Homework 1: Spatial Pyramid Matching for Scene Classification

Srividya Gandikota
sgandiko
sgandiko@andrew.cmu.edu

1 Representing the World with Visual Words

1.1 Extracting Filter Responses

1.1.1 What properties do each of the filter functions pick up? (See Fig 3) Try to group the filters into broad categories (e.g. all the Gaussians). Why do we need multiple scales of filter responses?

Gaussian Filters:

When considering a multi-scale filter bank, each filter function has the ability to pick up different properties. The first row depicts a set of various Gaussian filters which tries to remove the higher frequencies and some noise in an image using a low pass filter, which also blurs it.

Gaussian Laplacian:

The next row of filters are Gaussian Laplacian ones that work well at identifying image contours which could be in the form of horizontal or vertical edges. As this utilizes a Gaussian, it also smooths and reduces noise from the derivative function.

Derivative of Gaussian Filters in X Direction:

The next row contains a few derivative of Gaussian filters in the X direction, which have the ability to pick up vertical features, vertical edges, or more specifically changes in the image intensity across the x direction.

Derivative of Gaussian Filters in Y Direction:

The next set of filters in the bank are derivative of Gaussian filters in the Y direction that tend to do the same as the previous ones but have a focus on horizontal aspects. They work well with picking up horizontal features, horizontal edges, or changes in the image intensity across the y direction.

Multiple Scales Usage:

Multiple scales of filter responses are needed so that we can identify different types of features that we may want. For example, if we consider an image with larger edges, we can use a large kernel filters but if the image has a small edge somewhere tiny, we would need to utilize a small kernel filter and search through the entire image to find it.

1.1.2 Apply all 4 filters with at least 3 scales on aquarium/sun aztvjgubyrgvirup.jpg, and visualize the responses as an image collage as shown in Fig 4. The included helper function util.display filter responses (which expects a list of filter responses with those of the Lab channels grouped together with shape M ×N ×3) can help you to create the collage.

Collage of Different Filters:

The image below depicts the collage of image ran against the 4 filters below. In column order, they were ran against a Gaussian, Laplacian Gaussian, Derivative of Gaussian in X axis, and Derivative of Gaussian in Y axis. The rows correspond to the various σ scales, [1 2 4 8].

As we can see, each row is able to pick up different distinct features due to filter characteristics. The first column, Gaussian filter, is able to remove the higher frequencies and blur the image. The column below uses the Laplacian Gaussian to find the areas where there is rapid changes. Next is the Derivative of Gaussian in X axis filter that only picks up the vertical edges, while the last is the Y axis derivative filter that finds horizontal edges.

