16720-B F20 Bag of Visual Words

Instructor: Kris Kitani

TAs: Zhengyi Luo (Lead), Alireza Golestaneh, Nadine Chang, Cormac O'Meadhra, Anand Bhoraskar

Due Oct 5, 2020 11:59 PM

Total Points: 140 + 20 (ec)

Perhaps the most common problem in computer vision is classification. Given an image that comes from a few pre-defined categories, can you determine which category it belongs to? For this assignment, you will be developing a system for scene classification. You will be experimenting on a subset of the SUN Image database [3] consisting of eight scene categories and build an end to end system that will, given a new scene image, determine which type of scene it is, as shown in Figure 1.

Instructions

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must submit their own work. If you work as a group, include the names of your collaborators in your write up. Code should **NOT** be shared or copied. For this assignment, you can use common image processing libraries such as `cv2` or `skimage` for basic image operations, though **DO NOT** use functions that you are supposed to implement. Plagiarism is strongly prohibited and may lead to failure of this course.

2. **Start early!** Especially those not familiar with Python. Constructing the dictionary in **Q1.2.1**, as well as converting all the images to visual words in **Q1.3.1** can sometimes take up to 30 mins to run (if implemented well!). If you start late, you will be pressed for time while debugging.

3. **Questions:** If you have any question, please look at piazza first. Other students may have encountered the same problem, and it may be solved already. If not, post your question in the specified folder. TAs will respond as soon as possible.

4. **Write-up:** Items to be included in the writeup are mentioned in each question in **bold**. Please note that we **DO NOT** accept handwritten scans for your write-up in this assignment (that include digital handwritten work). Please type your answers to theory questions and discussions for experiments electronically.

5. **Handout:** After unpacking `hw2.zip`, you should have a folder `hw2` 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_data.npz`: a `.npz` file containing the training set.
   - `data/test_data.txt`: a `.npz` file containing the test set.
   - `data/vgg16_list.npy`: a `.npy` file with the weights of VGG-16.

6. **Submission:** The submission is on Gradescope, **you will be submitting both your writeup and code zip file**. The zip file, `<andrew-id.zip>`, contains your python implementations (including helper functions), results for extra credit (optional). Do not hand in the image files we distributed in the handout zip, or any of the generated wordmap files. However you should hand in the dictionary and recognition-system files that you generate (follow instructions by individual questions). **Note: You have to submit your writeup separately to Gradescope as `<andrew-id.pdf>`.**

   Your final upload should have the files arranged in this layout:

   `<AndrewID>.zip`

   - `<AndrewId>`
     - `code`
       - `dictionary.npy`
       - `trained_system.npz`
       - `trained_system_deep.npz`
       - ⟨ all of your .py files ⟩
     - `<andrew id> hw1.pdf` make sure you upload this pdf file to Gradescope as well. Please assign the locations of answers to each question on Gradescope.

7. Assignments that do not follow this submission rule will be **penalized 10% of the total score**.

8. Please make sure that the file paths that you use are relative and not absolute.

Overview

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.

This assignment is based on an approach for document classification called **Bag of Words**. It represents a document as a vector or histogram of counts for each word that occurs in the document, as shown in Figure 2. The hope is that different documents in the same class will have a similar collection and distribution of words, and that when we see a new document, we can find out which class it belongs to by comparing it to the word distribution already seen in that class. This approach has been very successful in Natural Language Processing, which is surprising due to its relative simplicity (we are throwing away all the sequential relationships and representing each document as mere counts of words it contains!). We will be taking inspiration from this approach and apply it to image classification: imagine each image as a document, and we want to classify each document as a scene.

This assignment has 4 major sections:

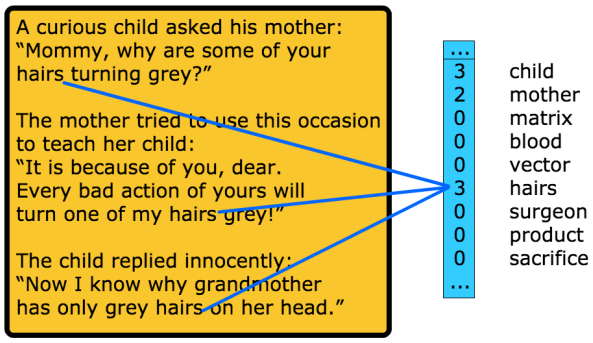

Figure 2: Bag of words representation of a text document

- **Section 1**: build a dictionary of *visual words* from training data. In this section, you will use Harris corner detector to pick interest points, extract visual words from interest points on images, form a visual dictionary, and represent each image as a vector of visual words.
- **Section 2**: build a recognition system using visual word dictionary and training images. In this section, you will use the visual words extracted from the previous section and a technique called *Spatial Pyramid Matching* to extract visual features from images. You will use these features and ground truth labels to build a simple recognition system.
- **Section 3**: evaluate the recognition system on test images. In this section, you and evaluate your system and classify a given image to 8 types of scenes.
- **Section 4**: Explore an alternative to BoW – Deep Learning Features, and compare results between the two approaches.

