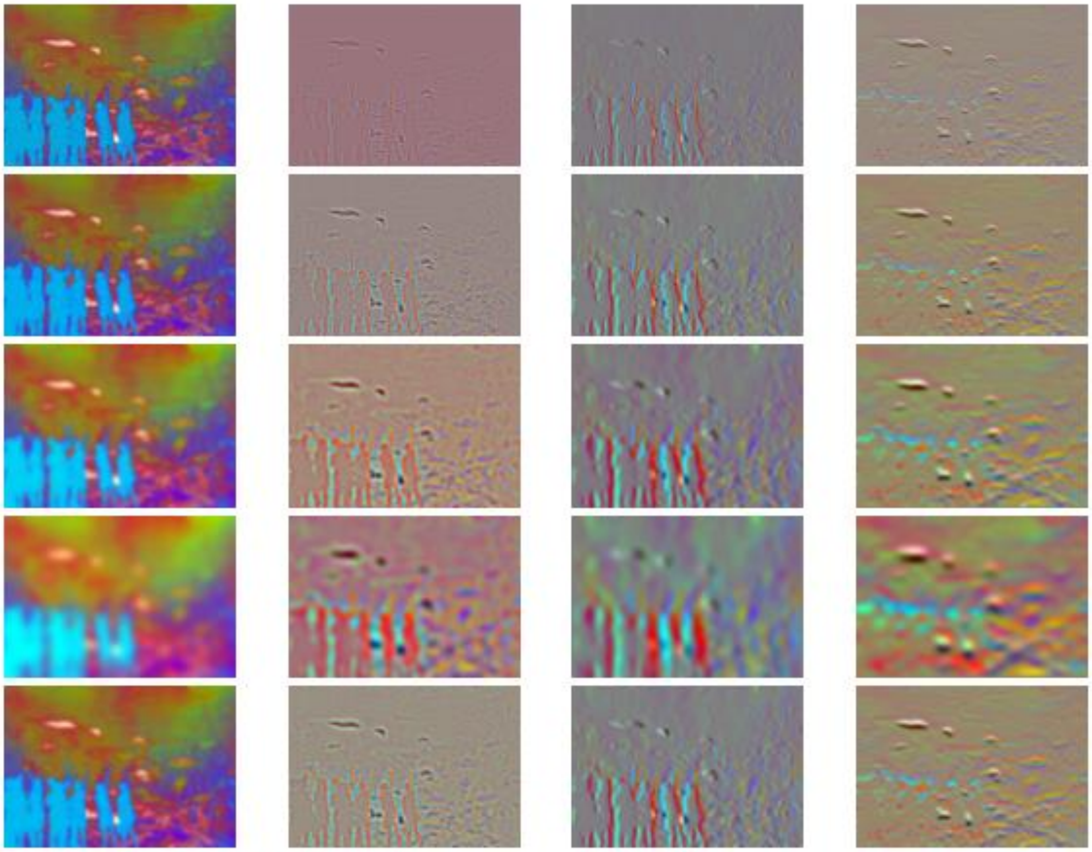
Jiyoon Park (AndrewID: jiyoonp)

Q1.1.1:

We are using four filters. The (1) Gaussian Filter is used to blur the images to take out the noise and small details in the image. The (2) Laplacian Filter is used to detect edges as it detects the fast intensity change occurrence. The (3) Derivative of Gaussian in the direction x detects the vertical edges and the (4) Derivative of Gaussian in the direction y detects the horizontal edges of an image. Since we separate the image into RGB channels, we will detect such features for all RGB. I would put (1)/ (2)(3)(4) into groups since the second group focuses on detecting edges. We use several filter sizes because the features mentioned above come in different sizes. With different sized filters, we can detect the features better.

Q1.1.2

Visualization of filter responses on filter scales = [1, 2, 4, 8, 8**0.5] for image aquarium/sun_aztvjgubyrgvirup.jpg