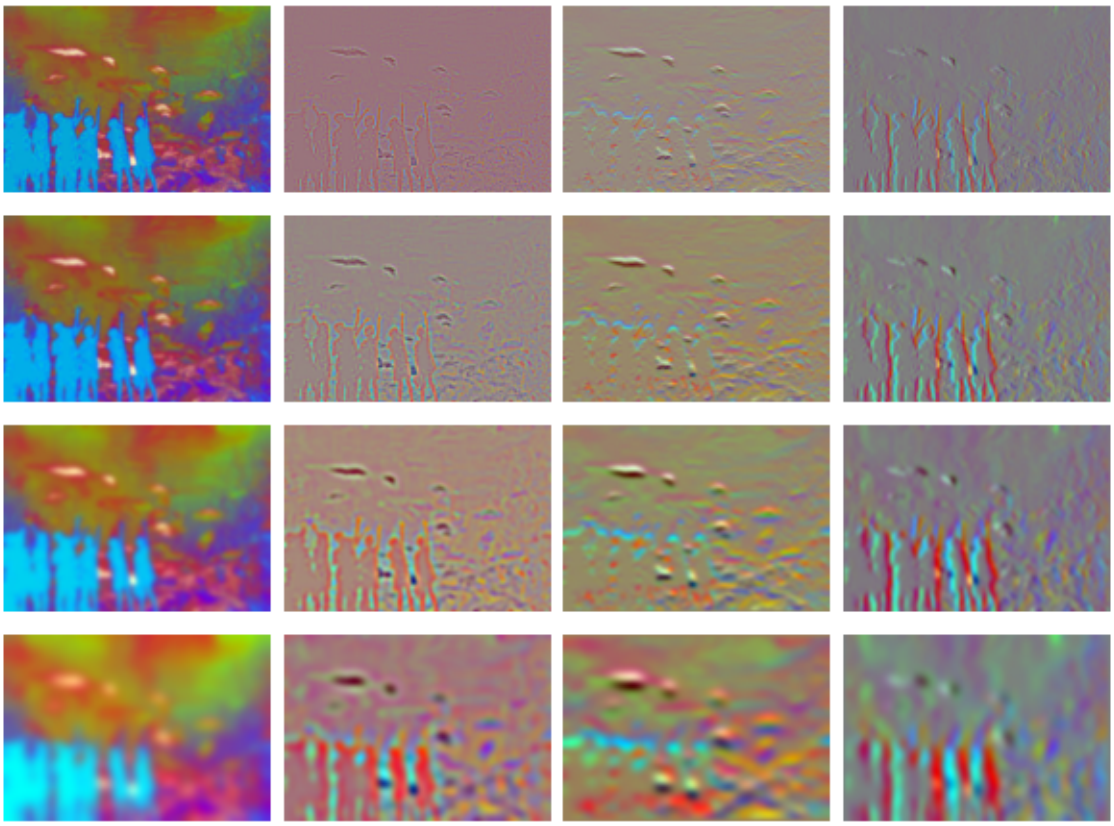


Figure 1: Filter Bank Response on Image

1.2 Creating Visual Words

1.2.1 You will now create a dictionary of visual words from the filter responses using k-means. Then, to generate a visual words dictionary with K words (opts.K), you will cluster the responses with k-means using the function sklearn.cluster.KMeans.

The implementation for this dictionary can be found in Figure 2 and 3 below.

```python
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
    '''
    opts = get_opts()

    #Split args into corresponding variables
    train_files = args[0]
    img_num = args[1]
    alpha = int(args[2])

    #Get image data, help from Q1.1
    img_path = join(opts.data_dir, train_files)
    img = Image.open(img_path)
    img = np.array(img).astype(np.float32) / 255

    #Responses at alpha random pixels
    filter_response = extract_filter_responses(opts, img)
    #https://numpy.org/doc/stable/reference/random/generated/numpy.random.Generator.choice.html
    new_img = filter_response[np.random.choice(filter_response.shape[0], alpha), np.random.choice(filter_response.shape[1], alpha)]
    img_path = join(opts.feat_dir, str(img_num) + '.npy')
    np.save(img_path, new_img)
```

Figure 2: Compute Dictionary One Image Function Implementation

```python
def compute_dictionary(opts, n_worker):
    '''
    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

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

    #Get alpha and image list
    img_list = np.arange(len(train_files))
    alpha_list = np.ones(len(train_files)) * opts.alpha

    #Attempt Multiprocessing https://docs.python.org/3/library/multiprocessing.html
    multi_process = Pool(n_worker)
    args = []
    for t, i, a in zip(train_files, img_list, alpha_list):
        args.append((t, i, a))
    multi_process.map(compute_dictionary_one_image, args)

    #Get image filer responses
    filter_responses = []
    print(len(train_files))
    for i in range(len(train_files)):
        img_path = join(opts.feat_dir, str(i) + '.npy')
        filter_responses.append(np.load(img_path))

    #Implement KMeans
    kmeans = sklearn.cluster.KMeans(n_clusters=K).fit(np.concatenate(filter_responses, axis=0))
    dictionary = kmeans.cluster_centers_
    np.save(join(out_dir, 'dictionary.npy'), dictionary)
```

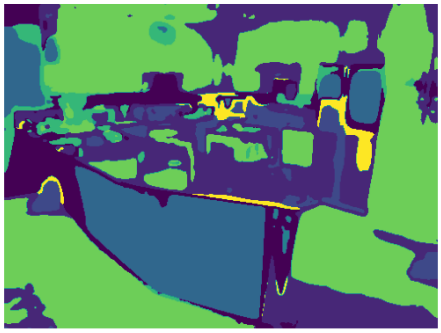
Figure 3: Compute Dictionary Function Implementation

1.3 Computing Visual Words

1.3.1 We want to map each pixel in the image to its closest word in the dictionary. Complete the following function to do this: `visual_words.get_visual_words(opts, img, dictionary)` and return wordmap, a matrix with the same width and height as img, where each pixel in wordmap is assigned the closest visual word of the filter response at the respective pixel in img.