An illustrative overview of the homework is shown in Figure. 3.

Notice that it may take a long time to finish running the baseline system, so make sure you start early and can have time to debug things. Also, try **each component** on **a subset of the data set** first before putting everything together.

Notice that, we include `num_workers` as input for some functions you need to implement. Those are not necessary, but can be used with multi-threading python libraries to significantly speed up your code.

This homework was tested using python3.6.9 and pytorch 1.6.0, installed through Anaconda 3. All libraries can be installed using conda. You would not need any other libraries apart from the ones already imported in the template code. Feel free to use other modules from Python Standard Library.

1 Representing the World with Visual Words

1.1 Extracting Filter Responses

To extract image features as visual words, we will rely on using a multi-scale filter bank. 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 20 filters consisting of 4 types of filters in 5 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 from *scipy*, `scipy.ndimage.convolve()`, can be used with user-defined filters, but you can also use predefined functions for filter generation to improve efficiency: `scipy.ndimage.gaussian_filter()` and `scipy.ndimage.gaussian_laplace()`. The 5 scales we will be using are 1, 2, 4, 8, and $8\sqrt{2}$, in pixel units.

Q1.1.1 (5 points): What visual properties do each of the filter functions (See Figure 4) pick up? You should group the filters into categories by its purpose/functionality. Also, why do we need multiple scales of filter responses? **Answer in the writeup. Answer in your write-up**.

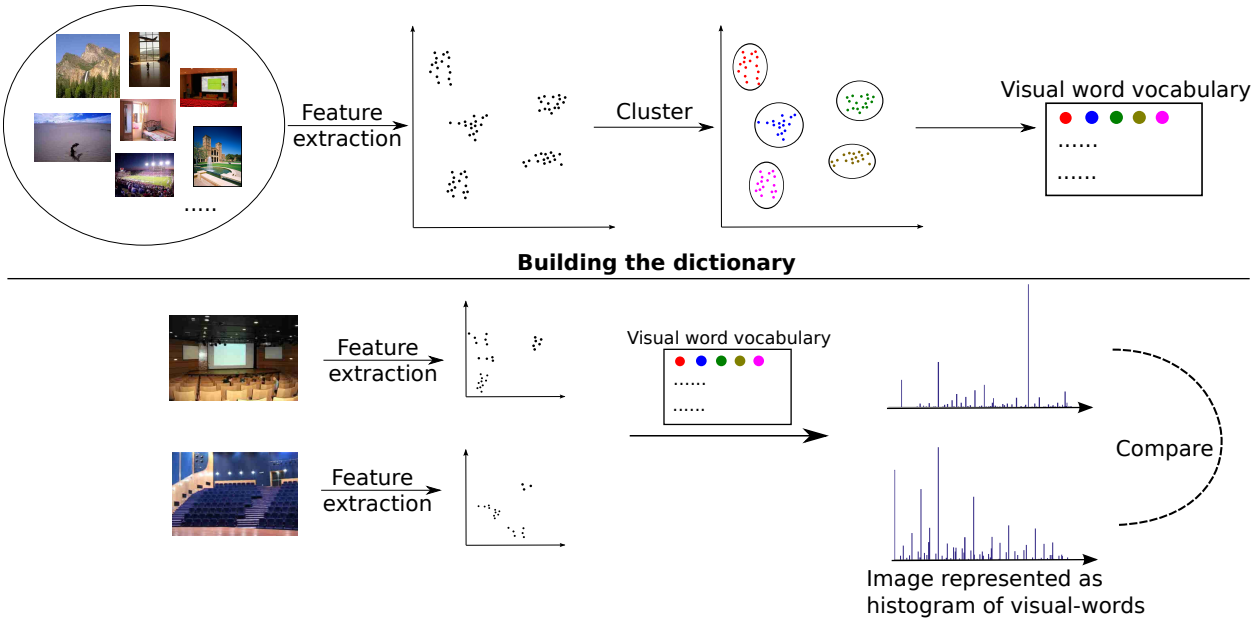

Figure 3: An overview of the bags-of-words approach to be implemented in the homework. Given the training set of images, the visual features of the images are extracted. In our case, we will use the filter responses of the pre-defined filter bank as the visual features. The visual words, *i.e.* dictionary, are built as the centers of clusterings of the visual features. During recognition, the image is first represented as a vector of visual words. Then the comparison between images is realized in the visual-word vector space. Finally, we will build a scene recognition system that classifies the given image into 8 types of scenes