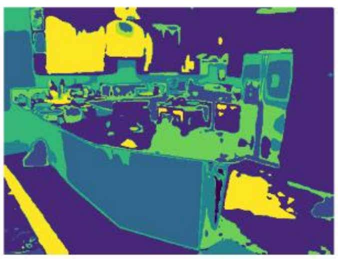
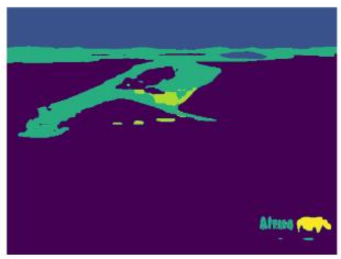
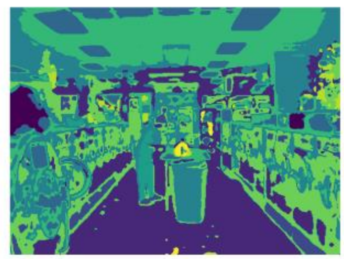
Q 1.2

```
def compute_dictionary(opts, n_worker=4):
    """
    Creates the dictionary of visual words by clustering using k-means.

    [input]
    * opts         : options
    * n_worker     : number of workers to process in parallel

    [saved]
    * dictionary : numpy.ndarray of shape (K,3F)
    """

    data_dir = opts.data_dir
    feat_dir = opts.feat_dir
    out_dir = opts.out_dir
    K = opts.K
    alpha = opts.alpha

    train_files = open(join(data_dir, "train_files.txt")).read().splitlines()

    # ----- TODO -----
    n_train_files = len(train_files)

    # make room for storing temp files
    os.makedirs("../temp/", exist_ok=True)

    args = zip([opts] * n_train_files, [data_dir] * n_train_files, train_files, [alpha] *
n_train_files)

    # multi-processing
    p = multiprocessing.Pool(n_worker * 3)
    p.map(compute_dictionary_one_image, args)

    # load saved data
    filter_responses = []
    for filename in train_files:
        saved = np.load(join("../temp/", filename.split('/')[1].split('.')[0] + '.npz'))
        # print('saved shape:',saved['a'].shape)
        filter_responses.append(saved['a'])
    filter_responses = np.concatenate(filter_responses)  # changed from concatenate
    # print('filter_response shape:', filter_responses.shape)

    kmeans = sklearn.cluster.KMeans(n_clusters=K).fit(filter_responses)
    dictionary = kmeans.cluster_centers_
    # print(dictionary)

    # check for shape
    # print('dictionary shape:',dictionary.shape)

    # example code snippet to save the dictionary
    np.save(join(out_dir, 'dictionary.npy'), dictionary)
    return dictionary

def compute_dictionary_one_image(args):
    """
    Extracts a random subset of filter responses of an image and save it to disk
    This is a worker function called by compute_dictionary

    Your are free to make your own interface based on how you implement compute_dictionary
    """

    # ----- TODO -----
    opts, data_dir, filename, alpha = args

    img = Image.open(join(data_dir, filename))
    img = np.array(img).astype(np.float32) / 255

    # ---- CHANGE ----
    img = img[::3, ::3, :]
```

```
filter_responses = extract_filter_responses(opts, img)
x = np.random.choice(filter_responses.shape[0], alpha)
y = np.random.choice(filter_responses.shape[1], alpha)
cropped = np.array(filter_responses[x, y, :])

np.savez("../temp/" + filename.split('/')[1].split('.')[0] + '.npz', a=cropped)
```

Q1.3

Visualization of wordmaps

Although there is no real way to know exactly how the center of clusters look like, since it took RGB images and made a dictionary, I assume it will label pixels by its visible edges/shadows/colors mostly. And the wordmap seems to have done that. So it does make sense to me.

Q 2.1

```
def get_feature_from_wordmap(opts, wordmap):
    """
    Compute histogram of visual words.

    [input]
    * opts      : options
    * wordmap   : numpy.ndarray of shape (H,W)

    [output]
    * hist: numpy.ndarray of shape (K)
    """

    K = opts.K

    # ----- TODO -----
    hist, bins = np.histogram(wordmap.flatten(), bins=[i for i in range(K + 1)])
    hist = hist / np.sum(hist)

    # plot histogram
    # plt.hist(wordmap.flatten(), bins=[i for i in range(K+1)], density=True)
    # plt.title("histogram")
    # plt.show()
    # print(hist.shape)

    return hist
```

Q 2.2

```
def get_feature_from_wordmap_SPM(opts, wordmap):
    """
    Compute histogram of visual words using spatial pyramid matching.

    [input]
    * opts      : options
    * wordmap   : numpy.ndarray of shape (H,W)

    [output]
    * hist_all: numpy.ndarray of shape K*(4^(L+1) - 1) / 3
    """

    K = opts.K
    L = opts.L

    # ----- TODO -----

    # divide wordmap to parts
    # get histogram for each part, store in part_histogram
    # add the part_histograms
    # normalize
    # give them weights -> add them to final_histogram

    hist_all = np.array([])

    for l in range(L + 1):
        if l <= 1:
            weight = 2 ** (-L)
        else:
            weight = 2 ** (l - L - 1)

        parts = divide_to_parts(wordmap, l)

        for part in parts:
            part_histogram = get_feature_from_wordmap(opts, part)  # array of normalized hist
            w_part_histogram = part_histogram * weight
            hist_all = np.append(hist_all, w_part_histogram)
    hist_all = normalize(hist_all)

    return hist_all
```