The visualization wordmap of image $kitchen/sun_asmevtpkslccptd.jpg$ is below.

(a) Base Image



(b) Visual Wordmap

The visualization wordmap of image $waterfall/sun_abcxnrzizjgcwkdn.jpg$ is below.

(a) Base Image

(b) Visual Wordmap

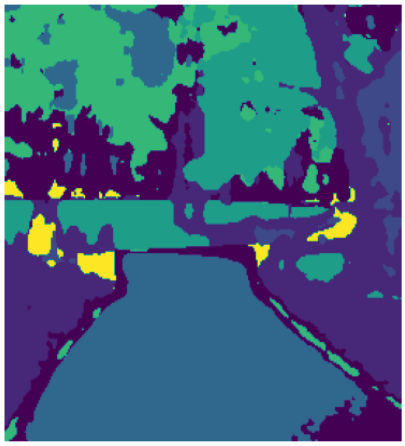
The visualization wordmap of image $highway/sun_bgyforanugtijxll.jpg$ is below.

(a) Base Image

(b) Visual Wordmap

The visualization wordmap of image $park/sun_aitrzglvqgozzaur.jpg$ is below.

(a) Base Image



(b) Visual Wordmap

2 Building a Recognition System

2.1 Extracting Features

2.1.1 Write the function `visual_recog.get_feature_from_wordmap(opts, wordmap)` that extracts the histogram (numpy.histogram()) of visual words within the given image (i.e., the bag of visual words).

The implementation for this histogram extraction can be found in Figure 8 below.

```python
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

    #Get histogram information
    hist, bins = np.histogram(wordmap.flatten(), bins = np.arange(0, K + 1))
    hist_norm = hist / np.sum(hist)

    return hist_norm
```

Figure 8: Get Feature from Wordmap Function Implementation

2.2 Multi-Resolution: Spatial Pyramid Matching

2.2.1 Create the following function that form a multi-resolution representation of the given image: `visual_recog.get_feature_from_wordmap_SPM(opts, wordmap)`. You need to specify the layers of pyramid (L) in opts.L. As output, the function will return hist all, a vector that is L1 normalized.

The implementation for this spatial pyramid matching can be found in Figure 9 below.

```python
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)/3)
    '''

    K = opts.K
    L = opts.L

    #Get dictionary
    dictionary = np.load(join(opts.out_dir, 'dictionary.npy'))

    hist_all = np.array([])
    #Pyramid calculations
    for i in range(L, -1, -1):
        l = pow(2, i)
        #Check weights
        if i > 1: #Non Base Layer
            weight = pow(2, i - L - 1)
        else: #Base Layer
            weight = pow(2, -L)

        #Creates sub matrices using Wordmap division
        level_mapping = []
        for j in np.array_split(wordmap, l, axis=0):
            for k in np.array_split(j, l, axis=1):
                level_mapping.append(k)

        #Calculate and Append all Histograms to an Array
        for total in range(pow(l, 2)):
            curr_hist = get_feature_from_wordmap(opts, level_mapping[total])
            hist_all = np.append(hist_all, curr_hist, axis = 0)

        hist_all = hist_all * weight

    #Normalize Histograms
    if np.sum(hist_all) > 0:
        hist_all = hist_all/np.sum(hist_all)

    return hist_all
```

Figure 9: Get Feature from Wordmap SPM Function Implementation

2.3 Comparing Images

2.3.1 Create the function `visual_recog.distance_to_set(word hist, histograms)` where word hist is a K (4(L+1)-1)/3 vector and histograms is a T × K (4(L+1)-1)/3 matrix containing T features from T training samples concatenated along the rows.

