16-720B Computer Vision: Homework 1
Spatial Pyramid Matching for Scene Classification

Instructors: Simon Lucey
TAs: Nate Chodosh, Allie Chang, Akshita Mittel, Gaurav Mittal, Chengqian Che, Purna Sowmya Munukutla
Due: September 26 at 11:59pm

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/Hints

1. Please pack your code into a single file named <**AndrewId**>**.zip**, see the complete submission checklist at the end. And submit your pdf file to gradescope(link?).
2. All questions marked with a **Q** require a submission.
3. **For the implementation part, please stick to the headers, variable names, and file conventions provided. You will lose marks if you don't.**
4. **Start early!** This homework will take a long time to complete.
5. **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.
6. Use relative paths with respect to the working directory.
7. If you have any questions or need clarifications, please post in Piazza or visit the TAs during the office hours.

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, the BoW representation has also been the subject of a great deal of study aimed at improving it, and you will see a large number of approaches that remain in the spirit of bag-of-words 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 Figure. 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.

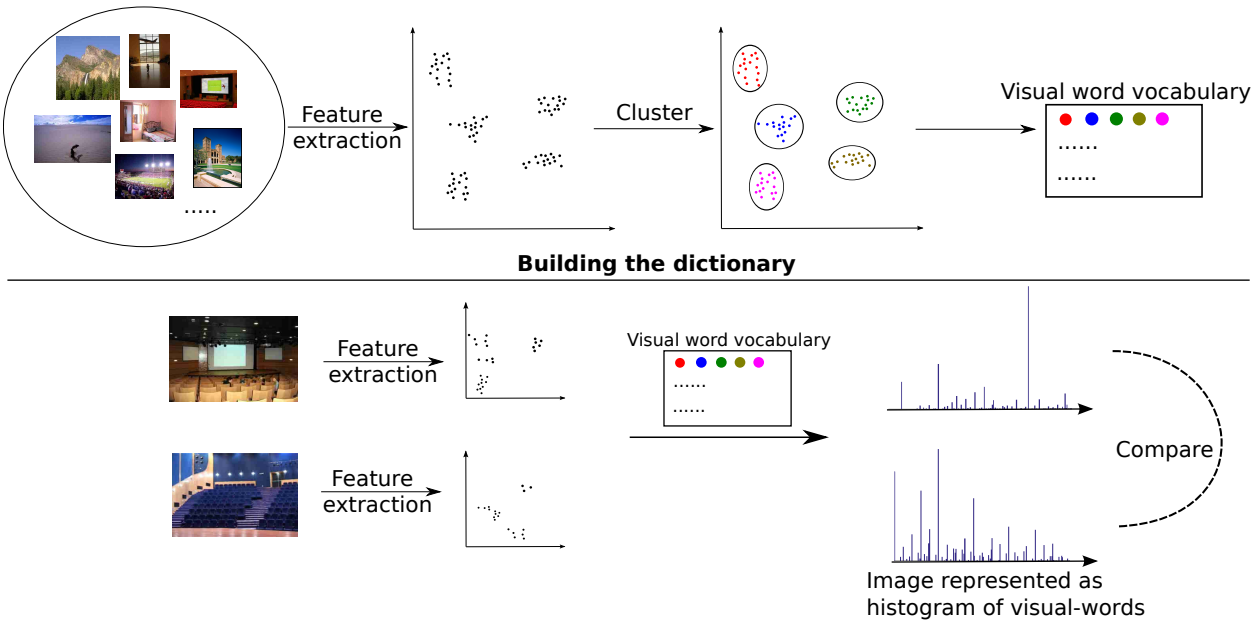

Figure 2: An overview of the bags-of-words approach to be implemented in the homework. Given the training set of images, the visual features of the images are extracted. In our case, we will use the filter responses of the pre-defined filter bank as the visual features. The visual words, *i.e.* dictionary, 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

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 at:

[1]This homework aims at being largely self-contained; however, reading the listed papers (even without trying to truly understand them) is likely to be helpful.

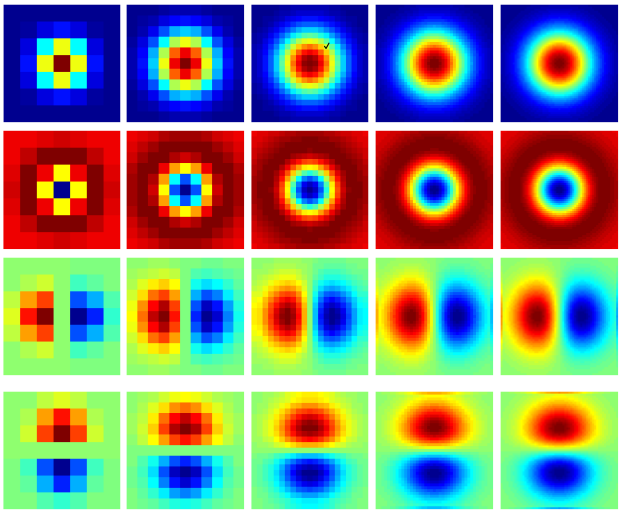

Figure 3: The provided multi-scale filter bank

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`

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

- first, take responses of a filter bank on images and build a dictionary of visual words;
- then, learn a model for the visual world based on the bag of visual 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. 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.

