16-720 Computer Vision: Homework 1 (Spring 2022)
Spatial Pyramid Matching for Scene Classification

Instructor: Deva Ramanan
TAs: Gautam Gare, Tarasha Khurana, Neehar Peri

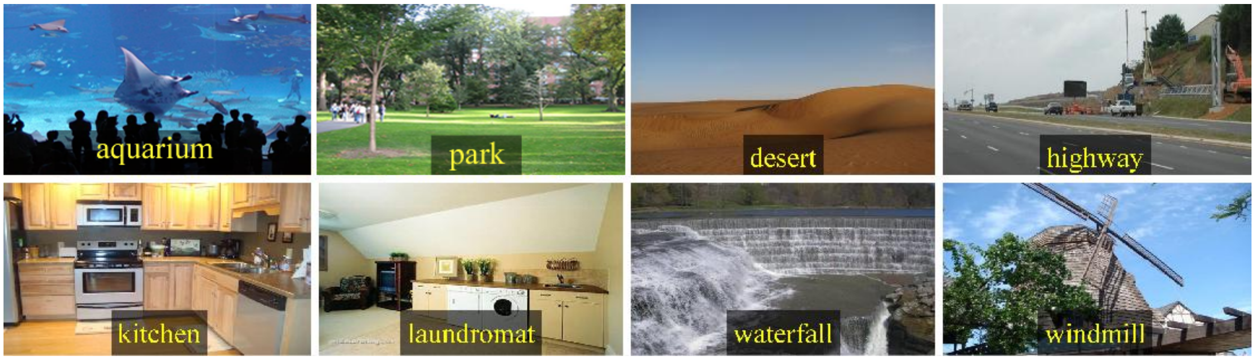

Figure 1: **Scene Classification:** Given an image, can a computer program determine where it was taken? In this homework, you will build a representation based on bags of visual words and use spatial pyramid matching for classifying the scene categories.

Instructions

1. You will submit both a pdf writeup and a zip of your code to Gradescope. See the complete submission checklist at the end to ensure you have everything. Handwritten writeups will not be accepted. You may complete your writeup using LaTeX, Microsoft Word, or in-line on a Jupyter Notebook.

2. Each question (for points) is marked with a **Q**.

3. **Start early!** This homework may take a long time to complete.

4. **Attempt to verify your implementation as you proceed.** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.

5. In your PDF, add a page break after each question. **When submitting to Gradescope, make sure that you select the page corresponding to your answer for each question.** Not doing this will incur a penalty.

6. If you have any questions or need clarifications, please post in Slack or visit the TAs during office hours.

7. The assignment must be completed using Python 3. We recommend setting up a conda environment, but you are free to set up your environment however you like. See the Slack thread

Overview

The bag-of-words (BoW) approach, which you learned about in class, has been applied to a myriad of recognition problems in computer vision. For example, two classic ones are object recognition [5, 7] and scene classification [6, 8][1].

Beyond that, a great deal of study has aimed at improving the BoW representation. You will see a large number of approaches that remain in the spirit of BoW but improve upon the traditional approach which you will implement here. For example, two important extensions are pyramid matching [2, 4] and feature encoding [1].

An illustrative overview of the homework is shown in Fig 2. In Section 1, we will build the visual words from the training set images. With the visual words, *i.e.* the dictionary, in Section 2 we will represent an image as a visual-word vector. Then the comparison between images is realized in the visual-word vector space. Finally, we will build a scene recognition system based on the visual bag-of-words approach to classify a given image into 8 types of scenes.