The implementation for this distance to set can be found in Figure 10 below.

```
def distance_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]
    * hist_dist: numpy.ndarray of shape (N)
    '''

    #Calculates Similarity and Distance
    similarity = np.sum(np.minimum(word_hist, histograms), axis=1)
    hist_dist = 1 - similarity

    return hist_dist
```

Figure 10: Distance to Set Function Implementation

2.4 Building a Model of the Visual World

2.4.1 Implement the function `visual_recog.build_recognition_system` that produces `trained_system.npz` . You may include any helper functions you write in `visual_recog.py` Implement `visual_recog.get_image_feature(opts, img_path, dictionary)` that loads an image, extract word map from the image, computes the SPM, and returns the computed feature. Use this function in your `visual_recog.build_recognition_system()`.

The implementation for the get image feature can be found in Figure 11 below.

```python
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*(4^L-1)/3)
    '''

    #Load Image
    img = Image.open(img_path)
    img = np.array(img).astype(np.float32)/255

    #Extract Wordmap
    wordmap = visual_words.get_visual_words(opts, img, dictionary)

    #Compute SPM
    features = get_feature_from_wordmap_SPM(opts, wordmap)
    return features
```

Figure 11: Get Image Feature Function Implementation

The main function did utilize two different helper functions to help multiprocessing run which can be found in Figure 12 below.

```python
def process_image_feature(args):
    img_path, dictionary, opts = args
    feature = get_image_feature(opts, img_path, dictionary)
    return feature

def multiprocess_map_features(n_worker, opts, train_files, dictionary):
    muti_process = Pool(processes=n_worker)
    args_list = [(join(opts.data_dir, f), dictionary, opts) for f in train_files]
    features = muti_process.map(process_image_feature, args_list)
    return features
```

Figure 12: Helper Function Implementations

The main build recognition system function can be found in Figure 13 below.

```python
def build_recognition_system(opts, n_worker=1):
    '''
    Creates a trained recognition system by generating 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

    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'))

    #Calculate features with multiprocessing map
    features = multiprocess_map_features(n_worker, opts, train_files, dictionary)
    '''features = []

    for f in train_files:
        img_path = join(data_dir, f)
        feature = get_image_feature(opts, img_path, dictionary)
        features.append(feature)
    '''
    features = np.vstack(features)

    # 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,
    )
```

Figure 13: Build Recognition System Function Implementations

2.5 Quantitative Evaluation

2.5.1 Implement the function `visual_recog.evaluate_recognition_system()` that tests the system and outputs the confusion matrix. Include the confusion matrix and your overall accuracy in your write-up.

The evaluate recognition system function can be found in Figure 14 below.

```python
def evaluate_recognition_system(opts, n_worker=1):
    '''
    Evaluates the recognition system for all test images and returns the confusion matrix.

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

    [output]
    * conf: numpy.ndarray of shape (8,8)
    * accuracy: accuracy of the evaluated system
    '''

    data_dir = opts.data_dir
    out_dir = opts.out_dir

    trained_system = np.load(join(out_dir, 'trained_system.npz'))
    dictionary = trained_system['dictionary']

    # using the stored options in the trained system instead of opts.py
    test_opts = copy(opts)
    test_opts.K = dictionary.shape[0]
    test_opts.L = trained_system['SPM_layer_num']

    test_files = open(join(data_dir, 'test_files.txt')).read().splitlines()
    test_labels = np.loadtxt(join(data_dir, 'test_labels.txt'), np.int32)
    # ----- TODO -----
    conf = np.zeros((8, 8), dtype=int)

    for i in range(len(test_files)):
        #Open Image
        img_path = join(opts.data_dir, test_files[i])
        img = Image.open(img_path)
        img = np.array(img).astype(np.float32)/255

        #Get words, features, and distance
        wordmap = visual_words.get_visual_words(opts, img, dictionary)
        test_feat = get_feature_from_wordmap_SPM(opts, wordmap)
        hist_dist = distance_to_set(test_feat, trained_system['features'])

        #Calculate confusion matrix
        predicted_label = trained_system['labels'][np.argmin(hist_dist)]
        actual_label = test_labels[i]
        conf[actual_label, predicted_label] += 1

    #Calculate accuracy
    acc = np.trace(conf) / np.sum(conf)

    return conf, acc
```