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 python3.5 and pytorch 0.4.1.

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 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 `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. 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 properties do each of the filter functions (See Figure 3) pick up? You should group the filters into broad categories (*e.g.* all the Gaussians). Also, why do we need multiple scales of filter responses? Answer in your write-up.

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

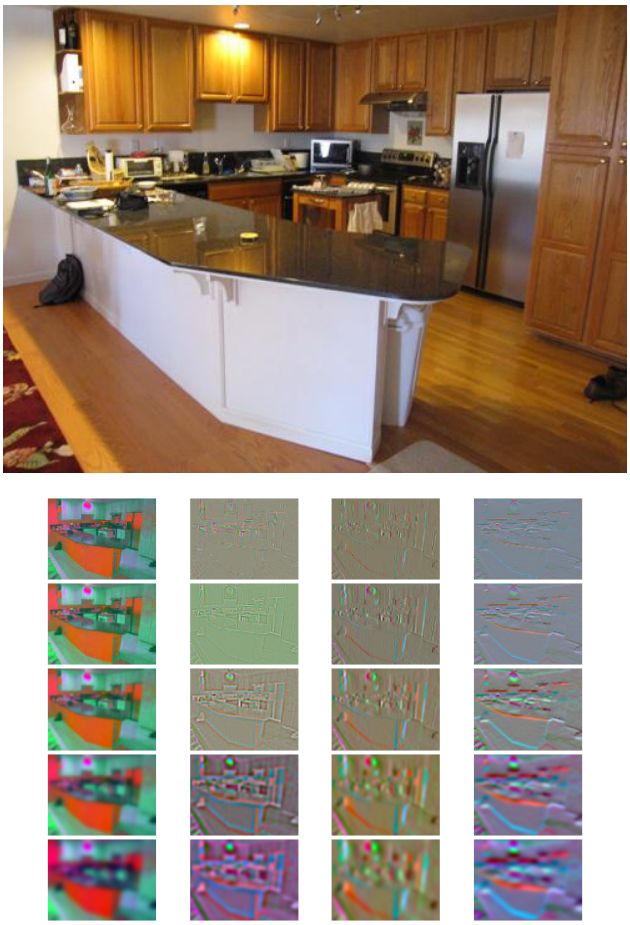

Figure 4: An input image and filter responses for all of the filters in the filter bank. (a) The input image (b) The filter responses of Lab image corresponding to the filters in Figure. 3

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 duplicated them into three channels using the command `repmat`. 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.) 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.

Apply all 20 filters on a sample image, and visualize as a image collage (as shown in Figure 4). 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 fixed-size. Instead of using all of the filter responses (**that can exceed the memory capacity of your computer**), you will use responses at α random pixels[3]. 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, 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 (10 points): 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()` – we have prepared this part of the code for you. 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