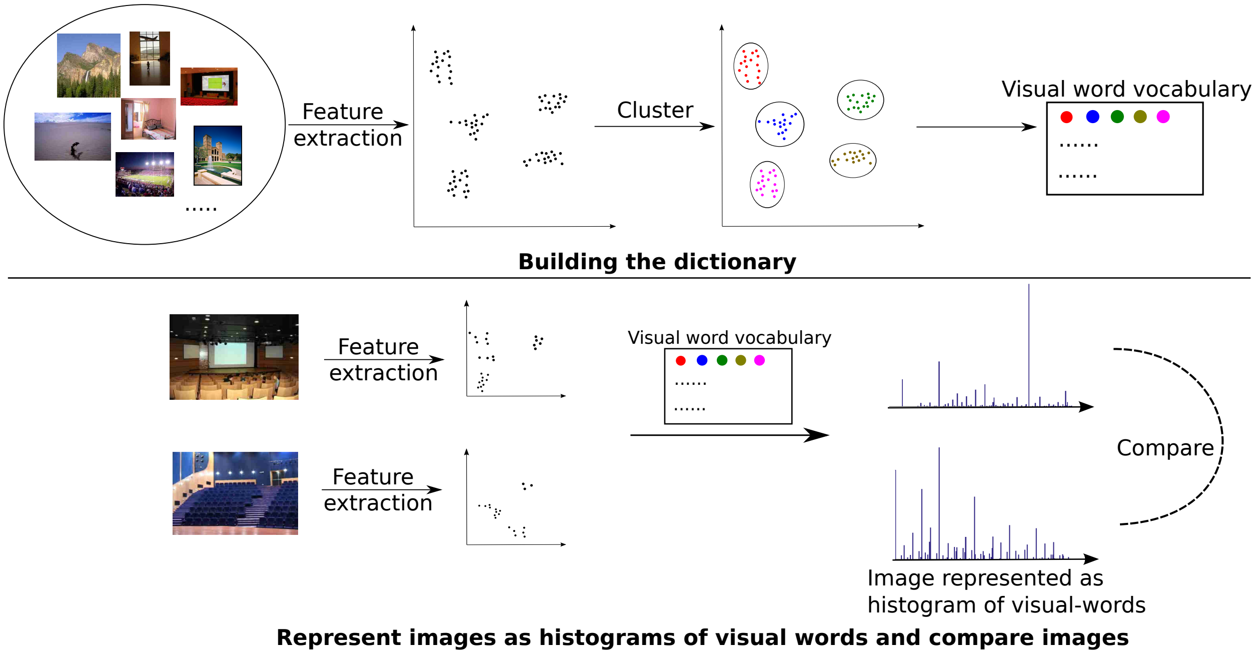

Figure 2: An overview of the bags-of-words approach to be implemented in the homework. First, given the training set of images, we extract the visual features of the images. In our case, we will use the filter responses of the pre-defined filter bank as the visual features. Next, we build visual words, *i.e.* a dictionary, by finding the centers of clusters of the visual features. To classify new images, we first represent each image as a vector of visual words, and then compare new images to old ones in the visual-word vector space – the nearest match provides a label!

What you will be doing: You will implement a scene classification system that uses the bag-of-words approach with its spatial pyramid extension. The paper that introduced the pyramid matching kernel [2] is

> K. Grauman and T. Darrell. *The Pyramid Match Kernel: Discriminative Classification with Sets of Image Features.* ICCV 2005. http://www.cs.utexas.edu/~grauman/papers/grauman_darrell_iccv2005.pdf

Spatial pyramid matching [4] is presented in

> S. Lazebnik, C. Schmid, and J. Ponce, *Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories*, CVPR 2006. http://www.di.ens.fr/willow/pdfs/cvpr06b.pdf

[1] This homework is largely self-contained, but reading the listed papers (or even just skimming them) will likely be helpful.

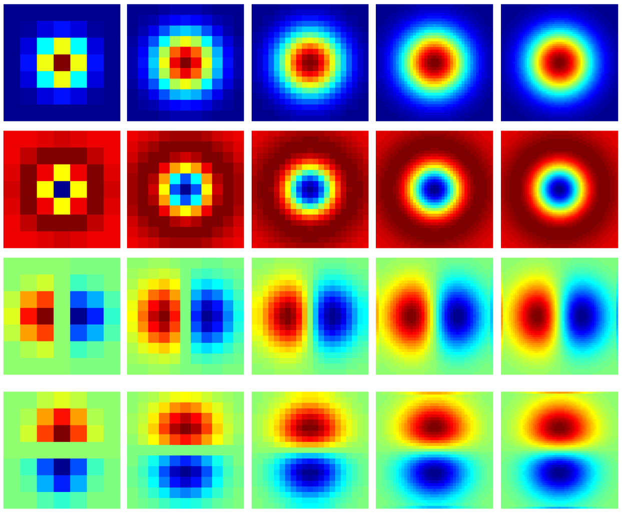

Figure 3: Multi-scale filter bank

You will be working with a subset of the SUN database[2]. The data set contains 1600 images from various scene categories like "aquarium, "desert" and "kitchen". And to build a recognition system, you will:

- take responses of a filter bank on images and build a dictionary of visual words, and then
- learn a model for images based on the bag of words (with spatial pyramid matching [4]), and use nearest-neighbor to predict scene classes in a test set.

In terms of number of lines of code, this assignment is fairly small. However, it may take *a few hours* to finish running the baseline system, so make sure you start early so that you have time to debug things. Try printing statements within long-running functions to verify that the function did not hang. Also, try **each component** on **a subset of the data set** first before putting everything together. We provide you with a number of functions and scripts in the hopes of alleviating some tedious or error-prone sections of the implementation. You can find a list of files provided in Section 4. *Though not necessary, you are recommended to implement a multi-processing*[3] *version to make use of multiple CPU cores to speed up the code.* Functions with `n_worker` as input can benefit greatly from parallel processing.

Hyperparameters We provide you with a basic set of hyperparameters, which might not be optimal. You will be asked in Q3.1 to tune the system you built and we suggest you to keep the defaults before you get to Q3.1. All hyperparameters can be found in a single configuration file `opts.py`.

1 Representing the World with Visual Words

1.1 Extracting Filter Responses

We want to run a filter bank on an image by convolving each filter in the bank with the image and concatenating all the responses into a vector for each pixel. In our case, we will be using 4 types of filters of multiple scales (`opts.filter_scales`). The filters are: (1) Gaussian, (2) Laplacian of Gaussian, (3) derivative of Gaussian in the x direction, and (4) derivative of Gaussian in the y direction. The convolution function `scipy.ndimage.convolve()` can be used with user-defined filters, but the functions `scipy.ndimage.gaussian_filter()` and `scipy.ndimage.gaussian_laplace()` may be useful here for improved efficiency. Note that by default `scipy.ndimage` applies filters to all dimensions including channels. Therefore you might want to filter each channel separately. You can also pass in a parameter indicating you want either the x or y derivative.

[2] http://groups.csail.mit.edu/vision/SUN/

[3] Note that multi-threading in python does not make use of multiple CPU cores. It may not work on windows jupyter notebook.