Q 2.3

```
def similarity_to_set(word_hist, histograms):
    """
    Compute similarity between a histogram of visual words with all training image histograms.

    [input]
    * word_hist: numpy.ndarray of shape (K)
    * histograms: numpy.ndarray of shape (N,K)

    [output]
    * sim: numpy.ndarray of shape (N)
    """

    # ----- TODO -----
    similarity = np.sum(np.minimum(word_hist, histograms), axis=1)
    return similarity
```

Q 2.4

```
def get_image_feature(opts, img_path, dictionary):
    """
    Extracts the spatial pyramid matching feature.

    [input]
    * opts      : options
    * img_path  : path of image file to read
    * dictionary: numpy.ndarray of shape (K, 3F)


    [output]
    * feature: numpy.ndarray of shape (K) <== ?????
    """

    # ----- TODO -----
    data_dir = opts.data_dir

    img = Image.open(join(data_dir, img_path))
    img = np.array(img).astype(np.float32) / 255

    # ---- CHANGE ----
    img = img[::3, ::3, :]

    wordmap = visual_words.get_visual_words(opts, img, dictionary)
    feat = get_feature_from_wordmap_SPM(opts, wordmap)  # K*(4^(L+1) - 1) / 3
    np.savez("../temp/" + img_path.split('/')[1].split('.')[0] + '.npz', a=feat)

    feature = get_feature_from_wordmap(opts, wordmap)
    return feature


def build_recognition_system(opts, n_worker=4):
    """
    Creates a trained recognition system by generating training features from all training
images.

    [input]
    * opts        : options
    * n_worker  : number of workers to process in parallel

    [saved]
    * features: numpy.ndarray of shape (N,M)
    * labels: numpy.ndarray of shape (N)
    * dictionary: numpy.ndarray of shape (K,3F)
    * SPM_layer_num: number of spatial pyramid layers
    """

    data_dir = opts.data_dir
    out_dir = opts.out_dir
    SPM_layer_num = opts.L
    feat_size = int(opts.K * (pow(4, SPM_layer_num + 1) - 1) / 3)

    train_files = open(join(data_dir, "train_files.txt")).read().splitlines()
    train_labels = np.loadtxt(join(data_dir, "train_labels.txt"), np.int32)
    dictionary = np.load(join(out_dir, "dictionary.npy"))

    # ----- TODO -----
    os.makedirs("../temp/", exist_ok=True)

    n_train_files = len(train_files)
    features = np.zeros([n_train_files, feat_size])

    args = zip([opts] * n_train_files, train_files, [dictionary] * n_train_files)

    # multi-processing
    p = multiprocessing.Pool(n_worker * 2)
    p.starmap(get_image_feature, args)

    for i, filename in enumerate(train_files):
```

```
        saved = np.load(join("../temp/", filename.split('/')[1].split('.')[0] + '.npz'))
        features[i, :] = saved['a']
    # features = np.array(features)
    # features.reshape()
    # print(features.shape)

    # example code snippet to save the learned system
    np.savez_compressed(join(out_dir, 'trained_system.npz'),
                        features=features,
                        labels=train_labels,
                        dictionary=dictionary,
                        SPM_layer_num=SPM_layer_num,
                        )
```

Q 2.5

Tested with default opts:

Accuracy 0.5075

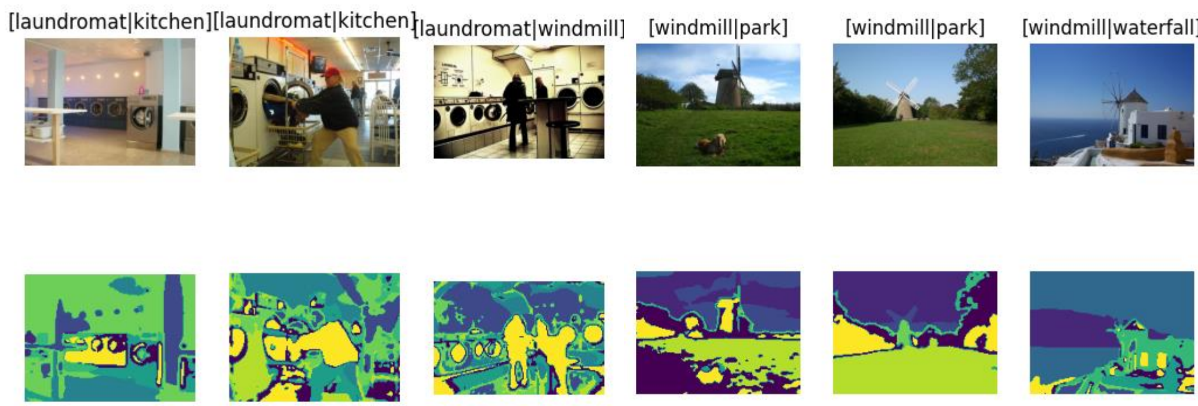
Confusion matrix:

```
[[29.  0.  1.  2.  4.  2.  4.  8.]
 [ 2. 28.  8.  5.  0.  1.  1.  5.]
 [ 2.  5. 28.  0.  1.  6.  1.  7.]
 [ 2.  3.  1. 32. 10.  1.  0.  1.]
 [ 2.  2.  2. 16. 18.  3.  4.  3.]
 [ 3.  2.  5.  2.  3. 27.  4.  4.]
 [ 4.  3.  1.  1. 10.  8. 22.  1.]
 [ 2.  3. 11.  1.  2.  7.  5. 19.]]
```