Q1.1.2 (10 points): To implement, 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 stack them into three channels using the command `np.stack`. Then output the result as a $3F$ channel image. Complete the function

```
visual_words.extract_filter_responses(image)
```

and return the responses as `filter_responses`. We have provided you with a template code with detailed instructions in it. You would be required to input a 3-channel RGB or gray-scale image and filter bank to get the responses of the filters on the image.

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 was designed to more effectively quantify color differences with respect to human perception (See here for more information.). Notice that after converting the image to `Lab` color space, it will no longer in the range of [0,1], this is expected. If `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 (by passing in the right options).

Apply all 20 filters on the image aquarium/sun_aztvjgubyrgvirup.jpg, and visualize as a image collage (as shown in Figure 5). You can use 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 $M \times N \times 3$) to create the collage. **Submit the collage of 20 images in the 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

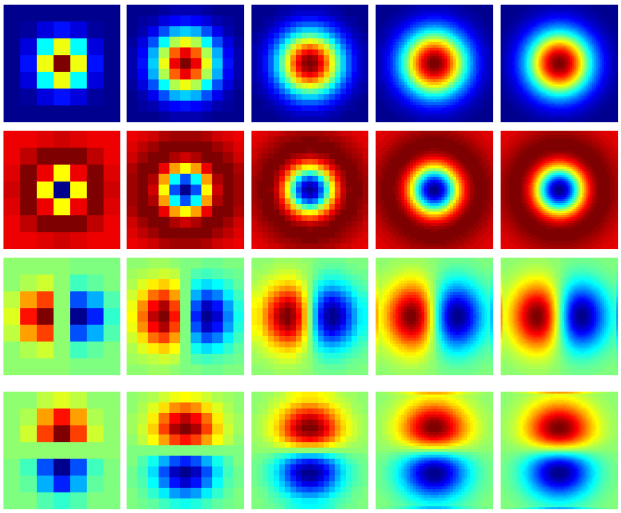

Figure 4: The provided multi-scale filter bank

a fixed number of visual words. Instead of using all of the filter responses at each of the pixels from the given image(**that can exceed the memory capacity of your computer**), you will use responses at α interest points from each image, chosen using the Harris Corner Detector. 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 (20 in our case). Then, to generate a visual words dictionary with K words, 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_
```

You can alternatively pass the `n_jobs` argument into the `KMeans()` object to utilize parallel computation.

Q1.2.1 (10 points): As covered in class, the Harris Corner Detector is an effective way of selecting points of interest from images. This algorithm finds corners by building a covariance matrix of edge gradients within a region around a point in the image. The eigenvectors of this matrix point in the two directions of greatest change. If they are both large, then this indicates a corner. See class slides for more details.

In this question, you will implement:

```
visualwords.get_harris_points(image, alpha, k)
```

This function takes the input image, α as number of points we want to choose, and k the sensitivity factor. The input image may either be a color or grayscale image, but the following operations should be done on the grayscale representation of the image (feel free to use functions from `cv2`, `skimage`, etc. for the conversion). For each pixel, you need to compute the covariance matrix:

$$H = \begin{bmatrix} \sum_{p \in P} I_{xx} & \sum_{p \in P} I_{xy} \\ \sum_{p \in P} I_{yx} & \sum_{p \in P} I_{yy} \end{bmatrix}$$

where $I_{ab} = \frac{\partial I}{\partial a} \frac{\partial I}{\partial b}$, and p is the current window.. You can use a 3×3 or 5×5 window. To compute image's X and Y gradients, you can use the sobel filter (*e.g.* cv2.Sobel(...)). For the sum, also think about how you could do it using a convolution filter.

You then want to detect corners by finding pixels who's covariance matrix eigenvalues are large. Since its expensive to compute the eigenvalues explicitly, you should instead compute the response function with:

$$R = \lambda_1 \lambda_2 - k \left(\lambda_1 + \lambda_2\right)^2 = \det(H) - k \operatorname{tr}(H)^2$$

`det` represents the determinant and `tr` denotes the trace of the matrix. Recall that when detecting corners with the Harris corner detector, corners are points where both eigenvalues are large. This is in contrast to

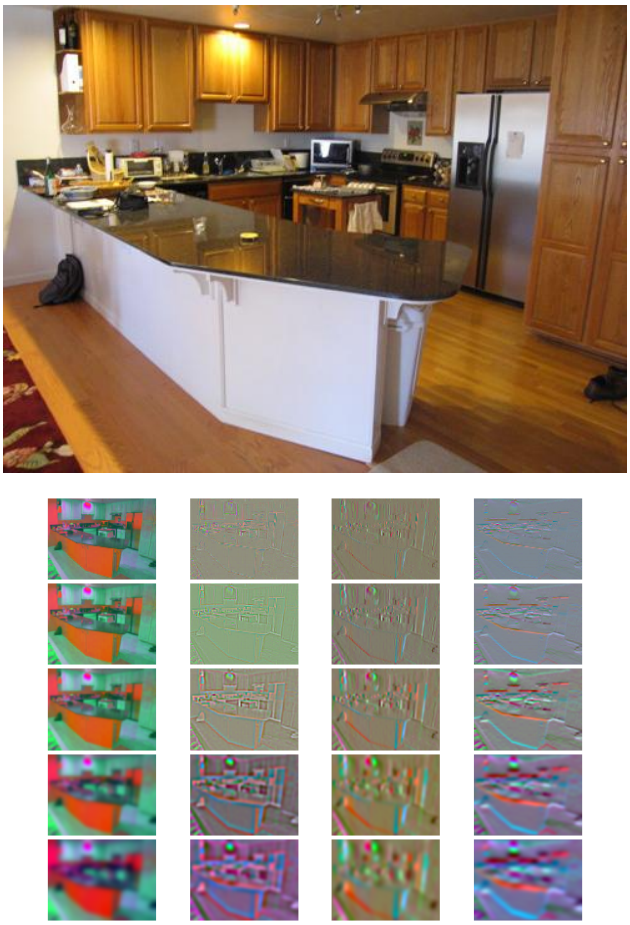

Figure 5: An input image and filter responses for all of the filters in the filter bank. (a) The input image (b) The filter responses of Lab image corresponding to the filters in Figure. 4

edges (one eigenvalue is larger, while the other is small), and flat regions (both eigenvalues are small). In the response function, the first term becomes very small if one of the eigenvalues are small, thus making $R < 0$. Larger values of R indicates similarly large eigenvalues.

Instead of thresholding the response function, simply take the top α response as the corners, and return their coordinates. A good value for the k parameter is 0.04 - 0.06. **In your writeup: Show the results of your corner detector on 3 random images from the provided dataset.**

Note: You will be applying this function to a lot of images, try to implement this without loops using vectorization (you should have implemented something similar in HW1). However, there is no penalty if your implementation is slow.

Another Note: For the next part, you can also try selecting points of interest at random from the images and use it as a baseline method. Compare the two methods (Harris vs Random) and think about why the performance are different or similar. Make sure that your submitted version is using the Harris corner detector though, since we want to make sure that your implementation is sound.

Q1.2.2 (10 points): Now that we have a way to pick interest points from our images, we will build a dictionary using features from these points of interest. You should write the functions:

```
visual_words.compute_dictionary_one_image(args)
visual_words.compute_dictionary()
```

to generate a dictionary given a list of images. 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 are finished, load the temporary files back, collect the filter responses, and run k-means. A sensible initial value to try for K is between 100 and 300, and for α is between 50 and 500, but they depend on your system configuration and you might want to play with these values.

Finally, execute `compute_dictionary()` and go get a coffee. 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. If you have debugging issues, try passing in a small number of training files manually.

1.3 Computing Visual Words

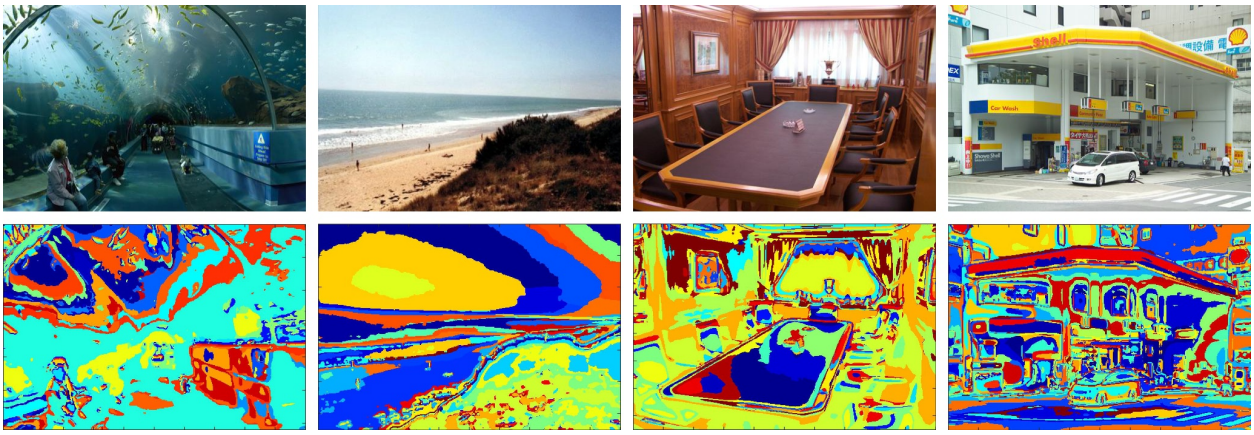

Figure 6: Visual words over images. You will use the spatially un-ordered distribution of visual words in a region (a bag of visual words) as a feature for scene classification, with some coarse information provided by spatial pyramid matching [2]

Q1.3.1 (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(image,dictionary)
```

and return `wordmap`, a matrix with the same width and height as *image*, where each pixel in `wordmap` is matched to the closest visual word based on the filter response at the respective pixel in `image`. We will use the standard Euclidean distance to match the filter response and our visual words; to do this efficiently, use the function `scipy.spatial.distance.cdist()`. Some sample results are shown in Fig. 6.

Visualize three wordmaps of images from any one of the category. **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?**. We have provided helper function to save and visualize the resulting wordmap in the `util.py` file. They should look similar to the ones in Figure 6.

2 Building a Recognition System

We have formed a convenient way to represent images 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 where the image was taken.

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", "kitchen") and then builds a model of how the data relates to the labels: "if green, then kitchen". At test time, the computer takes features and uses these rules to infer the label: *e.g.*, "this is green, so therefore it is kitchen".

In this assignment, we will use the simplest classification model: 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, *e.g.*, a very cool graphics applications from [1].

The key components of any nearest-neighbor system are:

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

You will implement both in this section.

2.1 Extracting Features

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

Q2.1.1 (10 points): Write the function

```
visual_recog.get_feature_from_wordmap(wordmap,dict_size)
```

that extracts the histogram[1] of visual words within the given image (*i.e.*, the bag of visual words). As inputs, the function will take:

- `wordmap` is a `H` $\times$ `W` image containing the IDs of the visual words
- `dict_size` is the maximum visual word ID (*i.e.*, the number of visual words, the dictionary size). Notice that your histogram should have `dict_size` bins, corresponding to how often that each word occurs.

As output, the function will return `hist`, a `dict_size` histogram that is L_1 normalized, (*i.e.*, the sum equals 1). **Load a single visual word map, visualize its histogram, and include it in the write up.** This will help you verifying that your function is working correctly before proceeding.

2.2 Multi-resolution: Spatial Pyramid Matching

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 [2]. 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. If there are $L+1$ layers and K visual words, the resulting vector has dimensionality $K \sum_{l=0}^{L} 4^l = K\left(4^{(L+1)}-1\right)/3$.

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 [2]), features from layer l gets half the weight of features from layer $l+1$, with the exception of layer 0, which is assigned a weight equal to layer 1. A popular choice is for layer 0 and layer 1 the weight is set to 2^{-L}, and for the rest it is set 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. 7). Take level 2 as an example, there will be 16 histograms in total, each has a norm equal to one. You should concatenate these histograms, normalize this layer (multiply the concatenated vector by $1/16$), and apply the $1/2$ layer weight of level 2 on top of that. Note that following this operation, concatenating the weighted features of each layer will result in a final vector of norm equal to 1.

Q2.2.1 (15 points): Create a function `getImageFeaturesSPM` that form a multi-resolution representation of the given image.