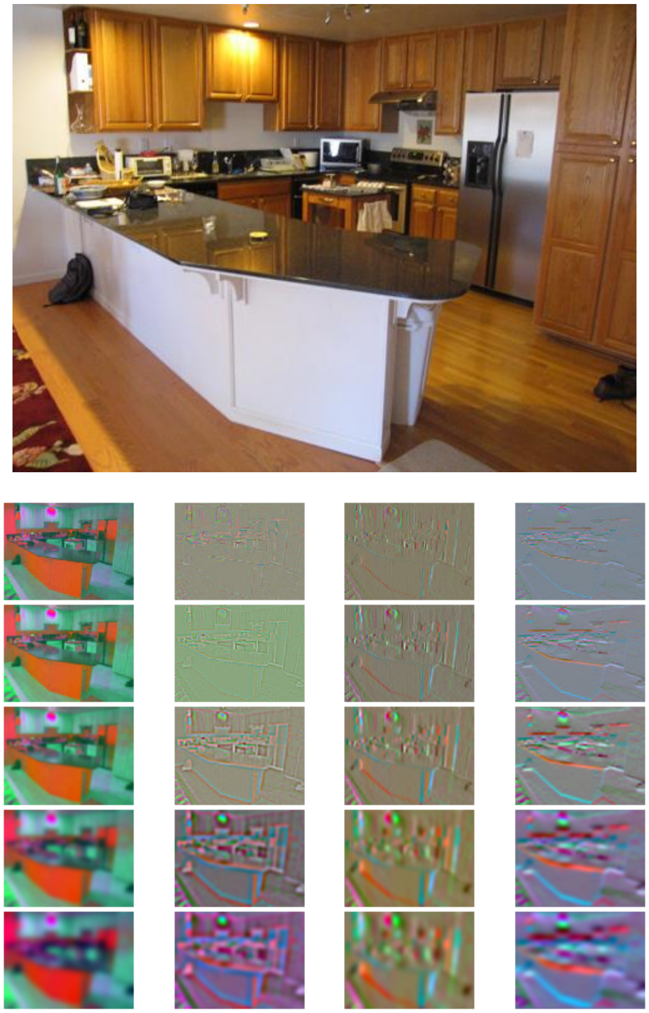

Figure 4: An input image and filter responses for all of the filters in the filter bank. (a) The input image (b) The filter responses in Lab colorization, corresponding to the filters in Fig 3

Q1.1.1 (5 points): 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? **Answer in your write-up.**

Q1.1.2 (10 points): For the code, loop through the filters and the scales to extract responses. Since color images have 3 channels, you are going to have a total of $3F$ filter responses per pixel if the filter bank is of size F. Note that in the given dataset, there are some gray-scale images. For those gray-scale images, you can simply duplicate them into three channels. Then output the result as a $3F$ channel image. Try to first iterate across scales and then for each scale, iterate across each channel (i.e. Scale$_1$ {Gaussian {R,G,B}, Laplacian {R, G, B}, ...}, Scale$_2$ {Gaussian{R,G,B}, Laplace{R, G, B}, ...}). Use zero-padding if necessary. Normalize the input before passing the image to extract_filter_responses. Complete the function

```
visual_words.extract_filter_responses(opts, img)
```

and return the responses as `filter_responses`. We have provided you with template code, with detailed instructions commented inside.

Remember to check the input argument `image` to make sure it is a floating point type with range $[0, 1]$, and convert it if necessary. Be sure to check the number of input image channels and convert it to 3-channel if it is not. Before applying the filters, use the function `skimage.color.rgb2lab()` to convert your image into the `Lab` color space, which is designed to more effectively quantify color differences with respect to human perception. (See **here** for more information.) If the input `image` is an $M \times N \times 3$ matrix, then `filter_responses` should be a matrix of size $M \times N \times 3F$. Make sure your convolution function call handles image padding along the edges sensibly.

Apply all 4 filters 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 $M \times N \times 3$) can help you to create the collage. **Submit the collage of images in your write-up.**

1.2 Creating Visual Words

You will now create a dictionary of visual words from the filter responses using k-means. After applying k-means, similar filter responses will be represented by the same visual word. You will use a dictionary with a fixed size. Instead of using all of the filter responses (**which might exceed the memory capacity of your computer**), you will use responses at α random pixels. If there are T training images, then you should collect a matrix `filter_responses` over all the images that is $\alpha T \times 3F$, where F is the filter bank size. 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` as follows:

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

If you like, you can pass the `n_jobs` argument into the `KMeans()` object to utilize parallel computation.

Q1.2 (10 points): Write the functions

`visual_words.compute_dictionary(opts, n_worker)`,
`visual_words.compute_dictionary_one_image(args)` (optional, multi-processing),

Given a dataset, these functions generate a dictionary. The overall goal of `compute_dictionary()` is to load the training data, iterate through the paths to the image files to read the images, and extract αT filter responses over the training files, and call k-means. This can be slow to run; however, the images can be processed independently and in parallel. Inside `compute_dictionary_one_image()`, you should read an image, extract the responses, and save to a temporary file. Here `args` is a collection of arguments passed into the function. Inside `compute_dictionary()`, you should load all the training data and create subprocesses to call `compute_dictionary_one_image()`. After all the subprocesses finish, load the temporary files back, collect the filter responses, and run k-means. A list of training images can be found in `data/train_files.txt`.

Finally, execute `compute_dictionary()`, and go do some push-ups while you wait for it to complete. If all goes well, you will have a file named `dictionary.npy` that contains the dictionary of visual words. If the clustering takes too long, reduce the number of clusters and samples. You can start with a tiny subset of training images for debugging.