Time took: 702.4184007644653 sec

Q 2.6

Looking at the confusion matrix above, the Laundromat and Windmill have bad accuracy compared to other images. I think this is because the features that can be detected by the four filters are not very distinct in telling what the scenery is. For the Laundromat, there are too many features in the image. The result shows that it has been labeled as kitchen. Which makes sense as both have a lot of features. A lot of edges, highlights etc. may have been too much information to predict the right label. For the windmill it may have been too less features. It got confused with highway and park which is understandable as they don’t have a lot of distinctive features as well. Below are some images that was mislabeled.



Q 3.1

Intuitively, if we increase the number of clusters, we will be able to make a more precise dictionary, leading to better accuracy. Also, increasing the number of spatial pyramid layers will allow us to understand the spatial structure of the image more. Adding more filter scales will also improve accuracy as it can detect features of different sizes better. Making alpha bigger will also make the accuracy better as the system has more to learn from. *The highlighted is my final accuracy*

Filter Scales	K	Alpha	L	Accuracy
[1, 2]	10	25	1	0.5475
[1, 2]	20	25	1	0.59
[1, 2]	20	50	1	0.5725
[1, 2]	30	25	2	0.65
[1, 2]	100	25	2	0.655
[1, 2, 4, 8]	100	25	2	0.6525

Q 3.2

I increased the speed of the system by changing two things.

First, I resized the image.

(1) I shrank the height and width of the image by 3 using the code

```
# ---- CHANGE ----
img = img[::3, ::3, :]
```

every time I import the image.

(2) Since the image size is smaller, I expected it to be a bit faster, but thought the accuracy would go down.

(3) However, the increase in speed was more than I expected and the accuracy did not go down that significantly. I tested this with a few sets of opts and saw that the speed improved greatly without hurting the accuracy of the system. Even with large values for the opts, the speed was more than 5 times shorter than the original system. (Greyed out is the original system result)

Filter Scales	K	Alpha	L	Accuracy	Time (s)
[1, 2]	10	25	1	0.5475	536
[1, 2]	10	25	1	0.5125	65
[1, 2, 4, 8]	20	50	2	0.5625	76

Second,

(1) Instead of making the wordmap image by going through the for loops and assigning each pixel with a label, I took the image as a whole and compared the distances.

From this,

```
# take image by pixel -> assign dictionary index to each value
for i in range(h):
    for j in range(w):
        pixel_img = filtered_img[i, j, :]
        # print(dictionary.shape, np.array([pixel_img]).shape)
        dist = scipy.spatial.distance.cdist(dictionary, np.array([pixel_img]),
                                            'euclidean')
        picture[i, j] = np.argmin(dist, axis=0)
wordmap = np.array(picture)
```

To this

```
# ---- CHANGE ----
filter_response = filtered_img.reshape(filtered_img.shape[0] * filtered_img.shape[1], -1)
d = scipy.spatial.distance.cdist(filter_response, dictionary)
wordmap = np.argmin(d, axis=1).reshape(filtered_img.shape[0], filtered_img.shape[1])
```

(2) I was not expecting a great decrease in time as the operation is basically doing the same thing. But was hoping it would decrease the time a little since the for loops are gone.

(3) The speed increased greatly. The results of this change is below.

(Greyed out is the original system result)

Filter Scales	K	Alpha	L	Accuracy	Time (s)
[1, 2]	10	25	1	0.5475	536
[1, 2]	10	25	1	0.545	125
[1, 2]	100	25	2	0.63	222
[1, 2, 4, 8]	100	25	3	0.6275	417

Finally,

(1) I applied both resizing and labeling the whole image

(2) I expected it to work well as I checked it worked well independently.

(3) gave me this result: (Greyed out is the original system result)

Filter Scales	K	Alpha	L	Accuracy	Time (s)
[1, 2]	10	25	1	0.5475	536
[1, 2]	100	25	2	0.525	70
[1, 2, 4, 8]	100	50	3	0.6225	101