[1] Look into `numpy.histogram()`

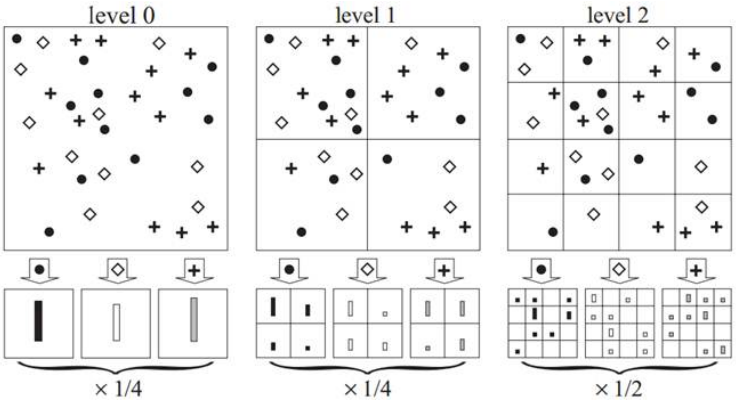

Figure 7: **Spatial Pyramid Matching:** From [2]. Toy example of a pyramid for L = 2. The image has three visual words, indicated by circles, diamonds, and crosses. We subdivide the image at three different levels of resolution. For each level of resolution and each channel, we count the features that fall in each spatial bin. Finally, weight each spatial histogram.

```
visual_recog.get_feature_from_wordmap_SPM(wordmap,layer_num,dict_size)
```

As inputs, the function will take:

- `layer_num` the number of layers in the spatial pyramid, *i.e.*, $L + 1$
- `wordmap` is a `H` $\times$ `W` image containing the IDs (*i.e.* index) of the visual words
- `dict_size` is the maximum visual word ID (*i.e.*, the number of visual words, the dictionary size)

As output, the function will return `hist_all`, a vector that is L_1 normalized. **Please use a 3-layer spatial pyramid ($L = 2$) for all the following recognition tasks.**

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 will also need a way of comparing 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. Note that since this is a similarity, you want the *largest* value to find the "nearest" instance.

Q2.3.1 (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 returns the histogram intersection similarity between `word_hist` and each training sample as a vector of length T. Since this is called every time you want to look up a classification, you want this to be fast, so doing a for-loop over tens of thousands of histograms is a very 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.

You will need to load the training file names from `data/train_data.npz` and the filter bank and visual word dictionary from `dictionary.npy`. You will save everything to a `.npz` numpy-formated (use `np.savez`) file named `trained_system.npz`. Included will be:

1. `dictionary`: your visual word dictionary.
2. `features`: a $N \times K\left(4^{(L+1)}-1\right)/3$ matrix containing all of the histograms of the N training images in the data set. A dictionary with 150 words will make a `train_features` matrix of size 1440×3150.
3. `labels`: an N vector containing the labels of each of the 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.

We have provided you with the names of the training images in `data/train_data.npz`. You want to use the dictionary entry `image_names` for training. You are also provided the names of the test images in `data/test_data.npz`, which is structured in the same way as the training data; however, *you cannot use the testing images for training.*
If it's any helpful, the below table lists the class names that correspond to the label indices:

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

Q2.4.1 (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(file_path,dictionary,layer_num,K)
```

that load image, extract word map from the image, compute SPM feature and return the computed feature. Use this function in your `visual_recog.build_recognition_system()`.

3 Quantitative Evaluation

3.1 Calculating confusion matrix

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

Load the corresponding test images and their labels, and compute the predicted labels of each, i.e., compute its distance to every image in training set and return the label with least distance difference as the predicted label. To quantify the accuracy, you will compute a confusion matrix `C`: given 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`.

Q3.1.1 (10 points): Implement the function