1.3 Computing Visual Words

Q1.3 (10 points): 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`. We will use the standard Euclidean distance to do this; to do this efficiently, use the function `scipy.spatial.distance.cdist()`. Some sample results are shown in Fig 5.

Visualize wordmaps for three images. **Include these in your write-up, along with the original RGB images. Include some comments on these visualizations: do the "word" boundaries make sense to you?** The visualizations should look similar to the ones in Fig 5. Don't worry if the colors don't look the same, newer `matplotlib` might use a different color map.

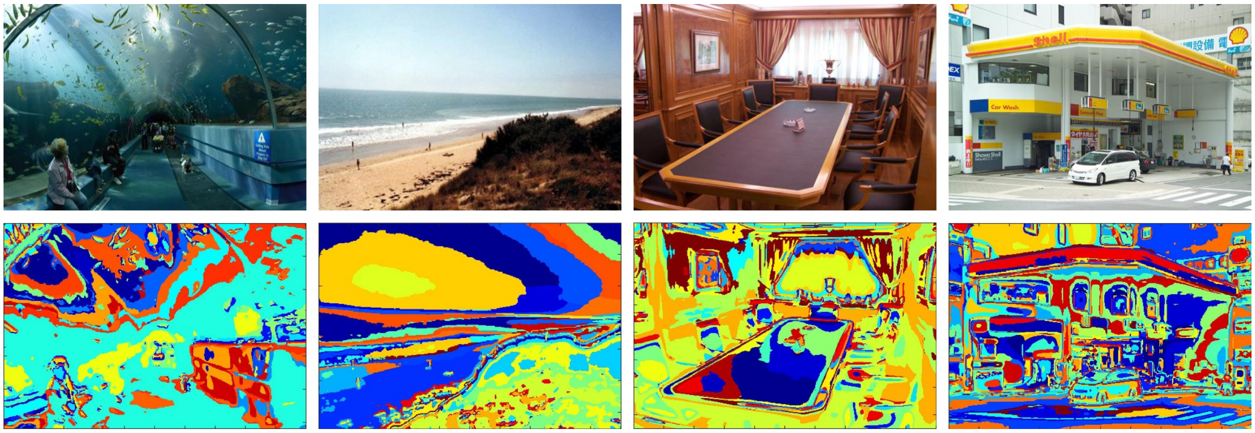

Figure 5: Visual words over images. You will use the spatially unordered distribution of visual words in a region (a bag of visual words) as a feature for scene classification, with some coarse information provided by spatial pyramid matching [4].

2 Building a Recognition System

We have formed a convenient representation for recognition. We will now produce a basic recognition system with spatial pyramid matching. The goal of the system is presented in Fig 1: given an image, classify (i.e., recognize/name) the scene depicted in the image.

Traditional classification problems follow two phases: training and testing. At training time, the computer is given a pile of formatted data (*i.e.*, a collection of feature vectors) with corresponding labels (*e.g.*, "desert", "park") and then builds a model of how the data relates to the labels (*e.g.*, "if green, then park"). At test time, the computer takes features and uses these rules to infer the label (*e.g.*, "this is green, therefore it is a park").

In this assignment, we will use the simplest classification method: nearest neighbor. At test time, we will simply look at the query's nearest neighbor in the training set and transfer that label. In this example, you will be looking at the query image and looking up its nearest neighbor in a collection of training images whose labels are already known. This approach works surprisingly well given a huge amount of data. (For a cool application, see the work by Hays & Efros [3]).

The key components of any nearest-neighbor system are:

- features (how do you represent your instances?) and
- similarity (how do you compare instances in the feature space?).

You will implement both.

2.1 Extracting Features

We will first represent an image with a bag of words. In each image, we simply look at how often each word appears.

Q2.1 (10 points): 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). As output, the function will return `hist`, an "L_1 normalized" `dict_size`-length histogram The L_1 normalization makes the sum of the histogram equal to 1. You may wish to load a single visual word map, visualize it, and verify that your function is working correctly before proceeding.