Figure 14: Helper Function Implementations

The confusion matrix and accuracy can be found below, which was found to be about 54%.

```
[[33  1  2  4  1  2  3  4]
 [ 0 29  5  7  3  1  2  3]
 [ 1  6 29  1  0  0  2 11]
 [ 4  2  2 30  8  2  1  1]
 [ 4  3  1 11 22  5  3  1]
 [ 2  1  6  1  4 30  3  3]
 [ 5  0  2  2  7 10 21  3]
 [ 4  7  6  1  2  5  4 21]]
0.5375
```

Figure 15: Evaluate Recognition Function Implementations

2.6 Find the Failures

2.6.1 In your writeup, list some of these hard classes/samples, and discuss why they are more difficult than the rest.

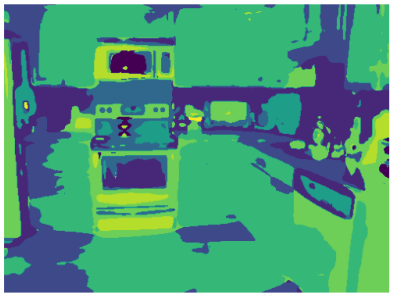
From what I can see, many of the failure points came from the laundromat being misidentified as a kitchen. From my understanding, these struggled because the tan/yellowish colors of the laundromat are very similar to the yellow/wood color of most of the kitchen pictures. Similarly, many of the waterfall files also struggled and got identified as a park. In this case, the waterfall contains a lot of variations of green, blue, and white, while the park photos share the blue and green, leading to possible identification.

The visualization of the similarities between laundromat and kitchen can be seen below.

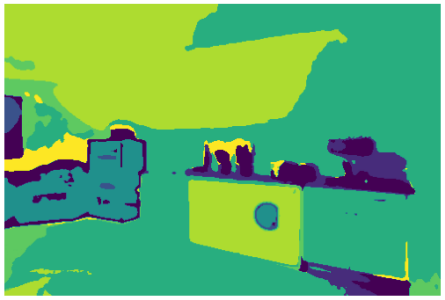
(a) Laundromat

(b) Kitchen

Their corresponding visual word visualization can be seen below.



(a) Laundromat



(b) Kitchen

3 Improving Performance

3.1 Hyperparameter Tuning

3.1.1 Tune the system you build to reach around 65 percent accuracy on the provided test set (data/test files.txt). A list of hyperparameters you should tune is provided below. They can all be found in opts.py. Include a table of ablation study containing at least 3 major steps (changing parameter X to Y achieves accuracy Z percent). Also, describe why you think changing a particular parameter should increase or decrease the overall performance in the table you show.

The steps I took to achieve a higher accuracy are:

- Changing K from 10 to 30 increased my accuracy from 53.8% to 62.3%. Increasing K increased the accuracy as I expected because of a few key reasons. K corresponds to the number of words in the dictionary and clusters in K-Means, meaning that there are more options to classify different objects. This also means that the program can recognize finer details much better.
- Changing L from 1 to 3 increased the accuracy from 62.3% to 65.3%. Increasing L increased the accuracy not as much as I expected. L is the number of layers in our spatial pyramid. This means that if we add more layers, we should theoretically be able to get more information. This leads to better detection of recognition of objects that have distinct spatial designs.
- By increasing alpha from 25 to 50, the accuracy increased from 65.3% to 66.3%. The alpha hyper parameter is in charge of acting as a weight for the SPM function by determining the impact of each layer. This parameter was one that I expected that decreasing it would increase the accuracy but instead the opposite was the case, where we had a slight increase.
- Changing K from 30 to 45 increased my accuracy from 66.3% to 67.5%.