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(image,dictionary)
```

and return `wordmap`, a matrix with the same width and height as *image*, where each pixel in `wordmap` is assigned the closest visual word of the filter response at the respective pixel in `image`. 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 three wordmaps of three images from any one of the category and submit in the write-up along with their original RGB image. Also, provide your comments about the visualization. They should look similar to the ones in Figure 5.

[3] Try using `numpy.random.permutation()`.

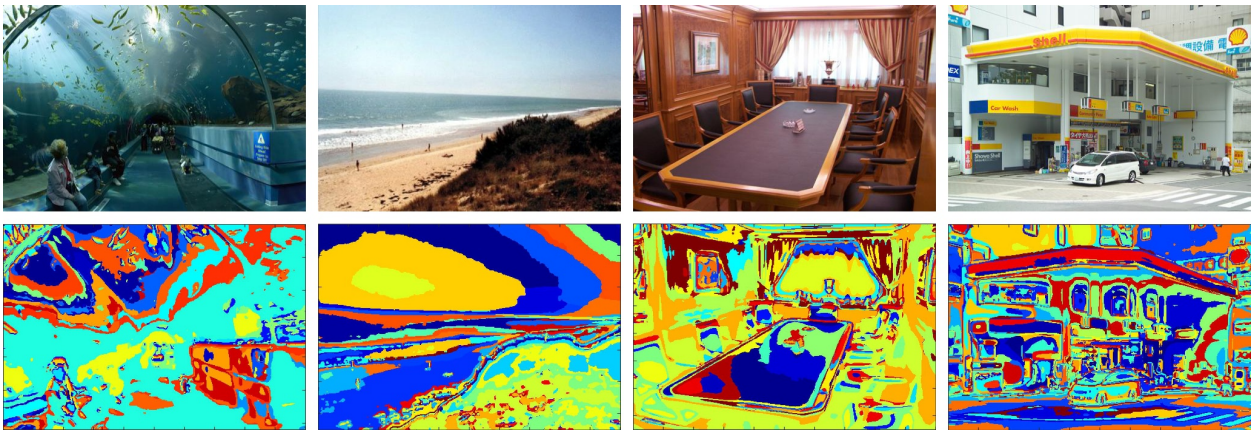

Figure 5: Visual words over images. You will use the spatially un-ordered distribution of visual words in a region (a bag of visual words) as a feature for scene classification, with some coarse information provided by spatial pyramid matching [4]

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 (colloquially, "name") the scene where the image was taken.

Traditional classification problems follow two phases: training and testing. During 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 [3].

The 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 approach. 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(wordmap,dict_size)`

that extracts the histogram[4] 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.

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

As output, the function will return `hist`, a `dict_size` histogram that is L_1 normalized, (*i.e.*, the sum equals 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

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 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 (in [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 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. 6). Note that the L_1 norm (absolute values of all dimensions summed up together) for the final vector is 1.

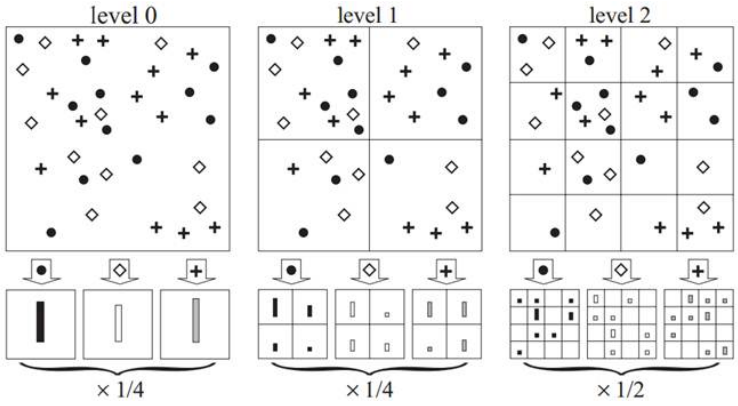

Figure 6: **Spatial Pyramid Matching:** From [4]. Toy example of a pyramid for L = 2. The image has three visual words, indicated by circles, diamonds, and crosses. We subdivide the image at three different levels of resolution. For each level of resolution and each channel, we count the features that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

Q2.2 (15 points): Create a function `getImageFeaturesSPM` that form a multi-resolution representation of the given image.

```
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 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 (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` 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
auditorium	baseball_field	desert	highway	kitchen	laundromat	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(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()`.

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 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`.

Q2.5 (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%.

2.6 Find the failed cases

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): List some of these classes/samples and discuss why they are more difficult in your write-up.

3 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). or later installed.

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).

3.1 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.

Q3.1 (25 points): We want to extract out deep features corresponding to the convolutional layers of the VGG-16 network.

To load the network, add 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`, use the Scipy convolution function `scipy.ndimage.convolve()` with for loops to cycle through the filters and channels. 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.

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.

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.

3.2 Building a Visual Recognition System: Revisited

We want to compare how useful deep features are in a visual recognition system.
Q3.2 (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. These functions should follow a very similar pipeline to those in the previous questions, so you are free to reuse much of the code.

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?

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.

- `network_layers.py`: function definitions for implementing deep network layers.
- `deep_recog.py`: function definitions for building a visual recognition system using deep features.
- `util.py`: utility functions
- `main.py`: main function for running the system

The data folder contains:

- `data/`: a directory containing `.jpg` images from SUN database.
- `data/train_data.npz`: a `.npz` file containing the training set.
- `data/test_data.npz`: a `.npz` file containing the test set.
- `data/vgg16_list.npy`: a `.npy` file with the weights of VGG-16.

5 HW1 Submission Checklist

The assignment should be submitted to Canvas. The writeup should be submitted as a pdf file named <**AndrewId**>**_hw1.pdf**. The code should be submitted as a zip file named <**AndrewId**>**.zip**. By extracting the zip file, it should have the following files in the structure defined below. (Note: Missing to follow the structure will incur huge penalty in scores).

When you submit, remove the folder `data/`, 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 .py files inside `code` directory >
  - <`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.