2.2 Multi-resolution: Spatial Pyramid Matching

A bag of words is simple and efficient, but it discards information about the spatial structure of the image and this information is often valuable. One way to alleviate this issue is to use spatial pyramid matching [4]. The general idea is to divide the image into a small number of cells, and concatenate the histogram of each of these cells to the histogram of the original image, with a suitable weight.

Here we will implement a popular scheme that chops the image into $2^l \times 2^l$ cells where l is the layer number. We treat each cell as a small image and count how often each visual word appears. This results in a histogram for every single cell in every layer. Finally to represent the entire image, we concatenate all the histograms together after normalization by the total number of features in the image. If there are $L+1$ layers and K visual words, the resulting vector has dimension $K \sum_{l=0}^{L} 4^l = K\left(4^{(L+1)} - 1\right)/3$.

Now comes the weighting scheme. Note that when concatenating all the histograms, histograms from different levels are assigned different weights. Typically (and in the original work [4]), a histogram from layer l gets half the weight of a histogram from layer $l+1$, with the exception of layer 0, which is assigned a weight equal to layer 1. A popular choice is to set the weight of layers 0 and 1 to 2^{-L}, and set the rest of the weights to 2^{l-L-1} (*e.g.*, in a three layer spatial pyramid, $L = 2$ and weights are set to 1/4, 1/4 and 1/2 for layer 0, 1 and 2 respectively. See Fig 6 for an illustration of a spatial pyramid. Note that the L_1 norm (absolute values of all dimensions summed up together) for the final vector is 1.

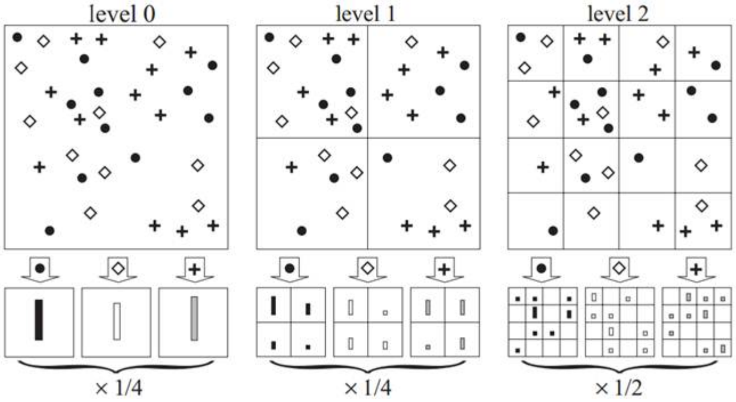

Figure 6: **Spatial Pyramid Matching:** From [4]. Toy example of a pyramid for L = 2. The image has three visual words, indicated by circles, diamonds, and crosses. We subdivide the image at three different levels of resolution. For each level of resolution and each channel, we count the features that fall in each spatial bin. Finally, weight each spatial histogram.

Q2.2 (15 points): Create a function `get_feature_from_wordmap_SPM` that forms 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 L_1 normalized.

One small hint for efficiency: a lot of computation can be saved if you first compute the histograms of the *finest* layer, because the histograms of coarser layers can then be aggregated from finer ones. Make sure you normalize the histogram after aggregation.

2.3 Comparing images

We need a way to compare images, to find the "nearest" instance in the training data. In this assignment, we'll use the histogram intersection similarity. The histogram intersection similarity between two histograms is the sum of the minimum value of each corresponding bins. This is a similarity score: the *largest* value indicates the "nearest" instance.

Q2.3 (10 points): Create the function

```
visual_recog.distance_to_set(word_hist, histograms)
```

where `word_hist` is a $K\left(4^{(L+1)}-1\right)/3$ vector and `histograms` is a $T \times K\left(4^{(L+1)}-1\right)/3$ matrix containing T features from T training samples concatenated along the rows. This function computes the histogram intersection similarity between `word_hist` and each training sample as a vector of length T and returns one minus the above quantity as a distance measure (distance is the inverse of similarity). Since this is called every time you look up a classification, you will want this to be fast! (Doing a for-loop over tens of thousands of histograms is a bad idea.)