```
visual_recog.evaluate_recognition_system()
```

that tests the system and outputs the confusion matrix. Report the confusion matrix and accuracy for your results 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. Additionally, do not worry if your accuracy is low: with 8 classes, chance is 12.5%. To give you a more sensible number, a reference implementation *with* spatial pyramid matching gives an overall accuracy of around 50%.
Q3.1.2 (5 points): As there are some classes/samples that are more difficult to classify than the rest using the bags-of-words approach, they are more easily classified incorrectly into other categories. **List some of these classes/samples and discuss why they are more difficult in your write-up.**
Q3.1.3 [Extra Credit](10 points): Now that you have seen how well your recognition system can perform on a set of real images, you can experiment with different ways of improving this baseline system. Here are a few suggestions:

- Hyperparameter Tuning: here is a list of hypterparameters in the system that you can tune to get better performance for your system:
  - `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.
- Image manipulation: Try using image augmentation techniques such as random-crop, flipping, etc. to obtain more training data for your system. You can also try resizing the images, subtracting the mean color, etc.
- Better classifier: in part 2 we used the nearest neighbor classifier to classify test images. However, with our extracted SPM features from training images, we can use other classifiers such as multi-class logistic regression, multi-class support vector machine, etc. to gain better performance. For this, you can use implementation of these algorithms from `scipy`.

Tune the system you build to reach around 65% accuracy on the provided test set (`data/test_data.npz`). **In your writeup, document what you did to achieve such performance: (1) what you did, (2) what you expected would happen, and (3) what actually happened.** Also, include a file called `custom.py` for running your code.

Q3.1.4 [Extra Credit](10 points): Inverse Document Frequency: With the bag-of-word 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 histogram 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 fucntion `visual_recog.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 `visual_recog.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 q3.1.4 as your classifier.

In your writeup: How does Inverse Document Frequency affect the performance? Better or worse? Explain your reasoning?

4 Deep Learning Features - An Alternative to "Bag of Words"

As we have discussed in class, another powerful method for scene classification in computer vision is the employment of convolutional neural networks (CNNs) - sometimes referred to generically as "deep learning". It is important to understand how previously trained (pretrained) networks can be used as another form of feature extraction, and how they relate to classical Bag of Words (BoW) features. We will be covering details on how one chooses the network architecture and training procedures later in the course. For this question, however, we will be asking you to deal with the VGG-16 pretrained network. VGG-16 is a pretrained Convolutional Neural Network (CNN) that has been trained on approximately 1.2 million images from the ImageNet Dataset (`http://image-net.org/index`) by the Visual Geometry Group

(VGG) at University of Oxford. The model can classify images into a 1000 object categories (e.g. keyboard, mouse, coffee mug, pencil).

One lesson we want you to take away from this exercise is to understand the effectiveness of "deep features" for general classification tasks within computer vision - even when those features have been previously trained on a different dataset (i.e. ImageNet) and task (i.e. object recognition).

4.1 Extracting Deep Features

To complete this question, you need to install the `torchvision` library from Pytorch, a popular Python-based deep learning library. If you are using the Anaconda package manager (`https://www.anaconda.com/download/`), this can be done with the following command:

```
conda install pytorch torchvision -c pytorch
```

To check that you have installed it correctly, make sure that you can `import torch` in a Python interpreter without errors. Please refer to `https://pytorch.org/` for more detailed installation instructions.

Q4.1.1 (25 points): We want to extract out deep features corresponding to the convolutional layers of the VGG-16 network. In this problem, we will use the trained weights from the VGG network directly, but implement our own operations.

To load the network, use the line

```
vgg16 = torchvision.models.vgg16(pretrained=True).double()
```

followed by

```
vgg16.eval()
```

The latter line ensures that the VGG-16 network is in evaluation mode, not training mode.

We want you to complete a function that is able to output `VGG-16` network outputs at the `fc7` layer in `network_layers.py`.

```
network_layers.extract_deep_feature(x,vgg16_weights)
```

where `x` refers to the input image and `vgg16_weights` is a structure containing the CNN's network parameters. In this function you will need to write sub-functions `multichannel_conv2d`, `relu`, `max_pool2d`, and `linear` corresponding to the fundamental elements of the CNN: multi-channel convolution, rectified linear units (ReLU), max pooling, and fully-connected weighting.

We have provided a helper function `util.get_VGG16_weights()` that extracts the weight parameters of VGG-16 and its meta information. The returned variable is a numpy array of shape $L \times 3$, where L is the number of layers in VGG-16. The first column of each row is a string indicating the layer type. The second/third columns may contain the learned weights and biases, or other meta-information (*e.g.* kernel size of max-pooling). Please refer to the function docstring for details.

VGG-16 assumes that all input imagery to the network is resized to 224×224 with the three color channels preserved (use `skimage.transform.resize()` to do this before passing any imagery to the network). And be sure to normalize the image using suggested mean and std before extracting the feature:

```
mean=[0.485,0.456,0.406]
std=[0.229,0.224,0.225]
```

In order to build the `extract_deep_feature` function, you should run a for-loop through each layer index until layer "fc7", which corresponds to **the second linear layer** (Refer to VGG structure to see where "fc7" is). **Remember**: the output of the preceding layer should be passed through as an input to the next. Details on the sub-functions needed for the `extract_deep_feature` function can be found below. Please use `scipy.ndimage.convolve` and `numpy` functions to implement these functions instead of using pytorch. Please keep speed in mind when implementing your function, for example, using double for loop over pixels is not a good idea.

1. `multichannel_conv2d(x,weight,bias)`: a function that will perform multi-channel 2D convolution which can be defined as follows,

$$\mathbf{y}^{(j)} = \sum_{k=1}^{K} \left[\mathbf{x}^{(k)} * \mathbf{h}^{(j,k)}\right] + \mathbf{b}[j] \tag{1}$$

where $*$ denotes $2D$ convolution, $\mathbf{x} = \{\mathbf{x}^{(k)}\}_{k=1}^{K}$ is our vectorized K-channel input signal, $\mathbf{h} = \{\mathbf{h}^{(j,k)}\}_{k=1,j=1}^{K,J}$ is our $J \times K$ set of vectorized convolutional filters and $\mathbf{r} = \{\mathbf{y}^{(j)}\}_{j=1}^{J}$ is our J channel vectorized output response. Further, unlike traditional single-channel convolution CNNs often append a bias vector $\mathbf{b}$ whose J elements are added to the J channels of the output response.

To implement `multichannel_conv2d`, you can use the Scipy convolution function directly with for loops to cycle through the filters and channels (`scipy.ndimage.convolve()`). All the necessary details concerning the number of filters (J), number of channels (K), filter weights ($\mathbf{h}$) and biases ($\mathbf{b}$) can be inferred from the shapes/dimensions of the weights and biases. Notice that pytorch's convolution function actually does correlation, so to get similar answer as pytroch with scipy, you need to flip the kernel on both axes using `np.flip()`.

2. `relu(x)`: a function that shall perform the Rectified Linear Unit (ReLU) which can be defined mathematically as,

$$\text{ReLU}(x) = \max_{x}(x, 0) \tag{2}$$

and is applied independently to each element of the matrix/vector `x` passed to it.

3. `max_pool2d(x,size)`: a function that shall perform max pooling over `x` using a receptive field of size `size` $\times$ `size` (we assume a square receptive field here for simplicity). If the function receives a multi-channel input, then it should apply the max pooling operation across each input channel independently. (Hint: making use of smart array indexing can drastically speed up the code.)

4. `linear(x,W,b)`: a function that will compute a node vector where each element is a linear combination of the input nodes, written as

$$\mathbf{y}[j] = \sum_{k=1}^{K} \mathbf{W}[j,k]\mathbf{x}[k] + \mathbf{b}[j] \tag{3}$$

or more succinctly in vector form as $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ - where $\mathbf{x}$ is the $(K \times 1)$ input node vector, $\mathbf{W}$ is the $(J \times K)$ weight matrix, $\mathbf{b}$ is the $(J \times 1)$ bias vector and $\mathbf{y}$ is the $(J \times 1)$ output node vector. You should not need for-loops to implement this function.

You should ignore all `DropoutLayer`s you encounter; they're functional only during the training phase. For efficiency you should check that each sub-function is working properly before putting them all together - otherwise it will be hard to track any errors. You can check the performance of each layer by creating your own single-layer network. To compare your implementation with pytroch, you should compare the extracted features between your `extract_deep_feature` and the pre-trained VGG-16 network. `deep_recog.evaluate_deep_extractor` should come in handy in comparing the result of the two extracted features. **In your writeup, report the difference between results of the two feature extractors on a random image (should be really small!).**

4.2 Building a Visual Recognition System: Revisited

We want to compare how useful deep features are in a visual recognition system. Since the speed of the function `scipy.ndimage.convolve` is not ideal, you can use the pytroch VGG-16 network directly (refer to the helper function `deep_recog.evaluate_deep_extractor` on how to use the pre-trained network as feature extractor).

Q4.2.1 (5 points): Implement the functions

```
deep_recog.build_recognition_system(vgg16)
```

and

```
deep_recog.eval_recognition_system(vgg16)
```

both of which takes the pretrained VGG-16 network as the input arguments.

The former function should produce `trained_system_deep.npz` as the output. Included will be:

1. `features`: a $N \times K$ matrix containing all the deep features of the N training images in the data set.
2. `labels`: an N vector containing the labels of each of the images. (`features[i]` will correspond to label `labels[i]`).

The latter function should produce the confusion matrix, as with the previous question. Instead of using the histogram intersection similarity, write a function to just use the negative Euclidean distance (as larger values are more similar). **Report the confusion matrix and accuracy for your results in your write-up. Can you comment in your writeup on whether the results are better or worse than classical BoW - why do you think that is?**

References

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

[2] 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.

[3] Jian xiong Xiao, J. Hays, K. Ehinger, A. Oliva, and A. Torralba. Sun database: Large-scale scene recognition from abbey to zoo. *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3485–3492, 2010.