2.4 Building A Model of the Visual World

Now that we've obtained a representation for each image, and defined a similarity measure to compare two spatial pyramids, we want to put everything up to now together.

Simple I/O code has been provided in the respective functions, which include loading the training images specified in `data/train_files.txt` and the filter bank and visual word dictionary from `dictionary.npy`, and also saving the learned model to `trained_system.npz`. Specifically in `trained_system.npz`, you should have:

1. `dictionary`: your visual word dictionary.
2. `features`: an $N \times K\left(4^{(L+1)}-1\right)/3$ matrix containing all of the histograms of the N training images in the data set.
3. `labels`: an N vector containing the labels of each of training images. (`features[i]` will correspond to label `labels[i]`).
4. `SPM_layer_num`: the number of spatial pyramid layers you used to extract the features for the training images.

Do not use the testing images for training!

The table below lists the class names that correspond to the label indices:

0	1	2	3	4	5	6	7
aquarium	desert	highway	kitchen	laundromat	park	waterfall	windmill

Q2.4 (15 points): 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()`.

2.5 Quantitative Evaluation

Qualitative evaluation is all well and good (and very important for diagnosing performance gains and losses), but we want some hard numbers.

Load the test images and their labels, and compute the predicted label of each one. That is, compute the test image's distance to every image in the training set, and return the label of the closest training image. To quantify the accuracy, compute a confusion matrix `C`. In a classification problem, the entry `C(i,j)` of a confusion matrix counts the number of instances of class `i` that were predicted as class `j`. When things are going well, the elements on the diagonal of `C` are large, and the off-diagonal elements are small. Since there are 8 classes, `C` will be 8×8. The accuracy, or percent of correctly classified images, is given by the trace of `C` divided by the sum of `C`. **Hint:** The accuracy with default parameters is 50%.

Q2.5 (10 points): 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.** This does not have to be formatted prettily: if you are using LaTeX, you can simply copy/paste it into a `verbatim` environment.

2.6 Find the failures

There are some classes/samples that are more difficult to classify than the rest using the bags-of-words approach. As a result, they are classified incorrectly into other categories.

Q2.6 (5 points): In your writeup, list some of these hard classes/samples, and discuss why they are more difficult than the rest.

3 Improving performance

3.1 Hyperparameter tuning

Now we have a full-fledged recognition system plus an evaluation system, it's time to boost up the performance. In practice, it is most likely that a model will not work well out-of-the-box. It is important to know how to tune a visual recognition system for the task at hand.

Q3.1 (15 points): Tune the system you build to reach around 65% 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%). Also, describe why you think changing a particular parameter should increase or decrease the overall performance in the table you show.

- `filter_scales`: a list of filter scales used in extracting filter response;
- `K`: the number of visual words and also the size of the dictionary;
- `alpha`: the number of sampled pixels in each image when creating the dictionary;
- `L`: the number of spatial pyramid layers used in feature extraction.

3.2 Further improvement

Q3.2 (10 points): Can you improve your classifier, in terms of accuracy or speed? Be creative! Or be well-informed, and cite your sources! For some quick ideas, try resizing the images, subtracting the mean color, changing the structure or weights of the spatial pyramid, or replacing the histogram intersection with some other similarity score. Whatever you do, explain (1) what you did, (2) what you expected would happen, and (3) what actually happened. Include these results and code in your write-up.

Q3.3 Extra Credit (20 points):

Inverse Document Frequency: With the bag-of-words model, image recognition is similar to classifying a document with words. In document classification, inverse document frequency (IDF) factor is incorporated which diminishes the weight of terms that occur very frequently in the document set. For example, because the term ”the” is so common, this will tend to incorrectly emphasize documents which happen to use the word ”the” more frequently, without giving enough weight to the more meaningful terms.

In the homework, the bag of words we computed only considers the term frequency (TF), i.e. the number of times that word occurs in the word map. Now we want to weight the word by its inverse document frequency. The IDF of a word is defined as:

$$IDF_w = log \frac{T}{|\{d : w \in d\}|}$$

Here, T is number of all training images, and $|\{d : w \in d\}|$ is the number of images d such that w occurs in that image.

Write a function `compute_IDF` to compute a vector IDF of size $1 \times K$ containing IDF for all visual words, where K is the dictionary size. Save the extracted IDF in idf.npy. Then write another function `evaluate_recognition_system_IDF` that makes use of the IDF vector in the recognition process. You can use either nearest neighbor or anything you have from Q3.1 as your classifier. In your writeup: How does Inverse Document Frequency affect the performance? Better or worse? Explain your reasoning.

4 HW1 Distribution Checklist

After unpacking `hw1.zip`, you should have a folder `hw1` containing one folder for the data (`data`) and one for your code (`code`). In the `code` folder, where you will primarily work, you will find:

- `visual_words.py`: function definitions for extracting visual words.
- `visual_recog.py`: function definitions for building a visual recognition system.
- `util.py`: some utility functions
- `main.py`: main function for running the system

The data folder contains:

- `data/`: a directory containing `.jpg` images from the SUN database.
- `data/train_files.txt`: a text file containing a list of training images.
- `data/train_labels.txt`: a text file containing a list of training labels.
- `data/test_files.txt`: a text file containing a list of testing images.
- `data/test_labels.txt`: a text file containing a list of testing labels.

5 HW1 submission checklist

Submit your write-up and code to Gradescope.

- **Writeup.** The write-up should be a pdf file named **<AndrewId>_hw1.pdf**. **You must select the pages of the writeup that correspond to each question**. Your writeup should include your answers to the following:
  - Q1.1.1 (around 4 lines of text)
  - Q1.1.2 (visualization of filter responses)
  - Q1.3 (visualization of wordmaps)
  - Q2.5 (a confusion matrix, and an accuracy value)
  - Q2.6 (hard examples, and an explanation)
  - Q3.1 (table of ablation study and your final accuracy)
  - Q3.2 (model improvements)
  - Extra credit (idea, expectation, result)

- **Code.** The code should be submitted as a zip file named **<AndrewId>_hw1.zip**. By extracting the zip file, it should have the following files in the structure defined below.

  When you submit, remove the folder `data/` and `feat/` if applicable, as well as any large temporary files that we did not ask you to create.

  - **<andrew_id>**/ # A directory inside .zip file
    - `code/`
      - `dictionary.npy`
      - `trained_system.npz`
      - <!– all of your .py files >
    - `<andrew_id>_hw1.pdf` make sure you upload this pdf file to Gradescope. Please assign the locations of answers to each question on Gradescope.

References

[1] K. Chatfield, V. Lempitsky, A. Vedaldi, and A. Zisserman. The devil is in the details: an evaluation of recent feature encoding methods. In *British Machine Vision Conference*, 2011.

[2] K. Grauman and T. Darrell. The pyramid match kernel: discriminative classification with sets of image features. In *Computer Vision (ICCV), 2005 IEEE International Conference on*, volume 2, pages 1458–1465 Vol. 2, 2005.

[3] James Hays and Alexei A Efros. Scene completion using millions of photographs. *ACM Transactions on Graphics (SIGGRAPH 2007)*, 26(3), 2007.

[4] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer Vision and Pattern Recognition (CVPR), 2006 IEEE Conference on*, volume 2, pages 2169–2178, 2006.

[5] D.G. Lowe. Object recognition from local scale-invariant features. In *Computer Vision (ICCV), 1999 IEEE International Conference on*, volume 2, pages 1150–1157 vol.2, 1999.

[6] Laura Walker Renninger and Jitendra Malik. When is scene identification just texture recognition? *Vision research*, 44(19):2301–2311, 2004.

[7] J. Winn, A. Criminisi, and T. Minka. Object categorization by learned universal visual dictionary. In *Computer Vision (ICCV), 2005 IEEE International Conference on*, volume 2, pages 1800–1807 Vol. 2, 2005.

[8] Jianxiong Xiao, J. Hays, K.A. Ehinger, A. Oliva, and A. Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3485–3492, 2010.